Lightweight Verification of Separate Compilation

Jeehoon Kang
Seoul National University, South Korea
jeehoon.kang@sf.snu.ac.kr

Yoonseung Kim
Seoul National University, South Korea
yoonseung.kim@sf.snu.ac.kr

Chung-Kil Hur *
Seoul National University, South Korea
gil.hur@sf.snu.ac.kr

Derek Dreyer
MPI-SWS, Germany
dreyer@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

Major compiler verification efforts, such as the CompCert project, have traditionally simplified the verification problem by restricting attention to the correctness of whole-program compilation, leaving open the question of how to verify the correctness of separate compilation. Recently, a number of sophisticated techniques have been proposed for proving more flexible, compositional notions of compiler correctness, but these approaches tend to be quite heavyweight compared to the simple "closed simulations" used in verifying whole-program compilation. Applying such techniques to a compiler like CompCert, as Stewart *et al.* [17] have done, involves major changes and extensions to its original verification.

In this paper, we show that if we aim somewhat lower—to prove correctness of separate compilation, but only for a *single* compiler—we can drastically simplify the proof effort. Toward this end, we develop several lightweight techniques that recast the compositional verification problem in terms of whole-program compilation, thereby enabling us to largely reuse the closed-simulation proofs from existing compiler verifications. We demonstrate the effectiveness of these techniques by applying them to CompCert 2.4, converting its verification of whole-program compilation into a verification of separate compilation in less than two person-months. This conversion only required a small number of changes to the original proofs, and uncovered two compiler bugs along the way. The result is SepCompCert, the first verification of separate compilation for the full CompCert compiler.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

Keywords Compositional compiler verification, separate compilation, CompCert

* Corresponding author.

1. Introduction

Over ten years ago, Xavier Leroy initiated the CompCert project [10, 11], a landmark effort that resulted in the first *realistic verified compiler*. The CompCert compiler [4] is *realistic* in the sense that it "could realistically be used in the context of production of critical software". In particular, it compiles a significant subset of the C language down to assembly, and it performs a number of common and useful optimizations. It is *verified* in the sense that it "is accompanied by a machine-checked proof [in Coq] of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program." As such, CompCert guarantees that program analyses and verifications performed on its input carry over soundly to its machine-level output. It has served as a fundamental building block in academic work on end-to-end verified software [1], as well as receiving significant interest from the avionics industry [16].

There is, however, a key dimension in which the verification of CompCert is not realistic—namely that, for simplicity, it only establishes the correctness of *whole-program* compilation: if CompCert is used to compile a self-contained C program consisting of a single file, then the output of CompCert preserves the semantics of that program. But clearly this does not correspond to what many clients of a verified compiler would expect. For example, it is often essential in practice to be able to compile a client module separately from the many standard libraries it depends on, yet be assured that linking the resulting binaries together will result in an executable that preserves the semantics of the linked source modules. Furthermore, it is commonplace for different modules in a program to be compiled with different sets of optimization passes turned on. Although the CompCert compiler does indeed support such forms of separate compilation, its verification statement says nothing about them.

The technical reason for this limitation is that it makes it possible for the CompCert verification to be carried out straightforwardly using *closed simulations*: simulations between closed (*i.e.,* self-contained, executable) programs. For each pass of the compiler, the output of the pass is shown to simulate the input of the pass, assuming and preserving whatever invariant the verifier wishes to impose on the relation between the states of the input and output. Working with closed simulations simplifies life in two ways: (1) the simulation proof for each pass can rely on whatever state invariant it chooses, independent of what invariants are used in other passes, and (2) these independent simulations collectively imply the end-to-end correctness of the whole compiler (this is sometimes called "vertical compositionality"). However, closed simulations are by definition simulations over whole program states, thus seemingly confining their applicability to the verification of whole-program compilation.

There has consequently been a great deal of work in the past several years attempting to prove compiler correctness *without* the whole-program restriction. Indeed, it turns out that even specifying, let alone verifying, when separate compilation is "correct"—often referred to as *compositional compiler correctness* [2]—is non-trivial, and has sparked a variety of interesting proposals involving technically sophisticated techniques, such as Kripke logical relations [6], multi-language semantics [13], and parametric simulations [7, 12]. All these approaches aim to achieve a highly flexible form of compositionality, guaranteeing for instance that the results of multiple different verified compilers can be correctly linked together, and that it is safe to link those results with hand-written assembly code.

It seems, however, that achieving such flexibility comes at the expense of significant complication to the proof method. For example, Perconti and Ahmed's approach [13] involves constructing logical relations over a multi-language semantics encompassing all languages used in a compiler, a considerable departure from CompCert-style verification. Neis *et al.*'s method [12] employs a novel notion of "parametric inter-language simulations (PILS)", whose proof of the aforementioned "vertical compositionality" is highly involved [8], whereas for closed simulations it is trivial. In the context of CompCert, Stewart *et al.* recently developed Compositional CompCert [17], a re-engineering of CompCert to support verified separate compilation, along with the ability to link C modules with assembly modules. Their approach, however, relies on a novel notion of "structured simulation" on top of a multi-language "interaction semantics" [3], which is different enough from the closed simulations employed in the CompCert verification that it required significant changes and extensions to the original proofs.

In this paper, we ask the question: If we aim somewhat lower, can we do a lot better? That is, if we pursue a more restricted notion of compositional correctness than prior work has done, can we develop a much simpler, more lightweight proof method that enables us to *reuse* existing verifications of whole-program compilation as much as possible instead of rewriting them?

Indeed, we can. In particular, we restrict attention here to verifying separate compilation for a *single* compiler. Our goal is to establish that, when different modules in a program are compiled separately by the *same* verified compiler, the linking of the resulting assembly modules preserves the semantics of the linking of the original source modules. Within the scope of this more modest but still important goal, we develop simple and effective techniques for verifying two levels of compositional correctness:

- **Compositional Correctness Level A:** Correctness of compilation is preserved when linking modules that were compiled with *the same exact compiler*.
- **Compositional Correctness Level B:** Correctness of compilation is preserved when linking modules that were compiled with the same compiler, *but possibly with different optimizations—i.e.,* different modules may be compiled with different optimization passes turned on.

Level B correctness is stronger than Level A, and correspondingly requires somewhat (but not much) more work to prove.

The key idea spanning both levels is to formulate a compositional correctness statement about a single *module* M in terms of a "contextual" correctness statement about how M behaves when linked with arbitrary other modules to form a complete program. The latter has the advantage of being a statement about closed (whole) programs, and as such, we can prove it using the kind of simple, closed-simulation-style proofs employed in traditional verifications of whole-program compilation. This in turn enables us to significantly reuse existing compiler proofs. We believe the lightweight nature of our techniques—and the consequent ease of adapting existing verifications to use them—will make them a highly attractive option for compiler verifiers.

We demonstrate the effectiveness of our techniques by applying them to CompCert 2.4. In less than two person-months total, we adapted the existing CompCert verification to support both Level A and Level B compositional correctness, and much of that time was spent trying to understand the original CompCert proof. **The result of this effort is SepCompCert, the first verification of separate compilation for the full CompCert compiler**. The SepCompCert verification (available online [15]) is mostly the same as the original CompCert verification, and is only 2% (for Level A) or 3% (for Level B) larger than the original verification in terms of lines of Coq. Furthermore, in the process of porting CompCert to SepCompCert, we uncovered two bugs in CompCert: one in an invalid axiom, and one (in CompCert's "value analysis") that was outside the scope of CompCert's original verification because it only showed up in the presence of separate compilation. These have been confirmed and subsequently fixed in the latest version of the compiler.[1]

The remainder of the paper is structured as follows. In §2, we give a high-level overview of our new techniques, which are presented in detail in the subsequent sections in the context of our CompCert adaptation. In §7, we conclude with a comparison to related work.

2. High-Level Overview

We begin by briefly reviewing CompCert's correctness statement for whole-program compilation, as well as its closed-simulation verification method (§2.1). We then explain our Level A and Level B notions of compositional correctness and our techniques for establishing them (§2.2 and §2.3). We also briefly give some intuition as to why it is easy to adapt CompCert's verification to employ our new techniques, but we leave a more thorough explanation of this adaptation to subsequent sections. Throughout the section, we keep the presentation semi-formal, abstracting away unnecessary detail to get across the main ideas.

2.1 CompCert's Whole-Program Compilation Correctness

End-to-End Correctness Roughly speaking, the correctness result of CompCert can be understood to assert the following. Suppose `s.c` is a "source" file (in C), `t.asm` is a "target" file (in assembly), and $\mathcal{C}$ is a verified compiler (represented as a function from C files to assembly files).

$$\frac{\mathcal{C}(\texttt{s.c}) = \texttt{t.asm} \qquad s = \mathrm{load}(\texttt{s.c}) \qquad t = \mathrm{load}(\texttt{t.asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

If `t.asm` is the result of compiling `s.c` with $\mathcal{C}$, then executing `t.asm` according to assembly semantics will result in a subset of the behaviors one could observe from executing `s.c` according to C semantics. (We write $s = \mathrm{load}(\texttt{s.c})$ to denote the *machine state* that results from loading `s.c` into memory, $\mathrm{Behav}(s)$ to denote the observable behaviors of the execution of s, and analogously for t and `t.asm`.) Hence, we say that `t.asm`, the target-level output of $\mathcal{C}$, *refines* its source-level input, `s.c`.

Per-Pass Correctness To verify whole-program compilation correctness for the compiler $\mathcal{C}$, we verify each pass of $\mathcal{C}$ independently. Specifically, for each pass (transformation) $\mathcal{T}$ from language L_1 to L_2—where the L_i's may be C, assembly, or some intermediate languages—we show the following:

$$\frac{\mathcal{T}(\texttt{s.l1}) = \texttt{t.l2} \qquad s = \mathrm{load}(\texttt{s.l1}) \qquad t = \mathrm{load}(\texttt{t.l2})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

[1] The latest version of CompCert is 2.5. It was released on June 12, 2015, after we had already completed our verification of SepCompCert. See §7 for discussion of what would be involved in porting SepCompCert to handle the new features of CompCert 2.5.

