Abstract

This paper reports on the development of Compositional CompCert, the first verified separate compiler for C.

Specifying and proving separate compilation for C is made challenging by the coincidence of: compiler optimizations, such as register spilling, that introduce compiler-managed (private) memory regions into function stack frames, and C’s stack-allocated addressable local variables, which may leak portions of stack frames to other modules when their addresses are passed as arguments to external function calls. The CompCert compiler, as built/proved by Leroy et al. 2006–2014, has proofs of correctness for whole programs, but its simulation relations are too weak to specify or prove separately compiled modules.

Our technical contributions that make Compositional CompCert possible include: language-independent linking, a new operational model of multilanguage linking that supports strong semantic contextual equivalences; and structured simulations, a refinement of Beringer et al.’s logical simulation relations that enables expressive module-local invariants on the state communicated between compilation units at runtime. All the results in the paper have been formalized in Coq and are available for download together with the Compositional CompCert compiler.

1. Introduction

Verified separate compilation is the process of independently translating a program’s components in a way that preserves correctness of the program as a whole. In the most general case, a verified separate compiler supports heterogeneous source programs, in which some modules are written in a high-level language such as C while others are written in a lower-level language such as assembly. A verified separate compiler in this context is one that preserves the behavior of the entire source program, comprising the C and assembly-language modules, when the C code is compiled.

Separate compilation has numerous practical benefits. It speeds up development cycles by enabling recompilation of just those source modules that have been edited by the programmer. It enables shared libraries—separately compiled modules mapped at runtime into the virtual address spaces of multiple processes. It promotes modularity in the large, by enabling programmers to write applications containing logically distinct translation units, each composed at a different level of abstraction or even in a different programming language altogether.

But perhaps equally important are applications of verified separate compilers to modular verification: proving a whole program correct by specifying and verifying its modules independently (with respect to the specifications of the other modules). Compiling verified modules with a semantics-preserving separate compiler results in correctness not only of each compiled module, but also of the target whole program linked and running on the machine.

In this paper, we present Compositional CompCert, a fully verified separate compiler from CompCert’s Clight language to x86 assembly. Each phase uses the exact same compilation function as CompCert 2.1, but a significantly strengthened specification that supports verified separate compilation of multimodule programs. In contrast, original CompCert’s specification is limited to whole programs and fails to account for general shared-memory interaction. That is, CompCert is not certified for calls to other modules, especially if two modules might access the same memory locations. We certify the correctness of such calls.

Compositional CompCert builds on work by Beringer et al. [6], which showed how to adapt CompCert to support shared-memory interaction between a single compilation unit and its environment, under the restriction that the environment was not itself compiled. The technical innovations that enabled the adaptation to shared memory were twofold: First, a novel flavor of operational semantics, called interaction semantics (“core semantics” in [6]), for modeling a module’s interactions with its environments; and second, a new proof method, called logical simulation relations (LSRs), for proving correctness of compiler phases with respect to the interaction semantics interface. LSRs supported interlanguage reasoning between the source, intermediate, and target languages of an optimizing compiler by modeling all languages uniformly, as interaction semantics. LSRs also composed vertically, or transitively, which made it possible to break down the correctness proof of a multiphase compiler into proofs of the individual compiler phases. (Leroy’s original CompCert was also vertically compositional, but only because stack-allocated memory was not observable.)

The main deficiency of LSRs was that they did not compose horizontally. It was not possible, in general, to infer correct compilation of a whole program from the correct compilation of its modules. The reason was that LSRs, like CompCert’s original simulation relations, imposed assumptions (relieties) on the evolution of memory over external function calls, such as “spilled local variables are unmodified,” but did not show that compiled code preserved the corresponding guarantees. This asymmetry between relies and guarantees made LSRs inapplicable whenever a module and its environment were both compiled—a situation that occurs not only when libraries are compiled, but also whenever there is a cyclic dependency between the modules of a program.
Contributions In Compositional CompCert we overcome these difficulties, and achieve horizontal compositionality, with the following innovations over previous work:

1. Language-Independent Linking is an extension of interaction semantics that gives the operational interpretation of horizontal composition. By abstracting from the details of how modules are implemented (or even in which language they are implemented), language-independent linking models the interactions of modules in different languages, even those with different calling conventions.

2. Structured Simulations refine LSRs to support the rely-guarantee relationship necessary for horizontal composition, while maintaining vertical compositionality. The key ingredients of structured simulations are: fine-grained subjective invariants on the state communicated between modules at runtime, and a leakage protocol that ensures that structured simulation proofs respect the reachability relation induced by C pointers.

Compositional CompCert’s top-level correctness theorem is a variant of contextual equivalence between source and compiled multi-module programs in which contexts are specified of contextual equivalence (Sections 3 and 6) are compositional interaction semantics, independent of the language level.

Contributions

In Compositional CompCert we overcome these difficulties, and achieve horizontal compositionality, with the following innovations over previous work:

1. Language-Independent Linking is an extension of interaction semantics that gives the operational interpretation of horizontal composition. By abstracting from the details of how modules are implemented (or even in which language they are implemented), language-independent linking models the interactions of modules in different languages, even those with different calling conventions.

2. Structured Simulations refine LSRs to support the rely-guarantee relationship necessary for horizontal composition, while maintaining vertical compositionality. The key ingredients of structured simulations are: fine-grained subjective invariants on the state communicated between modules at runtime, and a leakage protocol that ensures that structured simulation proofs respect the reachability relation induced by C pointers.

Compositional CompCert’s top-level correctness theorem is a variant of contextual equivalence between source and compiled multi-module programs in which contexts are specified semantically, as (nearly) arbitrary observations on the memory state at external call points. Contexts are not limited to programs in C or x86 but include mathematical relations expressed in Gallina.

In addition to sketching formal results that relate linking semantics, structured simulations, and contextual equivalence, we outline the instantiation of interaction semantics to the source and target languages of our compiler, Clight and x86, and present an overview of the adaptation of the relevant proofs of the CompCert phases.

All results in this paper have been proved formally in Coq, and are available, along with the Compositional CompCert compiler itself, at the following URL: 

https://sites.google.com/site/ccompcert/

Before proceeding with the technical details, we briefly discuss some key aspects of the above innovations.

1.1 Key Ideas

Language-independent linking provides a generic operator $L$ for composing interaction semantics, independent of the language level of individual modules. Our statements of separate compilation and contextual equivalence (Sections 3 and 5) are cross-language in the sense that they apply even to the situation in which a multilanguage source program is compiled to a multilanguage target program. Informally, imagine a source program $P_S$ consisting of source modules $S_1, \ldots, S_n$, each in a different language. Then if target modules $P_T = T_1 \cdots T_n$ are shown related to $S_1 \cdots S_n$ (e.g., by $n$ different structured simulations), the results of Sections 3 and 5 give us that the target “linked” program $P_T$ is contextually equivalent to the source “linked” program $P_S$ (under certain restrictions on the $S_i$ and $T_i$, explained in Section 3), for a strong semantic notion of program context. Of course, it’s not clear a priori what it means to link multilanguage programs, at least not in any operational sense: answering this question is the subject of the first part of Section 3. In principle, these results mean that the target modules $T_1 \cdots T_n$ need not be generated by any particular compiler: our contextual equivalence results depend only on the existence of the simulations. In practice in Compositional CompCert, the $S_i$ are programs in Clight or hand-written assembly while the $T_i$ are x86 assembly programs.

Structured simulations evolve from LSRs by lifting the restriction to fixed environments, in two steps: First, we impose a rely-guarantee discipline (inspired in part by the rely-guarantee simulations used by Liang et al. [15] to prove contextual refinement of concurrent programs) on the interactions of program modules. The rely-guarantee discipline ensures that module compilation preserves the same properties that modules themselves assume about the behavior of external functions (those defined in other modules). This, in turn, makes it possible to implement external functions or libraries with code that is itself compiled, as in Figure 1. Second, we enrich the simulation relations with additional “ownership” data, which makes it possible to distinguish memory regions that are rearranged during compilation of distinct translation units. For example, the portion of the stack frame reserved for spilling during compilation of a function $A.f$ can be distinguished from the spill region reserved for a second function $B.g$, defined in a distinct translation unit $B$.

A key insight here is that the invariants that apply to distinct regions of memory—such as the regions reserved by the compiler for $A.f$’s and $B.g$’s spilled locals—are subjective: function $A.f$ can write to its own spilled locals but not to $B.g$’s, and vice versa for $B.g$ with respect to $A.f$’s spills. Structured simulations deal with this subjectivity by imposing an “us vs. them” discipline on compiler correctness invariants: Each structured simulation distinguishes the parts of the state that it controls (the “us”) from the parts of the state controlled by the environment (the “them”). This discipline is reminiscent in some ways of Ley-Wild and Naneyvski’s subjective concurrent separation logic [13], though here we apply a rely-guarantee discipline to the two-program invariants used to prove compiler correctness rather than to the verification of concurrent programs. To ensure that structured simulations validate contextual equivalences, and thus are reusable in many different program contexts, we make them parametric in nearly all state updates that can occur to the “them” portion of the state at external call points.

Another ingredient is a “leakage” protocol, which ensures that the views of the memory state imposed by the compiler invariants for different modules remain consistent. For example, when $A.f$ calls $B.g$ with arguments $\gamma$, we require that $A.f$’s compilation invariant “give up exclusive control” of all the memory regions reachable from $\gamma$ (i.e., following pointer chains rooted in $\gamma$). This condition represents the guarantee that, while later compilation stages of $A.f$ can still reorganize parts of the state reachable from $\gamma$ (e.g., by changing the order in which memory regions are allocated), they cannot remove these memory regions entirely (e.g., by dead code/memory analysis): the existence of the memory regions in question has been leaked irrevocably to the environment. Similarly, at external function return points, memory regions reachable from the return value are “leaked in” to the caller’s compilation invariant—representing the rely that these regions will never later be removed by compilation of the environment. Our language-independent linking semantics and contextual equivalence proof ensure that these conditions are in rely-guarantee relation.

Interestingly, this leakage protocol bears much in common with the system-level semantics of Ghica and Tzevelekos [10]. There, Ghica and Tzevelekos define a game semantics for a C-like language that avoids imposing so-called combinatorial (i.e., syntactic) restrictions on the moves of the environment, by applying what they call “epistemic” restrictions instead. These epistemic conditions, which parallel our leakage conditions, allow the environment to update the state in nearly any way as long as the updates are to memory regions leaked to the environment during previous interactions with the client program. This leads to a strong semantic notion of program context similar to the one we develop in Section 6.

While Ghica and Tzevelekos were interested in modeling open C-like programs and their environments, not compiler correctness in this setting, we view the coincidence of our leakage conditions with their system-level semantics as evidence of the naturalness of our leakage protocol (Section 4).
Overview
We begin by reviewing interaction semantics \[6\], an operational model of shared-memory module and thread interaction. Section 2 employs interaction semantics to define the operational semantics of multilanguage linked programs, and semantic contextual equivalence. Then, we introduce structured simulations in Section 3. Section 4 presents the main theoretical results: vertical and horizontal compositionality, and contextual equivalence for CompCert. Section 5 describes the Compositional CompCert compiler itself, with a discussion of the effort required to port CompCert’s existing languages and proofs to structured simulations.

2. Interaction Semantics

Interaction semantics (called core semantics in \[6\]) are a protocol-oriented operational semantics of thread interaction, for modeling both multithread concurrent and multimodule programs. Interaction semantics grew from the insight that interaction, between well-synchronized concurrent threads or between modules, can be viewed as occurring exclusively at external function call points, that is, via calls to functions declared in one module but defined in another. This gives a compiler a weakly consistent memory model the freedom to optimize between interaction points.

Here we give a brief overview of interaction semantics, which we build on in later sections to model language-independent linking. First, we describe the interaction semantics protocol. Then, we give representative encodings of interaction semantics, in this case, of CompCert’s Clight and x86 assembly languages. In Compositional CompCert, all language semantics are encoded in this way, as interaction semantics.

2.1 Protocol

In a multithread concurrent program, threads are spawned, take normal (i.e., unobservable) steps, yield at synchronization points (e.g., call to unlock), and eventually halt or (silently) diverge. The same protocol applies to module interactions: a C program is initialized by spawning a new “thread” with a function pointer to main. This sequential thread (which we sometimes call a core) takes normal, unobservable steps by evaluating main or by calling other internal functions defined in the same translation unit, “yields” to the environment by calling external functions defined in other modules, “blocks” until the external function returns, and halts or nonterminates just as a concurrent thread does. At external-call interaction points, nearly anything can happen: For example, the (shared) memory state might be updated arbitrarily by an external function.

Figure 2 summarizes the protocol. Each interaction semantics is parameterized by five types: G is the type of global environments, C is the type of internal, or “core” states. Core states can be instantiated to, e.g., the register file, instruction stream, and processor flags for a language like x86, or to local variable environment and control continuation for a higher-level language like C. M is the type of memories. In the models of the CompCert languages that we employ in Compositional CompCert, M is instantiated to mem, the type of CompCert memories. (In semantic models of program logics such as the Verified Software Toolchain’s Verifiable C \[8\], M is instantiated to a step-indexed model of state used to model function pointer specifications and other higher-order features.) F is the type of external function identifiers. V is the type of values. \(\mathcal{V}\) is usually just CompCert’s value type.

\footnote{Our model is designed to support shared-memory concurrency. Well-synchronized (Dijkstra-Hoare) programs can be proved in Concurrent Separation Logic, and from such proofs we can derive (virtual) permission changes at interaction points \[6\] Ch. 44]. Racy programs can be accommodated as long as racy loads/stores can be modeled via permission changes (future work).}

The five functions at the bottom of Figure 2 together with a few governing axioms (not shown), encode the interaction protocol described above. New cores are initialized with \(\text{initial} \; \text{core} = \rho \; v \; \vec{v}\). The \(v\) is a value, typically a function pointer, while \(\vec{v}\) are the initial arguments to \(v\). initial \; core may fail with None when, e.g., the function spawned is not defined in the global environment \(\text{at} \; \text{external}\) interrogates core state \(c\) to determine whether \(c\) is “blocked” at an external function call interaction point. When \(\text{at} \; \text{external}\) succeeds, it does so with the name of the external function being called (of type \(F\)) and the arguments (of type list \(\mathcal{V}\)). The after \; external function is used to inject the return value of an external call into the calling at \; external core state, producing a new core state as result. halted \; \text{c} returns Some \(v\) with return value \(v\) if \(c\) is halted, otherwise None. corestep gives the small-step internal transition relation of the interaction semantics. We use the syntax \(ge \triangleright v\), \(m \mapsto c', m\) to denote this relation.