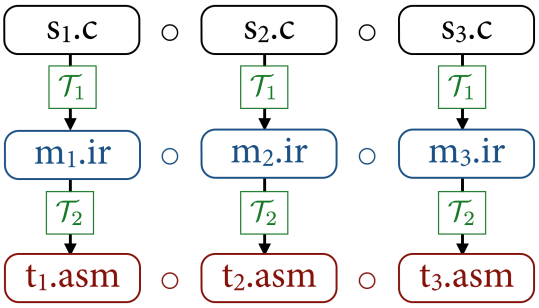

Figure 1. Proving Level A correctness. Here, we show the case of three separately-compiled modules and two compiler passes, $\mathcal{T}_1$ and $\mathcal{T}_2$. Since the compilations march in lock step, we can verify each pass as applied to all modules simultaneously.

That is, given the input `s.l1` and output `t.l2` of the $\mathcal{T}$ transformation, we show that the behaviors of `t.l2` are contained within those of `s.l1`. Since subset inclusion is transitive, it easy to see that the proofs of the constituent passes of $\mathcal{C}$ compose to establish the whole-program correctness of $\mathcal{C}$ as a whole.

Verifying Per-Pass Correctness Now how does one actually prove the verification condition for each individual pass? The standard approach taken by CompCert is to use (closed) simulations. Informally, we will say that a *simulation* R is a relation between running programs (*i.e.,* machine states) in L_1 and L_2 such that, if $(s,t) \in R$, then the behaviors one observes while stepping through the execution of t are matched by corresponding observable behaviors in the execution of s. One can think of R as imposing an invariant, which describes (and connects) the possible machine states of the source and target programs, and which must be maintained as the programs execute. We leave further details about simulations until later in the paper; suffice it to say that they satisfy the following "adequacy" property:

$$\frac{R \text{ is a simulation} \qquad (s,t) \in R}{\text{Behav}(s) \supseteq \text{Behav}(t)}$$

Thus, to establish the verification condition for pass $\mathcal{T}$, it suffices to exhibit a simulation R that relates $\text{load}(\texttt{s.l1})$ and $\text{load}(\texttt{t.l2})$.

Note: The verification approach described above, relying on "backward" simulations, is something of an oversimplification of what CompCert actually does. In fact, to make the proofs more convenient, CompCert uses a mixture of forward and backward simulations. We gloss over this point here because it is orthogonal to our high-level story, but we will return to it at the end of §3.

2.2 SepCompCert's Compositional Correctness Level A

End-to-End Correctness For Level A, we aim to show that if we separately compile n different C modules $(\texttt{s}_1\texttt{.c}, \ldots, \texttt{s}_n\texttt{.c})$ using the same exact verified compiler $\mathcal{C}$, producing n assembly modules $(\texttt{t}_1\texttt{.asm}, \ldots, \texttt{t}_n\texttt{.asm})$, then the assembly-level linking of the $\texttt{t}_i$'s will refine the C-level linking of the $\texttt{s}_i$'s. Formally:

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots n\}.\, \mathcal{C}(\texttt{s}_i\texttt{.c}) = \texttt{t}_i\texttt{.asm} \\ s = \text{load}(\texttt{s}_1\texttt{.c} \circ \ldots \circ \texttt{s}_n\texttt{.c}) \qquad t = \text{load}(\texttt{t}_1\texttt{.asm} \circ \ldots \circ \texttt{t}_n\texttt{.asm})\end{array}}{\text{Behav}(s) \supseteq \text{Behav}(t)}$$

Here, $\circ$ represents simple *syntactic* linking, *i.e.,* essentially concatenation of files (plus checks to make sure that externally declared variables/functions have the expected types). See §4 for further details about syntactic linking.

Per-Pass Correctness To prove that compiler $\mathcal{C}$ satisfies Level A compositional correctness, we again want to reduce the problem to one of verifying the individual passes of $\mathcal{C}$. The key idea here, as illustrated in Figure 1, is that since we know that the source modules are all compiled via the exact same sequence of passes, we can verify their compilations in lock step. In other words, it suffices to verify that, for each pass $\mathcal{T}$ from L_1 to L_2, the following holds:

$$\frac{\begin{array}{c}\forall i \in \{1 \ldots n\}.\, \mathcal{T}(\texttt{s}_i\texttt{.l1}) = \texttt{t}_i\texttt{.l2} \\ s = \text{load}(\texttt{s}_1\texttt{.l1} \circ \ldots \circ \texttt{s}_n\texttt{.l1}) \qquad t = \text{load}(\texttt{t}_1\texttt{.l2} \circ \ldots \circ \texttt{t}_n\texttt{.l2})\end{array}}{\text{Behav}(s) \supseteq \text{Behav}(t)}$$

As before, these per-pass correctness results can be transitively composed to immediately conclude end-to-end correctness of $\mathcal{C}$.

Verifying Per-Pass Correctness So how do we prove this Level A per-pass correctness condition? Assuming that we have already proven whole-program per-pass correctness and are trying to port the proof over, there are two cases.

Trivial case: Many compiler passes are inherently compositional, transforming the code of each module independently, *i.e.,* in a way that is agnostic to the presence of other modules. Put another way, such compiler passes commute with linking:

$$\mathcal{T}(\texttt{s}_1\texttt{.l1}) \circ \ldots \circ \mathcal{T}(\texttt{s}_n\texttt{.l1}) = \mathcal{T}(\texttt{s}_1\texttt{.l1} \circ \ldots \circ \texttt{s}_n\texttt{.l1})$$

If this commutativity property holds for a pass $\mathcal{T}$, then Level A per-pass correctness becomes a trivial corollary of whole-program per-pass correctness, where we instantiate the `s.l1` from §2.1 with $\texttt{s}_1\texttt{.l1} \circ \ldots \circ \texttt{s}_n\texttt{.l1}$. In verifying Level A correctness for CompCert 2.4, we found that 13 of its 19 passes fell into this trivial case.

Non-trivial case: If the trivial commutativity argument does not apply, then there is some new work to do to port a proof of whole-program per-pass correctness to Level A per-pass correctness.

However, at least for CompCert, we found it very easy to perform this adaptation. Why? First of all, since Level A correctness assumes that all modules in the program are transformed in the same way, we can essentially reuse the simulation relation R for pass $\mathcal{T}$ that was used in the original CompCert verification.

We do, however, have to worry about the soundness of the program analyses that the compiler performs, because the correctness of the compiler rests to a large extent on the correctness of these analyses. To prove Level A correctness of these analyses, we must prove that they remain sound even when they only have access to a single module in the program rather than the whole program. Intuitively, this should follow easily if: (1) the analyses have been proven sound under the assumption that they are fed the whole program (CompCert has done that already), *and* (2) the analyses are *monotone*, meaning that they only become *more conservative* when given access to a smaller fragment of the program (as happens with separate compilation).

In adapting CompCert to Level A correctness, the main work was therefore in verifying that its program analyses were indeed monotone. This was largely straightforward, with one exception: the "value analysis" employed by several optimizations was not monotone. It made an assumption about variables declared as `extern const`, which was valid for whole-program compilation, but not in the presence of separate compilation. As we explain in detail in §4, this manifested itself as a bug in constant propagation when linking separately-compiled files. After we fixed this bug in value analysis, monotonicity became straightforward to show, and thus so did Level A correctness (for the remaining 6 passes that did not fall into the trivial case).

2.3 SepCompCert's Compositional Correctness Level B

End-to-End Correctness The CompCert compiler performs several key optimizations at the level of its RTL intermediate language (*i.e.,* they are transformations from RTL to RTL). For Level B, we would like to strengthen Level A correctness by allowing each source module $\texttt{s}_i$ to be compiled by a *different* compiler $\mathcal{C}_i$. However, the differences we permit between the $\mathcal{C}_i$'s are restricted: they

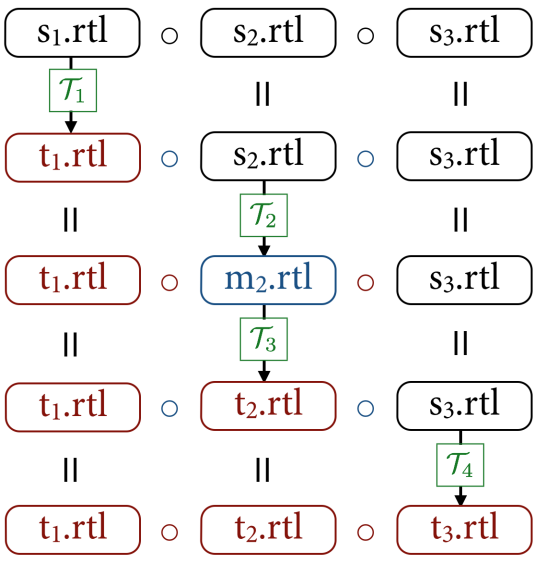

Figure 2. Proving Level B correctness. Here, we show only the RTL-level passes (for other passes, it is the same as Level A). Since each module is compiled with different optimization passes, we verify each pass applied to exactly one module, while simultaneously the other modules undergo the identity transformation.

may differ only in which optimization passes they apply at the RTL level. Given this restriction on the $\mathcal{C}_i$'s, the Level B correctness statement is the same as the Level A correctness statement except for the replacement of $\mathcal{C}$ with $\mathcal{C}_i$:

$$\frac{\forall i \in \{1 \ldots n\}.\, \mathcal{C}_i(\mathtt{s}_i.\mathtt{c}) = \mathtt{t}_i.\mathtt{asm} \qquad s = \mathrm{load}(\mathtt{s}_1.\mathtt{c} \circ \ldots \circ \mathtt{s}_n.\mathtt{c}) \qquad t = \mathrm{load}(\mathtt{t}_1.\mathtt{asm} \circ \ldots \circ \mathtt{t}_n.\mathtt{asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

Per-Pass Correctness To verify the above correctness statement, we yet again want to reduce it to verifying the individual passes of $\mathcal{C}$. For all passes besides the RTL-level optimizations, we can verify per-pass correctness exactly as in Level A, since all the $\mathcal{C}_i$'s must perform these same passes in the same order. However, for the RTL optimizations, we must do something different because at the RTL level the various $\mathcal{C}_i$'s do not all march in lock step.