2.2 Examples: Clight & x86 Assembly

Figures 3 and 4 give the syntax of Clight and CompCert x86 assembly, the source and target language of Compositional CompCert, respectively. Both languages are adapted from CompCert’s original Clight and x86 and have straightforward operational semantics, which we do not present here (but see the code that accompanies this paper for the complete definitions). Here we focus on the adaptations required to turn these two languages into interaction semantics: First, we give the core, or internal, states for each language; then, we provide an overview of the definitions of the interface functions, e.g., at \; external and after \; external.

Clight’s language of expressions, statements, internal function definitions, and external function declarations is given in the top half of Figure 3. Statements \(s\) are as in CompCert, and rely on a language of expressions \(a\) that includes the usual binary and unary operators. C control-flow constructs like while loops are derived forms (not shown). The semantics of Clight depends on continuations \(\kappa\), described in the figure, and core states \(c\), which come in three varieties: Normal states \(\text{State}_c\), \(f_s \; \kappa \; \rho\), \(\rho\) model the “running” states of a Clight program, during evaluation of anything but function calls, and consist of the name of the current function being executed \(f_s\), the function body \(s\), the control continuation \(\kappa\), and two environments, \(\rho\), \(\rho\), for mapping address–taken stack variables to their locations in memory, and \(\rho\) for mapping temporary variables to their values. CallState \(\vec{v}\) \(\kappa\) models Clight programs that are about to call function \(f\) (either internal or external) with arguments \(\vec{v}\) and continuation \(\kappa\). ReturnState \(v\) \(\kappa\) gives the state that results
after returning from function calls (either internal or external). $v$ is the value returned by the callee; $\kappa$ is the continuation to be executed after the call returns.

Adapting Clight to the interaction-semantics interface is straightforward. For example, here is the definition of Clight after external:

```latex
\text{cl}_{\text{after external}} v_{\text{opt}} \triangleleft \text{cl}_{\text{after external}} v_{\text{opt}}
\begin{cases}
  c \text{ of \ callState } f \;
  \kappa \rightarrow
  \begin{cases}
    \text{case } f \text{ of Internal } \rightarrow \text{None}
    \end{cases}
   \text{case } v_{\text{opt}} \text{ of }
   \begin{cases}
    \text{None } \rightarrow \text{Some } (\text{ReturnState } v_{\text{undef}} \kappa)
    \end{cases}
  \end{cases} \rightarrow \text{None}
\end{cases}
```

First, we check whether $c$ is a CallState, with continuation $\kappa$. If it is, and the function that was being called was external, then we produce a ReturnState with return value $v_{\text{undef}}$ (whenever $v_{\text{opt}}$ was None) and $v$ (whenever $v_{\text{opt}}$ was Some $v$). In all other cases, we just return None.

The definition of initial\_core $ge v v \rightarrow$ is simple, since function arguments are passed not on the stack but abstractly, without reference to memory: we check that $v$ is a valid pointer to a defined function $f$ (by checking that the arguments $\rightarrow$ are defined and match $f$’s type signature, then introduce state CallState (Internal $f$) $\rightarrow$ Kstop, which immediately steps to the body of function $f$ with the local initial variable environment $\rho_v$ that maps the function’s formal parameters to its arguments $\rightarrow$. The definitions of initial\_core in the languages below Clight follow a similar regime—all the way down to CompCert’s Linear language, which uses an environment of abstract locations such as incoming parameter stack slots to represent the state of the stack and registers.

Adapting x86 assembly (Figure 4) is a bit trickier, since arguments must be passed concretely, on the stack. (The same applies to CompCert’s Mach language.) As we will see in Section 3, we use the initial\_core function of the interaction semantics interface to model both program initialization (i.e., by the loader) and the function calls that occur at cross-module function invocations. If we knew that all modules in our program were written in x86 assembl and used, e.g., the standard cdecl calling convention, then modeling cross-module invocations would be less of an issue. The shared calling convention would mean that arguments to one function (say, $B.g$) would be placed by a caller $A.f$ on the stack or in registers exactly as expected by $B.g$.

But the restriction to a shared calling convention/ABI is rather limiting. We want to be able to model, at least abstractly, the interactions of modules in a variety of languages, at both higher and lower levels of abstraction. To accomplish this, we apply a “marshalling” transformation to the x86 language: To initialize a new x86 core calling function $id$ with arguments $\rightarrow$, we produce state MarshallState$_{id}$ $\rho_v \rightarrow$, which immediately steps to a running State. As a side effect of this step, however, we allocate a “dummy” stack frame in memory in which we store the incoming arguments $\rightarrow$, in right-to-left cdecl order as expected by CompCert and gcc. (MarshallState$_\text{out}$ performs the symmetric step of marshalling arguments out of memory.)

Concretely, for x86 modules all sharing the same calling convention, this modeling step does not occur on a real machine (nor does the compiler output any “marshalling” code). But by sticking to the abstract “calling convention” imposed by initial\_core, in which values are passed abstractly instead of in memory and registers according to a particular calling convention, we gain flexibility to model the interactions of modules in a wide variety of languages, not only Clight and x86 but also x86 modules following different calling conventions, such as, e.g., cdecl and Microsoft fastcall.
3. Linking and Contextual Equivalence

In the previous section, we introduced interaction semantics as a means of interpreting the behavior of isolated modules. In this section, we define an abstract operator \( L([S_0], [S_1], \ldots, [S_{N-1}]) \) over interaction semantics that defines the linked behavior of a set of interacting modules, as given by a multimodule program \( P = S_0, S_1, \ldots, S_{N-1} \).

As input, \( L \) takes \( N \) interaction semantics, each with (perhaps) a different global environment and core state type (i.e., modules programmed in perhaps different languages). The output of \( L \) is a new interaction semantics \( [P] = L([S_0], [S_1], \ldots, [S_{N-1}]) \) that models the execution of the linked program by maintaining as its own core state a (heterogeneous) stack of the modules’ core states. Each “frame” on the stack corresponds to a runtime invocation of one of the modules in the program. Cross-module function calls result in new cores being pushed onto the stack (initialized via \( \text{initial}_c \)).

The modules \( S_i \) are written in different languages, whose states may have different (Coq) types. In order to treat these modules uniformly in \( L \), we wrap their interaction semantics by existentially quantifying over the core state types of each module, an operation we encapsulate in the type \( \text{Modsem} \).

\[
\text{Modsem} \triangleq \begin{cases} 
F, V, C : \text{Type} \\
\text{ge} : \text{Genv F V} \\
\text{sem} : \text{Semantics (Genv F V)} \cap \text{mem}
\end{cases}
\]

In this dependently typed record, the types of \( \text{ge} \) and \( \text{sem} \) depend on \( F, V, \) and \( C \). This module is written in programming language \( F \) (e.g., Clight or x86), whose global variables have type-specification language \( V \) (e.g., Clight types or units); and whose core states have type \( C \) (e.g., Clight nonaddressable locals and control stack, or x86 register bank). We also existentially bind the global environment \( \text{ge} \) that was statically initialized for this module. It maps addresses to global variables and function-bodies.

The final component is \( \text{sem} \), an interaction semantics. It defines the interface functions \( \text{initial}_c, \text{at_external}, \text{etc.} \), as well as a step relation \( \text{ge} \vdash c, m \rightarrow c', m' \). Modules in the same language will typically have identical \( \vdash \cdot \rightarrow \cdot \) relations, specialized by different \( \text{ge} \) components that map disjoint sets of addresses to internal function bodies (as opposed to external function declarations). In what follows, we use \( \vdash \) to refer interchangeably to the interaction semantics of modules and their \( \text{Modsem} \) wrappers.

Figure 5. CALL case of the linking corestep relation. Gray dashed boxes are “stacks-of-cores.” Core state \( c \) is at \( \text{at_external} \) calling function \( idy \). Assuming \( \text{l.plt idy} = \text{Some idx} \) and \( \text{ge idy} = \text{Some b} \), initializing module \( \text{idx} \) with function pointer \( \text{Vptr b} \) results in initial core \( c' \) being pushed onto the stack.

\[
\begin{align*}
\text{ge} \vdash & c, m \implies c', m' \\
\text{ge} \vdash & l, m \implies l \text{ with } \{ \text{stack} := \text{push } c' \text{ (pop l.stack) } \}, m' \quad \text{(STEP)} \\
\text{ge} \vdash & l, m \implies l \text{ with } \{ \text{stack} := \text{push } c' \text{ (pop l.stack) } \}, m \quad \text{(RETURN)}
\end{align*}
\]

The output of \( L \) is an interaction semantics in the \( \text{LinkedState} \) “language.” \( \text{LinkedState} \) is parameterized by \( \text{modules} \), a map from module indices in the range \([0, N)\) to module semantics, where \( N \) is the (nonzero) number of translation units in the program.

\[
\text{Core} (N : \text{pos}) (\text{modules} : I_N \rightarrow \text{Modsem}) \triangleq \begin{cases} 
\text{idx} : I_N \\
\text{core} : C \text{ (modules idx)}
\end{cases}
\]

Core models the runtime state of a sequential execution thread. \( I_N \) is the (dependent) type of integers in range \([0, N)\). The idx of a Core is the index of the module from which the core was initialized.

The runtime state of a linked program is then:

\[
\text{LinkedState} (N : \text{pos}) (\text{modules} : I_N \rightarrow \text{Modsem}) \triangleq \begin{cases} 
\text{plt} : \text{ident} \rightarrow \text{option } I_N \\
\text{stack} : \text{Stack} (\text{Core } N \text{ modules})
\end{cases}
\]

The two fields of \( \text{LinkedState} \) are: the procedure linkage table \( \text{plt} \)—mapping function names (type ident) to the indices of the modules in which the functions are defined, if any (option \( I_N \))—and a stack of cores. We model the \( \text{plt} \) as a field in the \( \text{LinkedState} \) record, as opposed to deriving it from \( N \) and \( \text{modules} \), to retain flexibility to do dynamic linking in the future. The stack is always nonempty; all cores except the topmost one are \( \text{at_external} \) (forall \( c \in \text{modules} \)).

Figure 6 gives the step relation. There are three rules. The \text{step} rule deals with the case in which the topmost core on the call stack \( (c = \text{peek l.stack}) \) takes a normal internal step \((\text{ge c} \vdash c, m \rightarrow c', m')\). \( \text{ge c} \) is the global environment associated with the module from which \( c \) was initialized. In this case, we just propagate the new core state \( c' \) and memory \( m' \) to the result state of the overall linking judgment. The notation \( l \text{ with } \{ \text{stack} := \text{push } c' \text{ (pop l.stack) } \} \) updates the topmost core state on the stack. For readability, we elide the operations required to propagate the idx field of Core records.

The second rule, \text{call}, handles the case in which the topmost core on the stack is \( \text{at_external} \) (forall \( c \in \text{modules} \)). This makes a cross-module function call. In this case, we use the \( \text{initial}_c \) function of the module semantics that defines function \( idy (l.\text{plt idy} = \text{Some idx}) \) to initialize a new core state to handle the function call (\( \text{initial}_c (\text{modules idx}) \)).
The \textsc{return} rule models external function returns. Here, the
core state \(c\) is halted with return value \(v\) (\(i\ \text{with}\ \{\text{stack := push c' l.stack}\}\)). To resume execution, we use the \textit{after external}
function exposed by the caller’s semantics \(c' = \text{peek (pop l.stack)}\) to
inject the return value \(v\) (\textit{after external} (\text{Some} \(v\) \(c'' = \text{Some} \(c''\))). State \(c\) is then popped from the stack, and state \(c'\) is
updated to \(c''\) (\(i\ \text{with}\ \{\text{stack := push c'' (pop pop l.stack)}\}\)).

The stack is an abstraction of the activation-record stack of a
C or assembly program. Internal calls (within one module) do not
push on our stack; they transition from one core (and memory) to
another core (and memory) within the same top stack element. But
of course this core/memory may be the abstraction/implementation
of pushing and popping (module-local activation records). Differ-
ent modules may or may not share a “real” activation-record stack.

The final piece of linking semantics is the definition of the inter-
face functions: initial\_core, at\_external, after\_external, and halted.
We do not have space to give the full definitions here (they
are in the Coq code); instead, we briefly describe them.

A linking semantics is initialized (initial\_core) by spawning a
new core to handle the entry point function that was called. The
linking semantics is at\_external when the topmost core on the stack
is at\_external calling a function defined by one of the modules
(otherwise, we would have initialized and pushed a new core to
handle the function). To inject a return value into linker states
(after\_external), we inject the value into the topmost core state on
the stack (after\_external \(\text{opt c = Some} \ c\)). Finally, a linking semantics is halted when
the stack contains a singleton halted core state (halted \(c\) and size \(l.\text{stack} = 1\), i.e., the topmost core is halted
but has no return context.

**Contextual Equivalence** Linking semantics leads to a natural no-
tion of semantic context: Take program contexts \(C\) to be arbitrary
module semantics Modsem. Then the application of a program
case to an (open) multimodule program \(P\) is just the semantics
that results from linking the program with that context: \(L(C, [P])\).

This notion of context-as-interaction-semantics is quite gen-
eral: it supports the definition of program contexts in arbitrary lan-
guages, e.g., C and x86 but also Coq’s Gallina. (It is straightforward
to define an interaction semantics whose step relation is an
arbitrary Coq relation; the code that accompanies this paper in-
cludes one such example.)

Eliding some details related to global environments and mem-
ories, contextual equivalence of two open multimodule programs
\(P_S\) and \(P_T\) may be defined as equitermination (halted in inter-
action semantics)—when initialized at matching entry points—in all
contexts:

**Definition 1** (Contextual Equivalence), \(P_S \sim P_T \triangleq \forall C, L(C, [P_S]) \Downarrow \iff L(C, [P_T]) \Downarrow\)

The context \(C\) observes the state of memory (and the arguments
to external calls) when the program interacts with the environment.
To distinguish \(P_S\) and \(P_T\), \(C\) can, e.g., get stuck (as opposed to
safely terminating) at one of these interaction points if the memory
state and arguments fail to satisfy a specified predicate.