The key idea for handling the RTL optimizations, as illustrated in Figure 2, is to pad the $\mathcal{C}_i$'s with extra dummy identity passes (which do not affect their end-to-end functionality) so that, whenever one compiler is performing an RTL optimization pass, the other compilers will for that step perform an identity transformation. Thus, we first verify the RTL passes of $\mathcal{C}_1$ in parallel with identity passes for the other compilers, then verify the RTL passes of $\mathcal{C}_2$ in parallel with identity passes for the other compilers, and so on. For this to work, the Level B per-pass correctness statement (for optimization passes $\mathcal{T}$ from RTL to RTL) must be updated as follows:

$$\frac{\mathcal{T}(\mathtt{s.rtl}) = \mathtt{t.rtl} \qquad s = \mathrm{load}(\mathtt{u}_1.\mathtt{rtl} \circ \ldots \circ \mathtt{u}_m.\mathtt{rtl} \circ \mathtt{s.rtl} \circ \mathtt{v}_1.\mathtt{rtl} \circ \ldots \circ \mathtt{v}_n.\mathtt{rtl}) \qquad t = \mathrm{load}(\mathtt{u}_1.\mathtt{rtl} \circ \ldots \circ \mathtt{u}_m.\mathtt{rtl} \circ \mathtt{t.rtl} \circ \mathtt{v}_1.\mathtt{rtl} \circ \ldots \circ \mathtt{v}_n.\mathtt{rtl})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

One can view this notion of per-pass correctness as being essentially a form of *contextual refinement*: the output of $\mathcal{T}$ must refine its input when linked with a context consisting of some arbitrary other RTL modules (the $\mathtt{u}_i$'s and $\mathtt{v}_j$'s). If we can prove this, it should be clear from Figure 2 how the per-pass proofs link up transitively.

Verifying Per-Pass Correctness So how do we prove this contextual refinement? Unlike for Level A, we cannot simply reuse the existing simulation R from the whole-program per-pass correctness proof for $\mathcal{T}$, because R is not necessarily reflexive and thus does not necessarily relate the execution of code from $\mathtt{u}_i.\mathtt{rtl}$ (or $\mathtt{v}_j.\mathtt{rtl}$) with itself. Instead, we must use an amended simulation R', which accounts for two possibilities: either we are executing code from $\mathtt{s}$ on one side of the simulation and code from $\mathtt{t}$ on the other, in which case the proof that R' is indeed a simulation proceeds essentially as did the proof that R was a simulation; or we are executing code from $\mathtt{u}_i.\mathtt{rtl}$ (or $\mathtt{v}_j.\mathtt{rtl}$), in which case both sides of the simulation are executing exactly the same RTL instructions.

In principle, the latter case could involve serious new proof effort. However, at least for CompCert, we found that in fact the substance of this new part of the proof was hiding in plain sight within the original CompCert verification! The reason, intuitively, is that RTL-to-RTL optimization passes are rarely unconditional transformations: their output typically only differs from their input *when certain conditions (e.g. determined by a static analysis) hold*, and since these conditions do not always hold, these passes may end up leaving any given input instruction unchanged. To account for this possibility, the original CompCert verification must therefore already prove that arbitrary RTL instructions simulate themselves. Consequently, in porting CompCert 2.4 to Level B compositional correctness, we were able to simply extract and compose (essentially, copy-and-paste) these micro-simulation proofs into the simulation proof for the latter part of R'.

3. Constant Propagation in CompCert

To flesh out the technical details of how we adapted CompCert to SepCompCert, we will use constant propagation, one of CompCert's RTL-level optimizations, as a running example. In this section, we will begin by reviewing the RTL language itself, how constant propagation works, and how CompCert verifies it.

3.1 RTL Syntax and Semantics

We start with the syntax and semantics of CompCert's register transfer language (RTL), the compiler's internal language where constant propagation takes place. For presentation purposes, we simplify the language a bit by removing types and other unnecessary details.

Syntax The syntax of the CompCert's RTL is given in Figure 3. Programs are just a list of global declarations, which consist of (*i*) declarations of external variables and functions provided by different compilation units, and (*ii*) definitions of variables and functions provided by the current compilation unit. For global variable declarations and definitions, we also specify a (positive) integer number denoting the size of the declared block in bytes.

Function declarations only contain the function signature, which is a list of parameters, but function definitions additionally contain a list of local registers, the size of their stack frame, and the code. The code is essentially a control-flow graph of three-address code: it is represented as a mapping from node identifiers to instructions, where instructions either do some local computation (*e.g.,* write a constant to a register, or perform some arithmetic computation), load from a memory address, store to memory, do a comparison, call a function, or exit the function and return a result. Each instruction also stores the node identifier(s) of its successor instruction(s).

Throughout we assume that programs satisfy some basic well-formedness properties: there cannot be multiple definitions for the same global variable, declarations and definitions of the same variable should have matching signatures, and the parameter and local variable lists for each function do not have duplicate entries.

$$
\begin{array}{lll}
\mathit{Prog} & ::= \overline{\mathit{Decl}} & \\
\mathit{Decl} & ::= \texttt{extern}\ [\texttt{const}]\ \mathit{Id}[\mathit{Int}] & \text{// External Variable} \\
& \mid\ [\texttt{const}]\ \mathit{Id}[\mathit{Int}] := \{\ \overline{\mathit{GVal}}\ \} & \text{// Variable} \\
& \mid\ \texttt{extern}\ \mathit{Id}\ \mathit{FSig} & \text{// External Function} \\
& \mid\ \mathit{Id}\ \mathit{FDef} & \text{// Function} \\
\mathit{FSig} & ::= (\overline{\mathit{Reg}}) & \text{// Function Signature} \\
\mathit{FDef} & ::= (\overline{\mathit{Reg}})\ \{\ \overline{\mathit{Reg}};\ \texttt{sp}[\mathit{Int}]\ ;\ \mathit{Code}\ \} & \text{// Function Definition} \\
\mathit{Code} & ::= \overline{\mathit{NId} : \mathit{Instr}} & \\
\mathit{Instr} & ::= \mathit{Reg} := \mathit{FVal}\quad \texttt{jmp}\ \mathit{NId} & \text{// Immediate Value} \\
& \mid\ \mathit{Reg} := \mathit{op}\ \overline{\mathit{Reg}}\quad \texttt{jmp}\ \mathit{NId} & \text{// Operation} \\
& \mid\ \mathit{Reg} := \mathit{FVal}[\mathit{Int}]\quad \texttt{jmp}\ \mathit{NId} & \text{// Load} \\
& \mid\ \mathit{FVal}[\mathit{Int}] := \mathit{Reg}\quad \texttt{jmp}\ \mathit{NId} & \text{// Store} \\
& \mid\ \mathit{cond\text{-}op}\ \overline{\mathit{Reg}}\ ?\ \texttt{jmp}\ \mathit{NId} : \texttt{jmp}\ \mathit{NId} & \text{// Conditional} \\
& \mid\ \mathit{Reg} := \mathit{FVal}(\overline{\mathit{Reg}})\quad \texttt{jmp}\ \mathit{NId} & \text{// Call} \\
& \mid\ \texttt{return}\ \mathit{Reg} & \text{// Return} \\
& \mid\ \ldots & \\
\mathit{GVal} & ::= \mathit{Id} \mid \mathit{Int} \mid \texttt{undef} & \\
\mathit{FVal} & ::= \mathit{GVal} \mid \mathit{Reg} \mid \texttt{sp} & \\
\mathit{Int} & ::= \text{the set of 32-bit integers} & \\
\mathit{Id} & ::= \text{the set of identifiers for variables and functions} & \\
\mathit{Reg} & ::= \text{the set of register names} & \\
\mathit{NId} & ::= \text{the set of node labels} &
\end{array}
$$

Figure 3. RTL syntax.

$$
\begin{array}{lll}
\mathit{Mem} & \stackrel{\text{def}}{=} \{\ (nb, bs) \mid nb \in \mathit{BId} \wedge bs \in \mathit{BId} \rightharpoonup_{\text{fin}} \mathit{Block}\ \} & \\
\mathit{Block} & \stackrel{\text{def}}{=} \{\ (p, n, c) \mid p \in \{\texttt{valid}, \texttt{freed}\} \wedge n \in \mathbb{N} \wedge c \in \mathit{Val}^n\} & \\
\mathit{Val} & \stackrel{\text{def}}{=} \mathit{Int} \uplus \mathit{Addr} \uplus \{\texttt{undef}\} & \\
\mathit{Addr} & \stackrel{\text{def}}{=} \{\ (l, i) \in \mathit{BId} \times \mathit{Int}\ \} & \\
\mathit{GEnv} & \stackrel{\text{def}}{=} \{\ (g, d) \mid g \in \mathit{Id} \rightharpoonup_{\text{fin}} \mathit{BId} \wedge & \\
& \qquad d \in \mathit{BId} \rightharpoonup_{\text{fin}} \mathit{FSig} \uplus \mathit{FDef}\ \} & \\
\mathit{State} & \stackrel{\text{def}}{=} \mathit{IState} \uplus \mathit{CState} \uplus \mathit{RState} & \\
\mathit{IState} & \stackrel{\text{def}}{=} \{\ \texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \mid & \\
& \qquad m \in \mathit{Mem} \wedge s \in \overline{\mathit{StkFrm}} \wedge fd \in \mathit{FDef} \wedge & \\
& \qquad sp \in \mathit{Addr} \wedge pc \in \mathit{NId} \wedge rs \in \mathit{Reg} \rightharpoonup_{\text{fin}} \mathit{Val}\ \} & \\
\mathit{CState} & \stackrel{\text{def}}{=} \{\ \texttt{cst}\ m\ s\ fds\ vs \mid m \in \mathit{Mem} \wedge s \in \overline{\mathit{StkFrm}} \wedge & \\
& \qquad fds \in \mathit{FDef} \uplus \mathit{FSig} \wedge vs \in \overline{\mathit{Val}}\ \} & \\
\mathit{RState} & \stackrel{\text{def}}{=} \{\ \texttt{rst}\ m\ s\ v \mid m \in \mathit{Mem} \wedge s \in \overline{\mathit{StkFrm}} \wedge v \in \mathit{Val}\ \} & \\
\mathit{StkFrm} & \stackrel{\text{def}}{=} \{\ (r, fd, sp, pc, rs) \mid r \in \mathit{Reg} \wedge fd \in \mathit{FDef} \wedge & \\
& \qquad sp \in \mathit{Addr} \wedge pc \in \mathit{NId} \wedge rs \in \mathit{Reg} \rightharpoonup_{\text{fin}} \mathit{Val}\ \} &
\end{array}
$$

Figure 4. RTL semantic domains.