4. **Structured Simulations**

Section 3 showed how to define the semantics of open multimodule
programs, and what contextual equivalence meant in that setting.
Now we show how to prove contextual equivalences for CompCert.
We briefly review logical simulation relations \([6]\), and then show
our new enhancement, \textit{structured simulations}.

LSRs established compiler correctness by showing that compi-
lation preserved the protocol structure of interaction semantics,
using CompCert’s original match relations \(\sim_f\) with memory in-
jections \(f\) to relate source and target states. For internal execu-

**Figure 7.** Structured injections.

<table>
<thead>
<tr>
<th>(\mu)</th>
<th>(\text{own}_S)</th>
<th>(\text{own}_T)</th>
<th>(\text{fus})</th>
<th>(\text{fshem})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mu)</td>
<td>Priv</td>
<td>Pub</td>
<td>Frgn</td>
<td>Invis</td>
</tr>
</tbody>
</table>

\(\text{Own}_{\text{struct}}\) injection : Type \(\triangleq\)

\(\begin{align*}
\text{own}_S & : \text{block} \to \text{Ownership} \\
\text{own}_T & : \text{block} \to \text{Ownership} \\
\text{fus} & : \text{block} \to \text{option} \ (\text{block} \times \mathbb{Z}) \\
\text{fshem} & : \text{block} \to \text{option} \ (\text{block} \times \mathbb{Z})
\end{align*}\)

\(\begin{align*}
\text{owned} & : \{ b \mid \text{own}(b) \in \{\text{Priv, Pub}\}, i \in \{S, T\} \} \\
\text{shared} & : \{ b \mid \text{own}(b) \in \{\text{Pub, Frgn}\}, i \in \{S, T\} \} \\
\text{vis} & : \text{owned} \cup \text{shared}, i \in \{S, T\} \\
\text{f} \ s X & : \lambda b.\ \text{if} \ b \in X \ \text{then} \ f \ b \ \text{else} \ None \\
\mu \ s X & : \mu \ \text{with} \ \{ \text{fus} := \text{fus} \} \ \{ \text{fshem} := \text{fshem} \}
\end{align*}\)

\(\text{Ownership} \triangleq \text{Priv} \mid \text{Pub} \mid \text{Frgn} \mid \text{Invis} \mid \text{None}\)

3For example, if a source-language variable is represented in memory on
the stack, and in the translation to intermediate language the compiler
chooses to use a register (unaddressable local variable) instead, then we
say this memory region is removed by the compiler.
block → Ownership, which map blocks (in the source and target memories, respectively, of a related pair of program states) to values of an inductive Ownership type; and two CompCert-style memory injections: \( i_{\text{fn}} \) and \( f_{\text{inj}} \). \( i_{\text{fn}} \) records the source-target mapping of blocks that were allocated by the current module; \( f_{\text{inj}} \) maps external blocks (those allocated by other modules).

The Ownership modes are: Priv, for memory regions (blocks) allocated by the module being compiled but which haven’t been leaked to the environment; Pub, for allocated blocks that have been leaked at a previous interaction point; Frgn, for foreign blocks leaked into \( \mu \) at external calls; Invis, for blocks that have been allocated (by another module) but not leaked in; and None for blocks that may not yet have been allocated. A block is (locally) owned by \( \mu \) in the source or target memory when \( \text{own}_S(b) \) (resp. \( \text{own}_T(b) \)) is either Pub or Priv. External blocks in source and target are those mapped by \( \text{own}_S(S, T) \) to Frgn or Invis. Likewise, a block is shared if its ownership is either Pub or Frgn. The visible source blocks of \( \mu \) are those in the set \( \text{vis}_S \subseteq \text{owned}_S \cup \text{shared}_S \) (and likewise for \( \text{vis}_T \)). We use notation \( \text{foreign}(S, T) \) and \( \text{pub}(S, T) \) to denote memory regions that are foreign and public ownership, respectively. The \( \text{foreign} \) and \( \text{pub} \) blocks with foreign and public ownership, respectively.

We track ownership of blocks, rather than ownership byte-by-byte, because the CompCert languages and memory model permit pointer arithmetic within blocks. Once a location within a block has been made public, the whole block is made public as well.

Complementing the data in Figure 7 are axioms that ensure proper interaction of ownership, leakage, and compilation. These axioms (not shown) ensure that \( i_{\text{fn}} \) and \( f_{\text{inj}} \) (1) operate exclusively on blocks of appropriate ownership (i.e. \( f_{\text{inj}} \) only maps owned blocks, to owned blocks, and similarly for \( i_{\text{fn}} \) and external blocks); and (2) are total on their portion of shared blocks: \( f_{\text{inj}} \) must map all Pub blocks, and must map them to Pub blocks, and similarly for \( i_{\text{fn}} \) and Frgn. The result is that blocks which have been leaked to/from the environment in one compilation stage cannot be removed by later stages.

At interaction points between a module and its environment, we adjust the structured inclusions so that (at these points) the shared regions are closed under pointer arithmetic and dereferencing (there are no pointers from the shared to the nonshared region). We maintain as an additional invariant that the source visible set \( \text{vis}_S \) is always closed under pointer dereferencing and pointer arithmetic.

**Simulation Structures** Figure 8 presents the two core clauses of structured simulations \( E \), those for internal (i.e. unobservable) steps (Internal Steps) and for external interactions with the environment (External Steps) (the clauses for initial_core, at_external, and halted are not shown). The structure of the internal diagram is familiar from traditional forward simulation proofs: Assume we are in matching initial states \( c, m \) \( \sim_{\mu} (d, tm) \) and we take a source step \( g_{\text{E}} \vdash c, m \rightarrow c', m' \) with effect \( E_S \). Then there exists a matching \( d', tm' \), and Kripke-extended structured injection \( i_{\text{fn}} \) such that \( g_{\text{E}} \vdash d, tm \rightarrow (c', m') \sim_{\mu'} (d', tm') \). Clause (1) (Kripke extension, \( \mu \subseteq \text{vis}_S \mu' \)) says that \( \mu' \) may map more owned blocks than \( \mu \) (in order to deal with allocations) but otherwise is equal to \( \mu \). Clauses (2) and (3) are side conditions that are not important for understanding the key ideas:

Clause (6) is the guarantee condition. Clause (6a) asserts that the target effects \( E_T \) are contained in \( \text{vis}_T \), assuming that \( E_S \subseteq \text{vis}_S \mu \). In other words, the compiler preserves the property of "writing and freeing only to visible regions."

Clause (6b) guarantees that writes to (and frees of) memory locations in the target that are not owned by \( \mu \) (owned_to \( \mu b = \text{false} \) can be "tracked back" to corresponding writes and frees in the source \( (\exists b, \delta, i_{\text{fn}}(b) = \text{Some}(b, \delta) \) and \( (b, z_1 - \delta) \in E_S) \). Writes/frees of locations in blocks owned by the module being compiled are always permitted, which enables the compiler to introduce reloading code (for spilled variables) or to add function prologue/epilogue code that saves/restores callee-save registers.

The \( E_S \) and \( E_T \) that appear in clause (5) and in step judgments are effect annotations. For example, \( g_{\text{E}} \vdash c, m \xrightarrow{E_T} c', m' \) means: configuration \( c, m \) steps to \( c', m' \), writing to or freeing exactly the locations \( E_S \). Locations not contained in this set are guaranteed not to be modified. We state these "does not modify" guarantees intentionally in this way, as effect annotations, in order to prove vertical composition. The problem with a more extensional interpretation of effects (e.g., as input–output "unchanged on" conditions) is that effect union is no longer monotone: If a program takes two steps, from \( m \) to \( m'' \) with effect set \( E_1 \) and from \( m'' \) to \( m' \) with effect set \( E_2 \), with overall extensional effect \( E \), it may be the case that \( E_1 \cup E_2 \not\subseteq E \) if, for example, the second step restored a value that was overwritten by the first step.

The external step diagram occupies the bottom half of Figure 8. It relates an at_external source-target configuration pair \( (c, m) \rightarrow_{\mu} (d, tm) \) with the after_external configuration pair \( (c', m') \sim_{\mu'} (d', tm') \) that results from making an external call. The basic premise is: For any source–target return values \( v_s, v_t \), return memories \( m' \) and \( tm' \), and structured injection \( v_t \) satisfying the listed conditions, it’s possible to inject \( v_s \) and \( v_t \) into states \( c \) and \( d \), resulting in the new states \( c' \) and \( d' \) which match in \( \mu' \), \( m' \) and \( tm' \) \( \sim_{\mu'} (d', tm') \). The \( \mu \) dual to the \( \text{vis}_S \mu \) condition used in the internal step diagram. It says that \( \nu' \) may
Figure 9. Graphical representation of the structured injection leakage operations. The thick black arrows are pointers in memory. The white private and light gray public boxes are owned (“us”) blocks. The dark gray boxes are foreign (“them”) blocks. The black box is an Invis memory region that was allocated by another module but not yet leaked in. The leak-in operation marks as public a private region reachable from a public pointer.

Reachability

reach : mem \rightarrow set\ block \rightarrow list \rightarrow set\ block
reach m R nil \triangleq R
reach m R ((b', z') :: L) \triangleq
\{ b' | b' \in reach m R L \land \text{perm}\ m b' z' = \text{Readable} \land \exists z. m(b', z') = \text{Vptr} (b, z) \} 
REACH m R \triangleq \{ b | \exists L. b \in reach m R L \}

Export/Import

export, \mu B : \text{StructuredInjection} \triangleq \mu \text{with} \{ \text{own}_i := \lambda b. \text{if } b \in B \text{ then } \text{Pub else own}_i, \mu b \}, i \in \{ S, T \}
import, \mu B : \text{StructuredInjection} \triangleq \mu \text{with} \{ \text{own}_i := \lambda b. \text{if } b \in B \text{ then } \text{Frgn else own}_i, \mu b \}, i \in \{ S, T \}

Leakage

\text{leak}_{out} \mu \langle v, v' \rangle L \mu m \text{ tm} : \text{StructuredInjection} \triangleq
\text{let } L_S \triangleq \text{REACH } m (\text{blocksOf } \langle v, v' \rangle) \cup \text{shared}_S \mu \land \text{owned}_S \mu
L_T \triangleq \text{REACH } \mu (\text{blocksOf } \langle v, v' \rangle) \cup \text{shared}_T \mu \land \text{owned}_T \mu
\text{in } \text{export}_T (\text{import}_S \mu L_S) L_T
\text{leak}_{in} \mu v_s v_t m \text{ tm} : \text{StructuredInjection} \triangleq
\text{let } L_S \triangleq \text{REACH } m (\text{blocksOf } \langle v_s \rangle) \cup \text{shared}_S \mu \land \text{owned}_S \mu
L_T \triangleq \text{REACH } m (\text{blocksOf } \langle v_t \rangle) \cup \text{shared}_T \mu \land \text{owned}_T \mu
\text{in } \text{import}_T (\text{export}_S \mu L_S) L_T

Figure 10. Reachability and leakage.

map more external blocks than \nu—in order to deal with allocations performed by the environment—but otherwise is equal to \nu. The other nonbolded conditions are adapted from CompCert, and follow in our Coq proofs directly from symmetric conditions on the match-state relation and the internal step diagram.

The conditions listed in bold together compose the structured simulation rely. The predicate \text{unchanged}_{on} U m m' specifies that memories m and m' are equal (same contents and permissions) at the locations in set U. In the source execution, we use \text{unchanged}_{on} \{ (b, z) | \text{own}_S \nu b = \text{Priv} \} m m' to ensure that m and m' are equal at locations in the private blocks of the injection \nu, which is built from \mu by updating leakage information as described below. The target-execution condition \text{unchanged}_{on} (\text{local}_{out}\text{of}\text{reach } \nu \text{ m} ) \text{ tm tm'} says that tm and tm' are equal at owned target locations that either (1) do not correspond to readable source locations, or (2) are mapped from private source locations. By using \text{unchanged}_{on} here, we stipulate the nonmodification conditions of the rely extensionally.

The structured injection \nu is built from \mu—the injection that originally related at external states \langle c, m \rangle \sim_{\mu} \langle d, tm \rangle—using the \text{leak}_{out} function depicted graphically in Figure 9 and defined in Figure 10. The idea is: \text{leak}_{out} “leaks” to the public (other modules) blocks that are reachable by following pointer paths either from the arguments \langle v, v' \rangle to the external call (blocksOf \langle v, v' \rangle) or from blocks that were previously shared (shared, \mu). This is a consistency condition: It says that structured simulations may not assume anything about the contents of leaked blocks (the unchanged_on conditions that form the rely satisfied by the environment apply only to private blocks). The functions reach and \text{REACH} defined at the top of Figure 10 calculate the transitive closure of the points-to relation on CompCert memories. In the definition of \text{leak}_{out}, we use the auxiliary function export to update the ownership functions of an injection \mu to map blocks in the reachable set to Pub.

The \text{leak}_{in} function used to define \mu' at the end of the external step diagram plays a role analogous to that of \text{leak}_{out}, except that here, we are leaking into \mu' new foreign blocks reachable from the return value \nu_v of the external call. Likewise, the import function is almost equivalent to export, except that it updates the ownership functions of a structured injection to map the blockset B to Frgn as opposed to Pub.

Restriction The final consistency condition is: the simulation relation \sim_{\nu} is independent of the Invis (and None) blocks. This condition is used, e.g., in the proof of vertical composition (transitivity, Theorem 4). Technically, we enforce Invis-independence by requiring that \sim_{\nu} be closed under restrictions to reach-closed supersets of the visible blocks, an operation defined in Figure 7 as \mu \{ X \mid X \text{ with } X \text{ a block set}, \mu \{X \mid X \} \text{ denotes the structured injection obtained by restricting the maps } f_m \text{ and } f_{func} \text{ to the domain } X \}. The closure condition says that if \langle c, m \rangle \sim_{\mu} \langle d, tm \rangle, then \langle c, m \rangle \sim_{\nu_v \mid X} \langle d, tm \rangle for any block set X that contains at least \nu_v, and is closed under pointer dereferencing and arithmetic in m.

5. Main Results: Compositionality and Contextual Equivalence

Vertical Composition (Transitivity). One can compose compiler phases (Figure 13). The proof that structured simulations compose vertically follows the same outline as that of LSRs [6]. As discussed in Section 4, the proof of transitivity of the internal-step diagram is tightly dependent on our treatment of effect annotations. Proving transitivity of the external-call clause (lower half of Figure 8) requires the construction of an interpolating after_external memory \mu' in the intermediate execution between source and target.