Semantics We move on to the semantics of RTL. Figure 4 defines the necessary semantic domains.

Memory, $m \in \mathit{Mem}$, is represented as a finite collection of allocation blocks (a mapping from block identifiers to blocks), each of which is a contiguous portion of memory that may be either valid to access or already deallocated (freed) and therefore invalid to access. CompCert values, $v \in \mathit{Val}$, can be either 32-bit integers, logical addresses (pairs of a block identifier together with an offset within the block), or the special `undef` value used to represent uninitialized data. A global environment, $ge = (g, d) \in \mathit{GEnv}$, maps each global variable name to a logical block identifier, and each logical block identifier corresponding to some function's code to either the corresponding function signature for external functions or the corresponding function definition for functions defined in the program.

Next, program states can be of three kinds: normal instruction states (`ist`), call states (`cst`) just before passing control to an invoked function, and return states (`rst`), just after returning from an invoked function. Instruction states store the memory (m), the sequence of parent stack frames (s), the definition of the function whose body is currently executed (fd), the current stack pointer (sp), the program counter (pc), and the contents of the local registers (rs). Call states record the memory, the stack, and the function to be called (fds) with its arguments ($args$). The function to be called can be either an internal function, in which case we record its definition, or an external one, in which case we record its signature. Return states record just the memory, the stack, and the value that was returned by the function. A stack s is a list of stack frames, each of which records the same information as normal instruction states, except with the addition of a register name r where its return value should be stored, and minus the memory (m) and stack (s) components.

The meaning of programs is described by three definitions:

$$
\begin{array}{rcl}
\mathit{get\text{-}genv} & \in & \mathit{Prog} \to \mathit{GEnv} \\
\mathit{load} & \in & \mathit{Prog} \rightharpoonup \mathit{State} \\
\hookrightarrow & \in & \mathbb{P}(\mathit{GEnv} \times \mathit{State} \times \mathit{Event} \times \mathit{State})
\end{array}
$$

The first function, $\mathit{get\text{-}genv}(\mathit{prg})$, returns the global environment corresponding to the program: it 'allocates' the global variables of the program sequentially in blocks 1, 2, 3, and so on, and maps the blocks corresponding to function symbols to the relevant function definition or signature.

Similarly, $\mathit{load}(\mathit{prg})$ returns the initial state obtained by loading a program into memory: it initializes the memory m with the initial values of the global variables at the appropriate addresses generated by $\mathit{get\text{-}genv}(\mathit{prg})$, and returns a call state, `cst` $m\ [\,]\ fd\ [\,]$, where fd is the function definition corresponding to `main()`. Loading is a partial function because it is undefined for programs without a `main()` function.

The $\hookrightarrow$ relation is a small-step reduction relation describing how program states evolve during the computation. For clarity, we write $s \overset{\sigma}{\hookrightarrow}_{ge} s'$ instead of $(ge, s, \sigma, s') \in \hookrightarrow$. The operational semantics for RTL is fairly standard and shown in Figure 5: there is a rule for each of the various basic instructions of the language. Starting from normal instruction states, the instruction at the node pointed to by the program counter is scrutinized ($fd@pc$). Depending on what instruction is there, only one rule is applicable. The corresponding rule calculates the new values of the registers, the memory (for store instructions), and the next program counter. Calls and returns are treated a bit differently: they do not directly transition from an instruction state to the next instruction state—they go through an intermediate call/return state.

In more detail, calling a function (rule CALL) looks up the function in the global environment, evaluates its arguments, creates a new stack frame corresponding to the current instruction state, and transitions to a call state. From a call state, there are two possible execution steps. If the function to be called is internal, *i.e.,* we have its function definition $fd \in \mathit{FDef}$, rule INTERNAL-CALL applies. It allocates the necessary stack space for the called function, initializes the parameter registers with the values passed as arguments, sets the program counter to point to the first node of the called function, and moves to the appropriate instruction state of the called function. If the function to be called is external, *i.e.,* we have a function signature $fs \in \mathit{FSig}$, rule EXTERNAL-CALL goes directly to the return state, and generates an event σ indicating that it called an external function.

Conversely, returning from a function (rule RETURN) evaluates the result to be returned, deallocates the stack space used by the function, and transitions to the return state. The only possible step form a return state (rule RETURN2) then pops the top-most stack frame and transitions to a normal instruction state thereby restoring

$$\text{IMM}\quad \frac{fd@pc = (dst := src\ \texttt{jmp}\ pc') \quad rs' = rs[dst \leftarrow [\![src]\!]\,(ge, sp, rs)]}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

$$\text{OP}\quad \frac{fd@pc = (dst := op\ args\ \texttt{jmp}\ pc') \quad rs' = rs[dst \leftarrow [\![op]\!]\,(ge, sp, [\![args]\!]\,(rs))]}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

$$\text{LOAD}\quad \frac{fd@pc = (dst := src[n]\ \texttt{jmp}\ pc') \quad (l, i) = [\![src]\!]\,(ge, sp, rs) \quad rs' = rs[dst \leftarrow m[(l, i+n)]]}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

$$\text{STORE}\quad \frac{fd@pc = (dst[n] := src\ \texttt{jmp}\ pc') \quad (l, i) = [\![dst]\!]\,(ge, sp, rs) \quad m' = m[(l, i+n) \leftarrow [\![src]\!]\,(rs)]}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m'\ s\ fd\ sp\ pc'\ rs}$$

$$\text{COND}\quad \frac{fd@pc = (cond\text{-}op\ args\ ?\ \texttt{jmp}\ pc_1 : \texttt{jmp}\ pc_2) \quad b = [\![cond\text{-}op]\!]\,(ge, sp, [\![args]\!]\,(rs)) \quad pc' = b\ ?\ pc_1 : pc_2}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs}$$

$$\text{CALL}\quad \frac{fd@pc = (res := f(args)\ \texttt{jmp}\ pc') \quad (l, 0) = [\![f]\!]\,(ge, sp, rs) \quad fds' = \text{findfunc}(ge, l) \quad vs = [\![args]\!]\,(rs)}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{cst}\ m\ ((res, fd, sp, pc', rs)::s)\ fds'\ vs}$$

$$\text{RETURN}\quad \frac{fd@pc = (\texttt{return}\ r) \quad v = [\![r]\!]\,(rs) \quad m' = \text{free}(m, sp, \text{stacksize}(fd))}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{rst}\ m'\ s\ v}$$

$$\text{INTERNAL-CALL}\quad \frac{(m', l) = \text{alloc}(m, \text{stacksize}(fd)) \quad pc = \text{entrynode}(fd) \quad rs = \text{init-regs}(\text{params}(fd), vs)}{\texttt{cst}\ m\ s\ fd\ vs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m'\ s\ fd\ (l, 0)\ pc\ rs}$$

$$\text{RETURN2}\quad \frac{rs' = rs[res \leftarrow v]}{\texttt{rst}\ m\ (res, fd, sp, pc, rs)::s\ v \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc\ rs'}$$

$$\text{EXTERNAL-CALL}\quad \frac{(\sigma, v, m') \in \text{extcall-sem}(fs, ge, vs, m)}{\texttt{cst}\ m\ s\ fs\ vs \overset{\sigma}{\hookrightarrow}_{ge} \texttt{rst}\ m'\ s\ v}$$

Figure 5. Operational semantics of RTL.

the registers, program counter, and stack pointers of the calling function.

3.2 Constant Propagation in CompCert

Given a program *prg*, constant propagation walks through each function definition *fd* of the program and transforms it using the function transfun(*prg*, *fd*). This in turn runs a "value analysis" to determine which variables (whether global variables or local registers) hold a known constant value at each program point and then, based on that information, simplifies the program.

The analysis consists of two parts: (a) the global part, which detects which global variables cannot be updated (*i.e.,* those declared with the `const` qualifier), and (b) the local part, which analyzes the code of a function and calculates an abstract value for each register and stack variable. The abstract value of a variable can be either $\bot$ if the variable holds `undef`, or a constant number, or NS if the variable contains anything except for a pointer pointing into the current stack frame, or $\top$ if no more precise information is known. These abstract values form a lattice by taking the order $\bot \sqsubseteq num \sqsubseteq NS \sqsubseteq \top$.

The value analysis performs a usual traversal of the code. When calling a function, if it can be determined that no memory address can point to the current stack frame and none of the function's arguments point to the current stack frame (*i.e.,* their abstract value is at most NS), then the abstract value of the function's result is also NS, and the abstraction of the stack memory is preserved. If, however, a pointer to the current stack frame has escaped, then any information about the stack memory is forgotten.

The transformation part itself is straightforward: at each node n of the function's CFG, if the analysis has determined that a variable has a constant value at node n, then the use of that variable is replaced by the constant it holds, and the instruction is suitably simplified.

```
extern g(a,b);                          extern g(a,b);
const gv[1] := { 0 };                   const gv[1] := { 0 };

f() {                                   f() {
  x, y;                                   x, y;
  sp[4];                                  sp[4];

1: sp[0] := 0 jmp 2;                    1: sp[0] := 0 jmp 2;
2: x := 0 jmp 3;                        2: x := 0 jmp 3;
3: y := g(sp,x) jmp 4;                  3: y := g(sp,x) jmp 4;
4: sp[0] > 0 ?                          4: sp[0] > 0 ?
     jmp 5 : jmp 6;                          jmp 5 : jmp 6;
5: x > 0 ? jmp 6 : jmp 7;       ↦       5: jmp 7;
6: return gv[0];                ↦       6: return 0;
7: return y;                            7: return y;
}                                       }
```

Figure 6. Example of constant propagation.