Theorem 1 (Transitivity). Let \mu_L, \mu_L, and \mu_L be effect-annotated interaction semantics. If \mu_L \leq \mu_L is a structured simulation from \mu_L to \mu_L and \mu_L \leq \mu_L is a structured simulation from \mu_L to \mu_L, then there exists a structured simulation \mu_L \leq \mu_L from \mu_L to \mu_L.

Horizontal Composition (Linking) The second kind of compositionality is horizontal: We would like to know that composing the simulation relations established by independently compiling the modules in a program results in an overall simulation between the (linked) module source and target programs. We give the theorem statement first, then explain some of the subtleties, in particular, the restriction to reach-closed semantics, which enforces the single-program conditions corresponding to the structured simulation guarantees of Section 4.

Theorem 2 (Linking).
• If $P_S = S_0, S_1, \cdots, S_{N-1}$ is a multimodule program with $N$
translation units, each of which is reach-closed, and

• $P_S$ is compiled to $P_T = T_0, T_1, \cdots, T_{N-1}$ (possibly by $N$
different compilation functions) such that $[S_i] \leq [T_i]$ for each
source–target pair, then

• there is a simulation relation $L([P_S]) \leq L([P_T])$ between the
source and target programs that result from linking the $S_i$ and
independently linking the $T_i$.

The $\leq$ in the theorem denotes forward simulation on whole
programs, as in Figure 1 but without clauses (1-3) in the internal step
diagram, and without clauses for at_extern and after_extern.
As Corollary 1 will show, establishing $\leq$ is sufficient for proving
textual equivalence of open multimodule programs.

The restriction to reach-closed semantics (Figure 11) is best
motivated with an example. Consider the following program:

```
//Module A
void g(void);

//Module B
.globl _g
void h(int, x) {};
.globl g
void f(void) {
    int a=0;
    if (a) (h(&a));
    g();}
void main(void){f();}
```

in which A.f calls (assembly function) B.g, passing no arguments.
The strange bit is B.g: All it does is write the value 42 into mem-
ory at address 0x28ac1c, which just happens to be the address at
which local variable a is allocated on the stack in Windows.

Now imagine we compile module A through a compiler phase
like dead code elimination, which results in a (which was pre-
viously addressed in dead code if (a) h(&a);) and therefore
stack-allocated) being removed from memory. Since 0x28ac1c is
a’s address before dead code elimination, the unoptimized program
above does not get stuck (the write succeeds, to location a, with-
out significant effect). After optimization, the program will fail,
probably because the write to a now overwrites the return address
stored in g’s stack frame. (Incidentally, this program succeeds when
compiled with gcc -00 but seg-faults under gcc -O1. This is
not a bug in gcc; instead, it is evidence that the gcc developers
agree with us that Module B is an ill-formed program context.)

One might object that if (a) h(&a); is actually not dead
code, because it results in a being stack-allocated, which in turn
results in the safe execution of the (admittedly contrived) overall
program. But that way madness lies. The point of a compositional
compiler is to enable local modular compilation, which should de-
pend only on translation-unit-local analyses. Correctness of opti-
mizations like dead code elimination should be independent of the
larger program context in which a module is executed.

The challenge, then, is coming up with a characterization of the
source modules $S_0, S_1, \cdots, S_{N-1}$ that does admit linking as in
Theorem 2. We do this in general, for arbitrary interaction
semantics, by observing that the write to 0x28ac1c is ill-formed not
because it goes wrong (though it will lead to going wrong in most
program contexts) but because it’s a write to a location that the
assembly program shouldn’t have known about in the first place.
Put another way, address 0x28ac1c was not reachable via pointer
arithmetic even from g’s initial arguments, from global variables,
or from the return values of external calls g may have made previously.

This condition—no writes or frees to locations that are not

---

4 It might seem strange to say “not reachable via pointer arithmetic” in
the context of an assembly program, since in most assembly models the
entire address space is “reachable”. Here we mean not reachable in the
instrumented semantics of x86 assembly used by CompCert, in which
memory is allocated in blocks, as in CompCert’s Clight, and interblock
pointer arithmetic is disallowed.

5 Another condition is also required: The $T_i$ never store invalid
(not yet allocated) pointers into memory. This condition is true for all contexts
we care about (for example, it holds of all programs in CompCert’s x86
language, which does not permit storing invalid addresses into memory).
the return values of external calls (blocks of $\nu$). \texttt{initial\_core} (not shown) initializes the set with the blocks exposed by the initial arguments. \texttt{at\_external} and \texttt{halted} are simple wrappers.

As a corollary of Theorem 2 we get the following contextual equivalence result when the source modules are reach-closed, stated in terms of a variation of Definition 1 in which contexts satisfy a few additional properties.

**Definition 2** (Reach-Closed Contextual Equivalence).

$$P_S \trianglerighteq_{rc} P_T \iff \forall C_{\text{src}}, \ C \preceq C' \wedge \exists C' \wedge \text{safe} \mathcal{L}(C, [P_S]) \rightarrow \mathcal{L}(C, [P_T]) \downarrow \iff \mathcal{L}(C, [P_S]) \downarrow$$

**Corollary 1** (Simulation Implies Contextual Equivalence). Let

- $P_S = S_0, S_1, \ldots, S_{N-1}$; and
- $P_T = T_0, T_1, \ldots, T_{N-1}$

for reach-closed source modules $S_0, S_1, \ldots, S_{N-1}$. If for each $i$, $[S_i] \preceq [T_i]$ and $t_i$ is deterministic, then $P_S \trianglerighteq_{rc} P_T$.

In the above, we assume closing contexts $C$ (those that do not themselves call external functions not defined by any of the modules; callbacks into $P_S$ and $P_T$ are permitted). Safety of the source linked program and determinism of the target modules are required to prove the backward direction of the equivalence (the forward direction holds without these assumptions). The $C' \preceq C$ condition says that $C$ commutes with memory injections: If $C$ is initialized twice with injected arguments, both executions either go wrong, nonterminate, or equiterminate with injected results. Although this condition follows directly from the form of Theorem 2, it is strongly motivated: We should not allow contexts to distinguish source and target programs based solely on bijective renamings of memory blocks exposed to the context (pointer arithmetic is not allowed between blocks, only within blocks). The consistency conditions on structured injections and simulations that we described in Section 2 mean that in the proof of $C \preceq C'$, the context may assume that all public blocks leaked by the program are mapped from source to target (they are never removed during compilation of the program).

The main difficulty in proving Theorem 2 and, by extension, Corollary 1 is devising a simulation invariant to relate the stacks-of-cores runtime states of the linked programs $P$ and $T(P)$. The situation is presented schematically in Figure 12. In the source linked program, we have a stack of core states, growing downwards, with $c$ in callee position with respect to a (direct or indirect) caller core $c_0$, which may be implemented in a different language. We must relate this stack of cores to the corresponding stack in the target linked program. We use $\mu$ to denote the structured simulation that relates the callees $c$ and $d$, and $\nu$ to denote the injection that relates callers $c_0$ and $d_0$. For simplicity, we elide the memories (for callers, the memory at the call point is existentially quantified). A caller core may be a callee with respect to another caller higher on the callstack.