As an example, Figure 6 shows the effect of constant propagation applied to a simple program. The program contains one internal function, `f`, which calls an external function, `g`, and three zero-initialized variables: a local variable (a register), `x`, an address-taken variable on the stack, `sp[0]`, and a global variable, `gv[0]`. After the external function call, constant propagation can safely assume that `x` $= 0$ and thereby simplify the conditional at node 5, but cannot do the same for `sp[0]` at node 4 because its address was passed to the external function and its value might therefore have changed. Further, at node 6, constant propagation notes that the global variable `gv[0]` has been declared with the `const` qualifier, and can therefore assume that `gv[0]` $= 0$.

$$prg \vdash fd \sim_{\text{fdef}} fd' \stackrel{\text{def}}{=} fd' = \text{transfun}(prg, fd)$$

$$\frac{prg \vdash fd \sim_{\text{fdef}} fd' \qquad rs \leq_{\texttt{def}} rs'}{prg \vdash (r, fd, sp, pc, rs) \sim_{\text{frame}} (r, fd', sp, pc, rs')}$$

$$\frac{}{prg \vdash [\,] \sim_{\text{stack}} [\,]} \qquad \frac{prg \vdash sf \sim_{\text{frame}} sf' \qquad s \sim_{\text{stack}} s'}{prg \vdash sf{::}s \sim_{\text{stack}} sf{::}s'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \quad s \sim_{\text{stack}} s' \quad prg \vdash fd \sim_{\text{fdef}} fd' \quad rs \leq_{\texttt{def}} rs'}{prg \vdash \texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \sim_{\text{state}} \texttt{ist}\ m'\ s'\ fd'\ sp\ pc\ rs'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \quad s \sim_{\text{stack}} s' \quad prg \vdash fd \sim_{\text{fdef}} fd' \quad args \leq_{\texttt{def}} args'}{prg \vdash \texttt{cst}\ m\ s\ fd\ args \sim_{\text{state}} \texttt{cst}\ m'\ s'\ fd'\ args'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \qquad s \sim_{\text{stack}} s' \qquad v \leq_{\texttt{def}} v'}{prg \vdash \texttt{rst}\ m\ s\ v \sim_{\text{state}} \texttt{rst}\ m'\ s'\ v'}$$

$$(s, s') \in R(prg) \stackrel{\text{def}}{=} prg \vdash s \sim_{\text{state}} s' \wedge \text{sound-state}(prg, s)$$

Figure 7. Definition of CompCert's simulation relation for the constant propagation pass.

3.3 CompCert's Verification of Constant Propagation

The correctness proof of the constant propagation pass in CompCert establishes the existence of a simulation relation, R, that relates the loading of the source and target programs. That is, for every well-formed source RTL program prg, it proves there exists a simulation $R(prg)$ such that $(\text{load}(prg), \text{load}(\mathcal{T}_{\text{cp}}(prg))) \in R(prg)$.

The simulation relation, R, used for the constant propagation pass is given in Figure 7. It is defined in terms of matching relations on states, stack frames, and stacks and function definitions ($\sim_{\text{state}}$, $\sim_{\text{frame}}$, $\sim_{\text{stack}}$, and $\sim_{\text{fdef}}$ respectively). These relations take as a parameter the source program, prg, which is used to relate the function definitions of the source and target programs.[2]

We say that two function definitions are related in the program prg, written $fd \sim_{\text{fdef}} fd'$, if the target function, fd', is the result of applying constant propagation to the source function, fd. Two stack frames are related by $prg \vdash sf \sim_{\text{frame}} sf'$ if the function code in sf' is the transformation of the function code of sf, the stack pointer and program counters agree, and the registers of sf' hold equal or more defined values than those of sf. Two stacks are related, $prg \vdash s \sim_{\text{stack}} s'$, if they have the same length and their stack frames are related elementwise.

Two states are related, $prg \vdash s \sim_{\text{state}} s'$ if (1) they are of the same kind, (2) the memory of s' is an extension of that of s, (3) their stacks are related by $\sim_{\text{stack}}$, (4) the respective function definitions are related by $\sim_{\text{fdef}}$ (when applicable), (5) the stack pointer and program counter agree (when applicable), and (6) the registers/arguments/return value of s is equal or less defined than that of s'.

Finally, the two states are in the simulation relation R if they are related by $\sim_{\text{state}}$ and the source state satisfies the value analysis invariant, $\text{sound-state}(prg, s)$. This invariant basically says that the (concrete) value of each variable in the state s is included in the variable's abstract value computed by the analysis at the current program point. The invariant depends on the program for two reasons: (1) so that it can calculate the global environment, $ge = get\text{-}genv(prg)$, and (2) so that it can 'run' the analysis on the program so as to be able to compare its results with the current state.

The basic soundness properties of the value analysis are (1) that the sound-state invariant holds for the initial state of a program, and (2) that it is preserved by execution steps. Formally:

$$\frac{s = load(prg)}{\text{sound-state}(prg, s)} \qquad \frac{\text{sound-state}(prg, s) \quad ge = get\text{-}genv(prg) \quad s \overset{t}{\hookrightarrow}_{ge} s'}{\text{sound-state}(prg, s')}$$

The CompCert proof then establishes (i) that

$$(\text{load}(prg), \text{load}(\mathcal{T}_{\text{cp}}(prg))) \in R$$

and (ii) that R is a simulation relation. As for (i), the initial states after loading satisfy $\sim_{\text{state}}$ by construction, and the initial source state satisfies sound-state thanks to the soundness of the value analysis above.

It remains to show that R is indeed a simulation. Specifically, CompCert shows that it is a "forward" simulation, meaning that for any related states $(s, t) \in R$, if the source state s takes a step to s' with an event σ, the target t also takes a step to *some* state t' with the same event σ, such that $(s', t') \in R$. From $(s, t) \in R$, we know that we are executing the instructions at the same pc. Thus by the definition of constant propagation, the target instruction is either identical to the source instruction or obtained by replacing a variable with a constant or converting a conditional jump to an unconditional jump, depending on the value analysis result. Here, thanks to the soundness of the current state w.r.t. the analysis result and the relation between the two states specified by $\sim_{\text{state}}$, we can easily deduce that executing the source and target instructions results in related states. Also, the soundness of the new source state s' follows from the soundness preservation property of the value analysis stated above.

Finally, we briefly discuss why CompCert establishes a forward rather than a backward simulation, even though the former implies that the source's behaviors refine the target's, which seems the wrong way around. First, a forward simulation is easier to establish than a backward one because a single instruction in the source may be compiled down to several instructions in the target. Second, CompCert composes forward simulations of all passes using the transitivity of forward simulations, which is not hard to show. Then it converts the composed forward simulation between C and assembly to a backward simulation between them using some technical properties of C and assembly (namely, that C is *receptive* and assembly is *determinate*). Finally, from this backward simulation, one can establish that the target's behaviors refine the source's.

It is important to note that this approach of using forward simulations (when convenient) carries over without modification when porting CompCert to Level A and B compositional correctness, as we do in the next section.

4. Adapting the Constant Propagation Proof

In this section, we explain how to adapt the CompCert proof of constant propagation to support separate compilation. Before doing so, let us briefly explain syntactic linking in some more detail since it is central to the compositional correctness results we prove.

Syntactic linking merges global declarations for each identifier. The linker must check if the declarations meet the following conditions:

- The declarations have the same type. They should be either (i) function declarations or definitions of the same signature, or (ii) variable declarations or definitions of the same type.
- At most one of the declarations is a definition. If there is a single definition, then that is the result of the linking; otherwise,

[2] The version shown here is a slight simplification of the actual simulation used in the constant propagation pass. It abstracts away some tedious details of the actual $\sim_{\text{frame}}$ definition that are orthogonal to the problem of porting the simulation proof to SepCompCert. This is merely to streamline the presentation.

everything is necessarily the same declaration, and that is the result of the linking.

We have generically defined syntactic linking for all the languages used in CompCert, as it does not depend on specific language features.

4.1 Verifying Compositional Correctness Level A

Adapting the Simulation Relation Definition To verify compositional correctness Level A, we will—as in the original CompCert proof—construct a simulation relation R that relates the initial states of the source and target programs. The difference is that the source program consists of multiple files, each of which is separately compiled.

$$\frac{prg = \mathtt{s}_1 \circ \ldots \circ \mathtt{s}_n \qquad prg' = \mathcal{T}_{\mathrm{cp}}(\mathtt{s}_n) \circ \ldots \circ \mathcal{T}_{\mathrm{cp}}(\mathtt{s}_n)}{\exists R.\ \text{simulation}\ R \wedge (\mathit{load}(prg), \mathit{load}(prg')) \in R}$$

Therefore, for each function definition (fd) in the source program (prg), the corresponding function definition (fd') in the target program (prg') is no longer obtained by $\mathrm{transfun}(prg, fd)$, but rather by $\mathrm{transfun}(\mathtt{s}_i, fd)$ for some subprogram $\mathtt{s}_i$ of prg. Moreover, the value analysis run as part of $\mathrm{transfun}$ also gets a subprogram of prg as its first argument.

Consequently, the simulation relation we use for the proof of soundness has to change. The main change is, naturally, in the definition of $\sim_{\mathrm{fdef}}$. Two function definitions are now related if the second can be obtained by transforming the first in the context of a subprogram of prg.

$$prg \vdash fd \sim_{\mathrm{fdef}} fd' \stackrel{\text{def}}{=} \exists sprg \subseteq prg.\ fd' = \mathrm{transfun}(sprg, fd)$$

where $sprg \subseteq prg$ iff $\exists sprg'.\ sprg \circ sprg' = prg$.[3]

The second change is to decouple the two uses of prg in $\mathrm{sound\text{-}state}$, changing its signature so that it takes three arguments: two programs prg and $sprg$, and a state s. The first program, prg, corresponds to the full program and is used to calculate the global environment, whereas the second program, $sprg$, is a subprogram of the first one and is used to perform the global analysis (*i.e.,* to detect which variables are constant).

We then define a wrapper predicate $\mathrm{sound\text{-}state}'$ as follows:

$$\mathrm{sound\text{-}state}'(prg, s) \stackrel{\text{def}}{=} \forall sprg \subseteq prg.\ \mathrm{sound\text{-}state}(prg, sprg, s)$$

and change R to use $\mathrm{sound\text{-}state}'$ instead of $\mathrm{sound\text{-}state}$.

Adapting the Proof of Value Analysis There are multiple proofs that require adaptation. First, we have to prove that value analysis is correct with respect to our stronger invariant. That is, we have to show that $\mathrm{sound\text{-}state}'$ holds of the initial state of a loaded program and that it is preserved by execution steps.

The latter requirement is actually trivial to show and requires only very minor changes to the CompCert proof script. The reason for this, informally, is that the uses of the prg and $sprg$ parameters in the revised $\mathrm{sound\text{-}state}$ invariant are really decoupled and that the preservation proof never depends on them being the same.

The former requirement, however, requires some more work: not only should $\mathrm{sound\text{-}state}(prg, prg, \mathrm{load}(prg))$ hold for all programs prg, but rather $\mathrm{sound\text{-}state}(prg, sprg, \mathrm{load}(prg))$ should hold for all programs prg and all subprograms $sprg \subseteq prg$. In essence, to satisfy this stronger statement, the additional requirement that we have to show is that value analysis is monotone with respect to linking. That is, for each global variable `x`, we prove

$$sprg \subseteq prg \implies \text{abs-val}(sprg, \mathtt{x}) \sqsupseteq \text{abs-val}(prg, \mathtt{x})$$

[3] In the Coq development, we define $sprg \subseteq prg$ in an equivalent but more direct, semantic way, rather than relying on syntactic linking ($\circ$). This enables us to avoid having to prove associativity and commutativity of linking.

```
// a.c                          // b.c
#include <stdio.h>              #include <stdio.h>
int x;                          extern int x;
extern int* const xptr;         int* const xptr = &x;
int main() {
  x = 1;
  *xptr = 0;
  // expected: 0, actual: 2
  printf("%d\n",x+x);
  return 0;
}
```

Figure 8. A bug due to CompCert 2.4 value analysis: two C files whose separate compilation and linking exposes the bug.

where $\text{abs-val}(prg, \mathtt{x})$ is the abstract value of `x` computed by the global analysis on prg. In other words, running the analysis on a larger program may only give results that are at least as precise.

Somewhat surprisingly, (the global part of) the value analysis in CompCert 2.4 does not satisfy this monotonicity requirement because of its treatment of variables declared as both `extern` and `const`. Figure 8 presents two C files that, when compiled separately and linked together, expose the bug. Since the global variable `xptr` is declared using the `const` qualifier, the global part of CompCert 2.4's value analysis assumes that it is uninitialized and therefore assigns it the abstract value $\bot$. As a result, the value analysis deems that `xptr` cannot possibly alias with `x`. At the `printf` statement, it hence deduces that $\mathtt{x} = 1$, and constant propagation "optimizes" away the summation `x+x` to the constant 2, which gets printed. This analysis, however, is unsound. In particular, the assumption that `xptr` is uninitialized is invalid in the context of multiple separately compiled files: since `xptr` is also declared as `extern`, another file (`b.c`) can provide a definition that initializes it. And indeed, since the definition of `xptr` in `b.c` causes `x` and `xptr` to alias, the correct result is 0, not 2.

Restoring soundness of the value analysis is straightforward: one simple if rather crude fix (which has been adopted by CompCert 2.5 since we reported the bug) is just to ignore the `const` modifier on `extern` declarations. Having done that, it is easy to show that the analysis is monotone with respect to program linking and that therefore the initial state of loading a program satisfies the stronger invariant $\mathrm{sound\text{-}state}'(prg, \mathrm{load}(prg))$.

Adapting the Proof of Constant Propagation Showing that constant propagation is sound requires only very little additional work.

The only important difference is in the treatment of global environments that index the $\hookrightarrow$ relation. Generally, these environments are obtained by the respective programs ($ge = \mathit{get\text{-}genv}(prg)$ and $ge' = \mathit{get\text{-}genv}(prg')$).

The original CompCert 2.4 proof used the fact that $prg' = \mathcal{T}_{\mathrm{cp}}(prg)$ to establish a relationship between ge and ge'. It proved two basic properties relating the two global environments: (1) that they map each global variable name to the same block identifier, and (2) that if ge maps a block identifier to a function signature or definition fds, then ge' maps it to $\mathrm{transfun}(prg, fds)$, where $\mathrm{transfun}$ applied to a function signature returns the same signature. These lemmas were then used in the proof that R is a simulation relation.

Since, now, the relationship between prg and prg' is more involved, we have to update the proof of the first lemma, which is rather straightforward, as well as the statement and proofs of the second lemma. For the second lemma, we now assert that if ge maps a block identifier to a function signature or definition fds, then ge' maps it to $\mathrm{transfun}(sprg, fds)$ for *some* subprogram $sprg \subseteq prg$. Besides this change, the proof that now (the new definition of) R is a simulation relation is basically unchanged. The few lines of the

proof script that required editing were those invoking the lemmas about the relationship between the global environments.

4.2 Verifying Compositional Correctness Level B

Adapting the Simulation Relation Definition We move on to verifying the second level of compositional correctness, that of composition with the same compiler modulo optimization flags. As we have explained in §2.3, for every optional optimization pass, we need to show that linking it against the identity compiler is sound. Since constant propagation is one of the optional optimization passes, we have to prove the following:

$$\frac{\begin{array}{c} prg = \mathtt{u}_1 \circ \ldots \circ \mathtt{u}_m \circ \mathtt{s} \circ \mathtt{v}_1 \circ \ldots \circ \mathtt{v}_n \\ prg' = \mathtt{u}_1 \circ \ldots \circ \mathtt{u}_m \circ \mathcal{T}_{\mathrm{cp}}(\mathtt{s}) \circ \mathtt{v}_1 \circ \ldots \circ \mathtt{v}_n \end{array}}{\exists R.\ \text{simulation}\ R \wedge (\mathit{load}(prg), \mathit{load}(prg')) \in R}$$

In the scenario above, the corresponding target function definition fd' of a source function definition fd is either syntactically identical to fd if it belongs to one of the $\mathtt{u}_i/\mathtt{v}_j$ files, or has been obtained by optimizing fd if it belongs to the `s` file. To account for the change, we should therefore redefine the $\sim_{\mathrm{fdef}}$ relation as follows:

$$prg \vdash fd \sim_{\mathrm{fdef}} fd' \stackrel{\text{def}}{=} fd' = fd \vee (\exists \mathit{sprg} \subseteq prg.\ fd' = \mathrm{transfun}(\mathit{sprg}, fd))$$

This is the only change needed in the simulation relation.

Adapting the Proof of Constant Propagation To avoid changing the proof script of the main lemma showing that R is in fact a simulation, we prove a helper lemma (*transf_step_correct_identical*) saying that, given two matching instruction states with $fd' = fd$, when the source state takes a step, the target state can also take a step and reach a matching state. As explained in §2.3, the content of the proof of this helper lemma is already present in the existing simulation proof, just not in one place, so it simply needs to be extracted and consolidated. We then adapt the proof of the main lemma (showing R is a simulation) so that it performs a case split on whether $fd = fd'$ or not, and correspondingly either invokes our helper lemma or uses the same proof script as for Level A.

5. Porting CompCert to SepCompCert

In the previous sections, we looked at the specific example of constant propagation in detail and explained how we ported CompCert's proof of that pass to SepCompCert's Level A and B notions of compositional correctness. In this section, we discuss some details of other CompCert passes for which porting to SepCompCert required some interesting (but still not much) work.

Figure 9 shows which verification technique we apply to each optimization pass of CompCert. For Level A, we apply the trivial technique based on commutativity with linking to 13 passes and the non-trivial technique to 6 passes. For Level B, we apply our technique to all 6 RTL-to-RTL passes.

5.1 RTL-Level Optimizations that Rely on Value Analysis

Three RTL-level optimizations rely on the value analysis: constant propagation, common subexpression elimination (CSE), and dead-code elimination (DCE). These passes are inter-procedural solely because they rely on the value analysis. Thus, the porting of the proofs of CSE and DCE to support compositional correctness proceeded analogously to the porting of the constant propagation pass.

5.2 Selection

The selection pass "recognizes opportunities for using combined arithmetic and logical operations and addressing modes offered by the target processor." [4]. The pass is mostly intra-procedural, except for the following two transformations on function calls.

Recognizing Immediate Calls The selection pass transforms an indirect call via a function pointer expression, say e_p, into an immediate call, if it can determine that the expression e_p always evaluates to the pointer to an internal function. The pass uses a simple analysis `classify_call`(prg, e_p) for determining this. For separate compilation, we proved the monotonicity of the analysis: the result `classify_call`(sprg, e_p) for a subprogram $\mathit{sprg} \subseteq prg$ is sound w.r.t. the whole program prg.

64-bit Integer Operations into Library Calls The selection pass transforms some 64-bit integer operations into calls to library helper functions. For example, `(long long) f`, a cast from `float` to `long long`, is transformed into a call `__int64_dtos(f)` to the corresponding helper function. For this transformation to be valid, the pass should ensure that the helper function (*e.g.,* `__int64_dtos`) is declared as an external function in the source program's global environment. CompCert has a designated checker `check_helpers`(prg) to ensure this property.

For separate compilation, we proved that the checker for helper functions is monotone w.r.t. linking: `check_helpers`$(prg1)$ and `check_helpers`$(prg2)$ implies `check_helpers`$(prg1 \circ prg2)$ for all programs $prg1$ and $prg2$. The proof is a little bit involved, as linking reorders global declarations, and the corresponding logical blocks for the helper functions in the global environments may vary.

Compiler Bug We Found The original CompCert 2.4 used a function `get_helpers` instead of `check_helpers`. We found and reported a compiler bug related to `get_helpers`, which was subsequently confirmed.

We found the bug in the course of proving monotonicity regarding `get_helpers`. This function is directly implemented in OCaml and its property is axiomatized in Coq as follows.

```
Axiom get_helpers_correct:
  forall ge hf, get_helpers ge = OK hf ->
  i64_helpers_correct ge hf.
```

The problem was that this axiom is not strong enough to prove monotonicity, and even worse, not true for the OCaml implementation of `get_helpers`. One of the properties this axiom postulates is that helper functions like `__int64_dtos` are only declared but not defined in the source program. However, `get_helpers` does not check it at compile time!