The key rely–guarantee condition is to ensure that blocks labeled as foreign, or leaked-in, by callee injections $\mu$ are always labeled as public by caller injections $\nu$:

$$\text{foreign}_S \mu \cap \text{owned}_S \nu \subseteq \text{public}_S \nu \quad (1)$$

From the fact that source modules are reach-closed we then can show that the memory effects of the running callee core at the top of the callstack are confined to callee-allocated (owned) and foreign blocks. This implies that private caller memory regions in $\nu$, which are disjoint from the blocks marked as public by $\nu$, remain unmodified.

A difficulty here is how to relate the root sets of source modules to the visible sets $\text{vis}_S$ used in the simulation relations. We do this by maintaining the following two invariants:

$$\text{roots} \geq r \subseteq \text{vis}_S \mu \quad (3)$$

$$\text{REACH} \ m \ (\text{vis}_S \mu) \subseteq \text{vis}_S \mu \quad (4)$$

Invariant (3) says that the root set of the source semantics is a subset of the visible source blocks in $\mu$. This invariant holds initially, when $r$ is first created, and is maintained at external function calls and returns. Condition (4), which we maintain as an invariant of all structured simulations, says that the visible set is closed under reachability. These two conditions, plus (1) and monotonicity of the \texttt{REACH} relation, imply that $E_S$ is a subset of $\text{vis}_S \mu$. This fact, together with condition (1) above, is sufficient to prove the unchangeable-on relies of Figure 13 at the point at which the running core returns to its calling context.

**6. Compositional CompCert**

The proved-correct phases of the Compositional CompCert compiler are shown in Figure 13 with optimization phases in gray. The main differences with standard CompCert are: (1) We compile Clight to x86 assembly, whereas standard CompCert compiles a slightly higher-level language (CompCert C) to multiple assembly targets (x86, PowerPC, and ARM); and (2) standard CompCert
The process of making the proof of a transformation phase compositional typically proceeded as follows: We refined CompCert’s internal match-state notion $\sim$ (and the auxiliary relations for activation records, frame stacks, etc.) to relations $\sim_\mu$ indexed by structured injections. In particular, because external function call interactions may introduce memory regions related by memory injections in Compositional CompCert, the simulation relations of phases that were previously proved as memory equality or memory extension phases had to be reformulated as injection phases. Particular care was needed to assign correct ownership and visibility information to compiler-introduced memory blocks.

In addition, we had to add to each $\sim_\mu$ relation the clauses: $\forall s, \mathsf{vis}_s$, is closed under reachability, and the relation $\sim_\mu$ is closed under restriction to the visible set ($\mu |_{\mathsf{vis}_s}$). To ensure that global blocks were always mapped by each compiler phase, we treated them as $\text{Frfgn}$ to all modules. While the addition of these extra invariants proceeded in a mostly uniform manner across all phases, the refinement of $\sim$ to $\sim_\mu$ was phase-by-phase, due to the considerable internal differences between the various CompCert phases.

An issue that required special attention was the treatment of compiler builtins. Here we had to sharpen the distinction between, on the one hand, processor-specific 64-bit helpers and $\text{memcpy}$—these functions are typically inlined, and should never yield at external despite being axiomatized as external calls by mainline CompCert—and true external functions, which are never inlinable, on the other. To sharpen this distinction, we modified the definition of the $\text{Cminorgen}$ language to ensure that the compiler-introduced calls to 64-bit helpers were unobservable.

All in all, porting the CompCert phases in Figure 13 to structured simulations took approximately 10 person-months, though much of this time was spent at the “boundaries” of the proof, updating the interfaces that connected, in particular, our linking semantics and proofs to structured simulations. In general, the porting time decreased as the project went on. Adapting the first few phases of the compiler took a few weeks to a month per phase, whereas the later phases went much more quickly (a day or two per phase). This was due in part to greater familiarity with CompCert, whereas the later phases went much more quickly (a day or two per phase).

Proof. By Corollary 2 Theorem 3 and determinism of CompCert x86 assembly.

Theorem 3 (Compiler Correctness). Let CompCert denote the compilation function that composes the phases in Figure 13 in order. If $\text{CompCert}(S) = \text{Some } T$, for Clight module $S$ and x86 module $T$, then $[S] \preceq \llbracket T \rrbracket$.

Proof. By transitive composition of the simulation proofs for the individual phases in Figure 13 using Theorem 1.

Corollary 2 (Compositional Compiler Correctness). Let $S_0, S_1, \ldots, S_{N-1}$ be a set of reach-closed Clight modules such that $\text{CompCert}(S_i) = \text{Some } T_i$, for each $i$. Then $S_0, S_1, \ldots, S_{N-1} \sim_{\mu T_0, T_1, \ldots, T_{N-1}}$.

Proof. By Corollary 1 Theorem 3 and determinism of CompCert x86 assembly.

7. Related Work

Compiler verification is one of the “big problems” of computer science, as evidenced by the large body of research it has spawned in the 45 years or so since McCarthy and Painter [20]. We cannot hope to give a comprehensive survey here (but see [9]). Instead, we focus on the most closely related work.

Verified Whole-Program Compilers  Moore [22] was one of the first to mechanically verify a programming language implementation (a compiler for a language called Pilon). The most well-known work in this vein since Moore is Leroy’s CompCert C compiler in Coq [14], upon which Compositional CompCert is based. Chlipala has also built verified compilers in Coq—first, from lambda calculus to idealized assembly language [17], and then, later, for an impure functional language [8]. But both Chlipala and Leroy’s compilers were limited to whole programs—they did not provide correctness guarantees, as we do in this work, about the behavior of separately compiled multimodule programs.

Compositional Compilation  Benton and Hur were two of the first to explicitly do compositional specification of compilers and low-level code fragments, first for a compiler from a simply typed function language to a variant of Landin’s SECD machine [4], then for a functional language with polymorphism [5]. Benton and Hur’s work was followed by a string of papers—by Dreyer, Hur, and collaborators—that resulted in refinements of the basic techniques (step-indexed logical relations and biothogonality). The refinements included extensions to step-indexed Kripke logical relations, for dealing with state in the context of more realistic ML-like languages [13], and more recently, to relation transition systems (RTSs) [11] and the related parametric bisimulations [12]. RTSs demonstrated that it was possible to do bisimulation-style reasoning in the possible-worlds style of Kripke logical relations and state transition systems; parametric bisimulations refined RTSs by removing some technical restrictions. Both parametric bisimulations and RTSs compose transitively, like our structured simulations but unlike Kripke logical relations.

Although the context of their work is different, some of the techniques used by Benton, Dreyer, Hur, and their collaborators draw interesting parallels in our own work. Our "us vs. them" protocol is at least superficially similar to the "local vs. global knowledge" distinction that’s made in RTSs. One difference is, we distinguish between local and external invariants on the state shared by modules,
Compositional CompCert 12

8. Conclusion

CompCert is one of the great successes of formal methods for software verification. But as the authors of CompCert put it: “[CompCert’s] . . . formal guarantees of semantic preservation apply only to whole programs that have been compiled as a whole by [the] CompCert C [compiler].” 1 Our work overcomes this restriction.

References