Here is an example that exposes the bug.

```
#include <stdio.h>
long long __i64_dtos(float t) {
  return 3;
}
int main() {
  // expected: 5, actual: 3
  printf(\%lld\n", (long long) 5.0f);
  return 0;
}
```

Here the cast `(long long) 5.0f` is converted to a call to the library helper function `__i64_dtos(5.0f)` by the selection pass. However, we successfully hijack the function call by overwriting the function `__i64_dtos`, which results in printing 3 instead of the correct result 5. Now, strictly speaking, due to the hijacking of a reserved identifier, this example has undefined semantics according to the C99 standard, so CompCert's behavior here is technically legal. But the dependence on an invalid axiom is clearly a bug.

After we reported this bug, it was fixed in the development branch of CompCert (and subsequently CompCert 2.5). In this fix, which we backported to CompCert 2.4 using `git-cherry-pick`, the OCaml `get_helpers` function is replaced by the aforementioned `check_helpers`, which is implemented and verified directly in Coq, thereby avoiding the need for an invalid axiom.

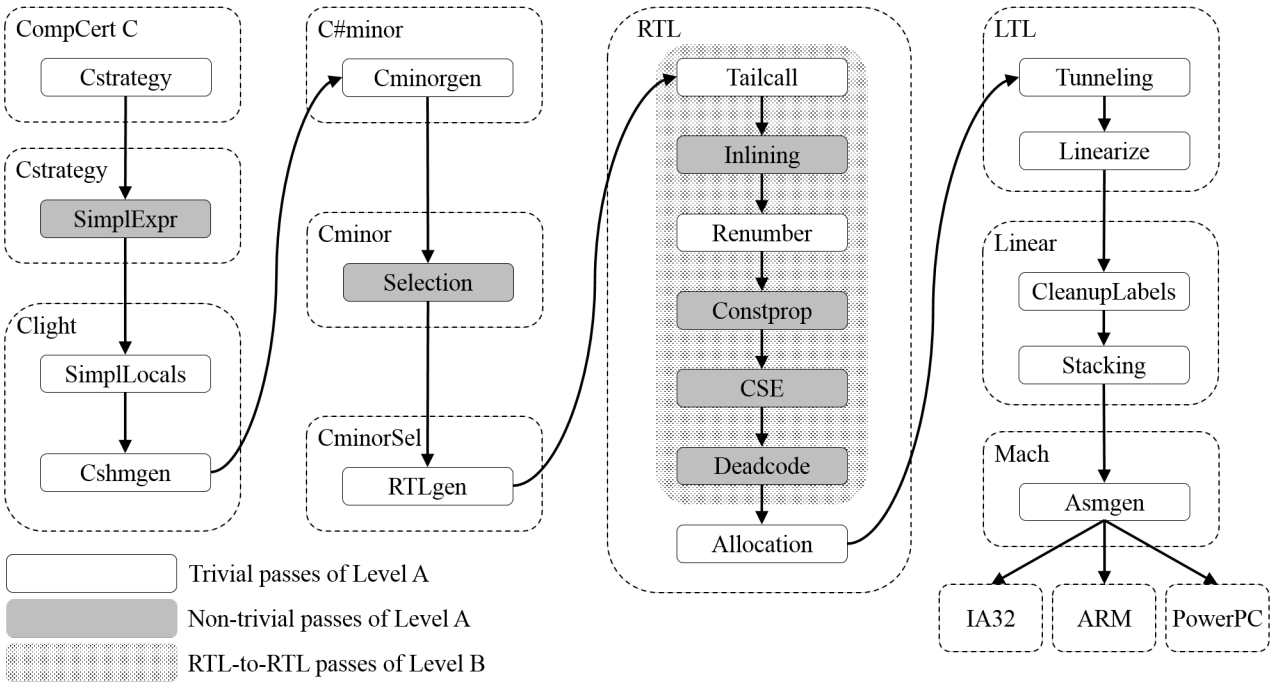

Figure 9. Optimization passes of SepCompCert Level A and Level B.

5.3 Inlining

The inlining pass is inherently inter-procedural, as it replaces a call to a simple function with the body of the function. In the pass, the selector `funenv_program`(prg) chooses internal function definitions of prg that are worth inlining in other functions. For the inlining pass to be valid, the pass should ensure that function definitions in `funenv_program`(prg) are indeed defined in the global environment $get\text{-}genv(prg)$.

For separate compilation, we proved that the global environment initialization is monotone for function definitions: if a function definition, say fd, is defined in $get\text{-}genv(sprg)$ and $sprg \subseteq prg$, then fd is also defined in $get\text{-}genv(prg)$. The proof is a little bit involved for the same reason as for `check_helpers` in the selection pass: linking reorders global declarations and the corresponding logical blocks in the global environments.

5.4 SimplExpr

The SimplExpr pass is essentially intra-procedural since just "side effects are pulled out of CompCert C expressions" [4]. However, it does not commute with linking because a single counter is used globally to generate temporary variable names for all function definitions.

Updating the existing proof was easy because the original simulation relation R does not bake in the specific way temporary variable names are generated. Thus, we did not need to change the simulation relation R and the simulation proof at all. We just needed to update the proof that the initial states after loading satisfy the relation R even in the presence of separate compilation, which was straightforward.

6. Results

We applied our Level A and B techniques to CompCert 2.4 with three patches applied (two bug fixes and RTL-level optimization flags), resulting in *SepCompCert-LevelA* and *SepCompCert-LevelB*. In the former, we prove the behavioral refinement result between the source C program obtained by syntactically linking several C files and the target assembly program obtained by syntactically linking the results of compiling source files with the *same* optimization flag. In the latter, we prove the same behavioral refinement result even when each source file is compiled with a different optimization flag.

In the table in Figure 10, we summarize the changes we made in number of lines of Coq. For these statistics, we first split the development of CompCert into two categories: (i) Compiler & Verification; and (ii) Metatheory (*i.e.,* everything else). The former includes all Coq files in the directories `cfrontend`, `backend`, `driver`, `arm`, `ia32`, and `powerpc`; and the latter includes all other files.

We calculated the statistics fully automatically using the Unix `diff` command. The column **Rm** shows the number of lines of code (LOC) removed from the original CompCert code reported by the diff command, and the columns **AddD** and **AddN** together show the number of LOC added in SepCompCert. Here, **AddD** counts the LOC that are derived from the original CompCert code including copy-pasted-and-modified code and **AddN** the LOC that we newly proved. We syntactically marked the newly proved code in SepCompCert, so that we can automatically distinguish **AddD** and **AddN**.

The shaded part in the table denotes interesting changes we made. In Level A, the shaded part is mainly due to proving various monotonicity properties. In Level B, the shaded part is mainly due to copy-paste-and-modifying the original proof of simulation. Examples of (what we consider) uninteresting changes, which are not shaded, include (a) proving straightforward "wheel-greasing" lemmas that merely serve to streamline the proof effort, and (b) updating automatically generated hypothesis names appropriately (*e.g.,* changing `apply H1` to `apply H2`).

SepCompCert supports verified separate compilation to all three target assembly languages of CompCert—PowerPC, ARM, and IA32—with very few changes made to the original proofs. In the whole development, for Level A, we modified only 0.2% of the existing code (**Rm**) and introduced an additional 1.6%

	CompCert (LOC)	SepCompCert-LevelA			SepCompCert-LevelB		
		Rm	AddD	AddN	Rm	AddD	AddN
Total	206702	318 (0.2%)	465 (0.2%)	3392 (1.6%)	372 (0.2%)	1439 (0.7%)	4845 (2.3%)
Compiler & Verification	88451	318 (0.4%)	465 (0.5%)	1153 (1.3%)	372 (0.4%)	1439 (1.6%)	1726 (2.0%)
`driver/Compiler.v`	367	6 (1.6%)	6 (1.6%)		9 (2.5%)	9 (2.5%)	
`.../ValueAnalysis.v`	1825	16 (0.9%)	33 (1.8%)	86 (4.7%)	16 (0.9%)	33 (1.8%)	86 (4.7%)
`.../Cstrategy.v`	3070						
`.../SimplExpr(proof|spec).v`	3383	14 (0.4%)	10 (0.3%)	68 (2.0%)	14 (0.4%)	10 (0.3%)	68 (2.0%)
`.../SimplLocalsproof.v`	2251			19 (0.8%)			19 (0.8%)
`.../Cshmgenproof.v`	1515			21 (1.4%)			21 (1.4%)
`.../Cminorgenproof.v`	2256			11 (0.5%)			11 (0.5%)
`.../Select*proof.v`	2686	86 (3.2%)	117 (4.4%)	203 (7.6%)	86 (3.2%)	117 (4.4%)	203 (7.6%)
`.../RTLgen(proof|spec).v`	2781			12 (0.4%)			12 (0.4%)
`.../Tailcallproof.v`	627			8 (1.3%)	21 (3.3%)	186 (29.7%)	37 (5.9%)
`.../Inlining(proof|spec).v`	1979	59 (3.0%)	98 (5.0%)	50 (2.5%)	60 (3.0%)	323 (16.3%)	64 (3.2%)
`.../Renumberproof.v`	267				30 (11.2%)	126 (47.2%)	35 (13.1%)
`.../Constpropproof.v`	644	50 (7.8%)	68 (10.6%)	26 (4.0%)	52 (8.1%)	180 (28.0%)	36 (5.6%)
`.../CSEproof.v`	1238	29 (2.3%)	56 (4.5%)	34 (2.7%)	30 (2.4%)	146 (11.8%)	45 (3.6%)
`.../Deadcodeproof.v`	1024	58 (5.7%)	77 (7.5%)	34 (3.3%)	54 (5.3%)	171 (16.7%)	45 (4.4%)
`.../Allocproof.v`	2219			14 (0.6%)			14 (0.6%)
`.../Tunnelingproof.v`	417						
`.../Linearizeproof.v`	750			8 (1.1%)			8 (1.1%)
`.../CleanupLabelsproof.v`	372						
`.../Stackingproof.v`	2894			10 (0.3%)			10 (0.3%)
`.../Asmgenproof0proof.v`	867						
`arm/*proof*.v`	4046						
`ia32/*proof*.v`	3683						
`powerpc/*proof*.v`	3912						
others in `cfrontend`, `backend`, `driver`, `arm`, `ia32`, `powerpc`	43378			549 (1.3%)		138 (0.3%)	1012 (2.3%)
Metatheory	118251			2239 (1.9%)			3119 (2.6%)

Rm: LOC removed from CompCert **AddD**: LOC added but derived from CompCert **AddN**: LOC newly added
%: ratio to the LOC of CompCert shaded cell : interesting changes

Figure 10. Changes to lines of code when porting CompCert to SepCompCert-LevelA and SepCompCert-LevelB.

(**AddN+AddD-Rm**); and for Level B, we modified only 0.2% of the existing code (**Rm**) and introduced an additional 2.8% (**AddN+AddD-Rm**). We spent less than two person-months in total for the whole development, much of which was spent understanding the existing CompCert development.

7. Discussion and Related Work

Compositional Compiler Correctness The most closely related work to ours is Stewart *et al.*'s Compositional CompCert [17], which establishes compositional correctness for a significant subset of CompCert 2.1. Their approach builds on their previous work on *interaction semantics* [3], which defines linking between modules in a (somewhat) language-independent way. (The languages in question must share a common memory model, as is the case for C, assembly, and all the intermediate languages of CompCert.) Essentially, interaction semantics enables Compositional CompCert to reduce the compiler verification problem to one of contextual refinement. The output of the compiler is proven to refine the source under an arbitrary "semantic context", which may consist of a linking of C and assembly modules.

On the one hand, Compositional CompCert is targeting a more ambitious goal than SepCompCert. Their approach inherently supports the possibility of linking results of multiple different compilers, as well as the ability to link C modules with hand-coded assembly modules, both of which are beyond the scope of our techniques.

On the other hand, as we explained in the introduction, the modesty of our goal is quite deliberate—it enables our SepCompCert verification to use a considerably more lightweight approach than they do. For example, they report that their porting of CompCert passes to verify compositional correctness took approximately 10 person-months and led to more than a doubling in the size of each pass. In contrast, our porting took less than 2 person-months, and even if we look at just the passes alone (ignoring the metatheory), they are on average less than 2% (for Level A) or 4% (for Level B) larger than the original CompCert passes. The new metatheory backing up our proof technique is also smaller than the corresponding new metatheory of Compositional CompCert by roughly a factor of 7.

One reason for this, we believe, is that the *structured simulations* employed in the Compositional CompCert proof require all passes in the compiler to be verified using a common "memory-injection" invariant. In contrast, with SepCompCert we were able to essentially reuse the invariants from the original CompCert proof, which are different for different passes. Another potential reason concerns the treatment of inter-module vs. external function calls. In CompCert, external functions are assumed to satisfy a number of axioms, which are used in the verification to establish that external function calls preserve the simulation relation in each pass. In Compositional CompCert, inter-module function calls are treated the same as external function calls, and as a result Stewart *et al.*'s verification must additionally establish that functions compiled by the compiler satisfy the external function axioms. In contrast, with SepCompCert we reduce the problem of verifying separate compilation to that of verifying correctness of compilation for a whole (multi-module) program. Thus, for us, inter-module function calls are *not* treated as external calls—they merely shift control to another part of the program—and there is no need for us to prove that CompCert-compiled functions satisfy the external function axioms.

There are two other points of difference worth mentioning. First, we have ported over the entire CompCert 2.4 compiler, from C to assembly (including the x86, Power, and ARM backends), whereas Compositional CompCert omitted the front-end of CompCert 2.1 (from C to Clight), along with three of its RTL-level optimizations (CSE, constant propagation, and inlining). Second, due to its use of interaction semantics, Compositional CompCert employs a bespoke "semantic" notion of linking (even for linking assembly files), which has not yet been related to the standard notion of syntactic linking (see §4) that we employ in the SepCompCert verification. Syntactic linking corresponds much more closely to the physical linking of machine code implemented by the `gcc` linker that CompCert uses by default. Proving this is outside the scope of our verification effort, however, since CompCert only verifies correctness of compilation down to the assembly level, not to the machine-code level where linking is actually performed.

Concurrently with Stewart *et al.*'s work, Ramananandro *et al.* [14] developed a different approach to compositional correctness for CompCert. While similar in many ways to the approach of Stewart *et al.*, the *compositional semantics* of Ramananandro *et al.* defines linking so that the linking of assembly-level modules boils down to syntactic linking (essentially, concatenation), as it does in SepCompCert. On the other hand, they have only used their approach to compositionally verify a few passes of CompCert.

Also concurrently with the above work, Gu *et al.* [5] have developed CompCertX, a compositional adaptation of CompCert specifically targeted for use in the compositional verification of OS kernels. CompCertX supports linking of compiled C code with hand-written assembly code, and ports over all passes of CompCert, but the source and target of the compiler are different from CompCert's. In particular, the ClightX source language of CompCertX does not generally allow functions to modify other functions' stack frames and thus does not support stack-allocated data structures. Although this restriction is not problematic for their particular application to OS kernel verification, it means that, as the authors themselves note, "CompCertX can not be regarded as a full featured separate compiler for CompCert."

There have been several approaches proposed for compositional correctness of compilers other than CompCert as well, although these all involve *de novo* verifications rather than ports of existing whole-program compiler verifications.

Perconti and Ahmed [13] present an approach to compositional compiler correctness for ML-like languages. They use multi-language semantics to combine all the languages of a compiler into one joint language, with wrapping operations to coerce values in one language to values of the appropriate type in the other languages. Like Compositional CompCert, their approach recasts the compiler verification problem as a contextual refinement problem, except that they model contexts syntactically rather than semantically and use *logical relations* as a proof technique for establishing contextual refinement rather than structured simulations. It is difficult to gauge how well this approach scales as a practical compiler verification method because it has not yet been mechanized or applied to a full-blown compiler.

Wang *et al.* [18] develop a compositional verification framework for a compiler for Cito, a simple C-like language [18]. Their approach is quite different from the others in that it characterizes the compiler verification problem in terms of Hoare-style specifications of assembly code. Currently, the approach is limited in its ability to talk about preservation of termination-sensitive properties, and as its verification statement is so different from the traditional end-to-end behavioral refinement result established by a compiler like CompCert, it is not clear how Wang *et al.*'s method could reuse existing CompCert-style verifications.

Most recently, Neis *et al.* [12] present *parametric inter-language simulations (PILS)*, which they use to compositionally verify Pilsner, a compiler for an ML-like core language, in Coq. PILS build on earlier work by Hur *et al.* on logical relations [2, 6] and parametric bisimulations (aka relation transition systems) [7, 8]. Pilsner supports a very strong compositional correctness statement, but it also required a major verification effort, involving several person-years of work and 55K lines of Coq.

Generality of Our Techniques In this paper, we have presented several simple techniques for establishing Level A and Level B compositional correctness, and demonstrated the feasibility and effectiveness of these techniques by porting a major landmark compiler verification (CompCert 2.4) to support compositional correctness without much difficulty at all. We hope the almost embarrassingly simple nature of these techniques will encourage future compiler verifiers to consider proving at least a restricted form of compositional correctness for their compilers from the start.

One may wonder, however, how general our techniques are. Are they dependent on particular aspects of CompCert 2.4? Can they be applied to other verified compilers? For C or for other languages? Given the landscape of compiler verification, dotted as it is with unique and majestic mountains, it is difficult to give sweepingly general answers to these questions. But we can say the following.

We believe our techniques should be applicable to the most recent version of CompCert (2.5), but a necessary first step is to determine the appropriate notion of syntactic linking. CompCert 2.5 introduces support for `static` variables (which in C means variables that are only locally visible within a single file). The presence of `static` variables means that the simple canonical definition of syntactic linking we have used no longer works and must be revisited. Assuming a reasonable definition can be found, as we expect, we do not foresee any problems adapting our techniques to handle it.

Regarding the application to compilers for other languages, we can only speculate, but we also do not foresee any fundamental problems. For instance, CakeML [9] is a verified compiler for a significant subset of Standard ML, implemented in HOL4. CakeML's end-to-end verification statement concerns the correctness of an x86 implementation of an interactive SML read-eval-print loop. In that sense, the verification is not exactly "whole-program" because new code can be compiled and added to a global database interactively. But it also does not support true separate compilation in the sense that SepCompCert does because modules cannot be compiled independently of the other modules they depend on.

We believe in principle it should be possible to use our techniques to adapt CakeML to verify correctness of separate compilation, because CakeML is not an optimizing compiler and in particular does not perform any optimizations that depend fundamentally on the whole-program assumption. The key challenge will be figuring out how to define separate compilation and linking themselves. The latter may be especially interesting since CakeML (unlike CompCert) verifies correctness of compilation all the way down to x86-64 machine code, and thus linking will need to be defined at the machine-code level.

Acknowledgements

This research has been supported in part by the ICT R&D program of MSIP/IITP (Grant R0132-15-1006), and in part by EC FET project ADVENT (308830). The first and second authors have been supported by Korea Foundation for Advanced Studies Scholarships.

We thank Jim Apple and Xavier Leroy for helpful feedback.

References

[1] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[2] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[3] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *ESOP*, 2014.

[4] The CompCert C compiler. `http://compcert.inria.fr/`.

[5] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *POPL*, 2015.

[6] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[7] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[8] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The transitive composability of relation transition systems. Technical Report MPI-SWS-2012-002, MPI-SWS, 2012.

[9] Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL*, 2014.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[11] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[12] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ICFP*, 2015.

[13] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.

[14] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. A compositional semantics for verified separate compilation and linking. In *CPP*, 2015.

[15] SepCompCert. `http://sf.snu.ac.kr/sepcompcert/`.

[16] J. Souyris. Industrial use of CompCert on a safety-critical software product, February 2014. Talk slides available at: `http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf`.

[17] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, 2015.

[18] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*, 2014.