# Compositional CompCert

Gordon Stewart Lennart Beringer Santiago Cuellar Andrew W. Appel
Princeton University
{jsseven,eberinge,scuellar,appel}@princeton.edu

## 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.

***Categories and Subject Descriptors*** F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

***General Terms*** Verification

***Keywords*** CompCert, Compiler Correctness

## 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 source language like 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.

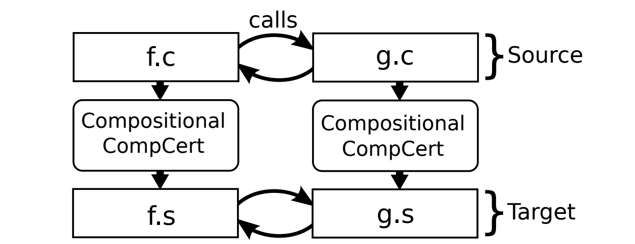



**Figure 1.** Compositional CompCert

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 (Leroy 2009) 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 multi-module 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 of (Beringer et al. 2014), 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 (Beringer et al. 2014)), 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


*POPL '15*, January 15 - 17 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676985

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 (relies) 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 *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 equvalence, 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, on GitHub.[1] 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 $\mathcal{L}$ for composing interaction semantics, independent of the language level of individual modules. Our statements of separate compilation and contextual equivalence (Sections 3 and 6) 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 $S_1, \cdots, S_n$, 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 6 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.* (Liang et al. 2012) 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 reorganized 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 $B$.

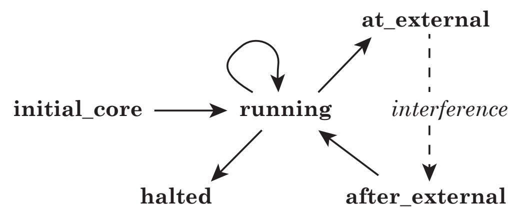
A key insight here is that the invariants that apply to distinct regions of memory—such as the regions reserved by the compiler for $A.f$'s and $B.g$'s spilled locals—are *subjective*: function $A.f$ can write to its own spilled locals but not to $B.g$'s, and vice versa for $B.g$ with respect to $A.f$'s 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 Nanevski's subjective concurrent separation logic (SCSL) (Ley-Wild and Nanevski 2013), 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 $\overrightarrow{v}$, we require that $A.f$'s compilation invariant "give up exclusive control" of all the memory regions reachable from $\overrightarrow{v}$ (*i.e.*, following pointer chains rooted in $\overrightarrow{v}$). This condition represents the guarantee that, while later compilation stages of $A.f$ can still reorganize parts of the state reachable from $\overrightarrow{v}$ (*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.

[1] `https://github.com/PrincetonUniversity/compcomp`.

Interestingly, this leakage protocol bears much in common with the *system-level semantics* of Ghica and Tzevelekos (Ghica and Tzevelekos 2012). 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 3. 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*, an operational model of shared-memory module and thread interaction. Section 3 employs interaction semantics to define the operational semantics of multilanguage linked programs, and semantic contextual equivalence. Then, we introduce structured simulations in Section 4. Section 5 presents the main theoretical results: vertical and horizontal compositionality, and contextual equivalence for CompCert. Section 6 describes the Compositional CompCert compiler itself, with a discussion of the effort required to port CompCert's existing languages and proofs to structured simulations.




$$\mathsf{Semantics}\ (G\ C\ M : \mathsf{Type}) : \mathsf{Type} \triangleq$$
$$\begin{cases} \mathsf{initial\_core} & : \ G \to \mathcal{V} \to \mathsf{list}\ \mathcal{V} \to \mathsf{option}\ C \\ \mathsf{at\_external} & : \ C \to \mathsf{option}\ (\mathcal{F} \times \mathsf{list}\ \mathcal{V}) \\ \mathsf{after\_external} & : \ \mathsf{option}\ \mathcal{V} \to C \to \mathsf{option}\ C \\ \mathsf{halted} & : \ C \to \mathsf{option}\ \mathcal{V} \\ \mathsf{corestep} & : \ G \to C \to M \to C \to M \to \mathsf{Prop} \end{cases}$$

**Figure 2.** Interaction semantics interface. The types $G$ (global environment), $C$ (core state), and $M$ (memory) are parameters to the interface. $\mathcal{F}$ is the type of external function identifiers. $\mathcal{V}$ is the type of CompCert values.

## 2. Interaction Semantics

*Interaction semantics* (called *core semantics* in (Beringer et al. 2014)) 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 occuring exclusively at external function call points, that is, via calls to functions declared in one module but defined in another. This gives a compiler or weakly consistent memory model the freedom to optimize between interaction points.[2]

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* (Appel et al. 2014), $M$ is instantiated to a step-indexed model of state used to model function pointer specifications and other higher-order features.) $\mathcal{F}$ is the type of external function identifiers. $\mathcal{V}$ is the type of values. $\mathcal{V}$ is usually just CompCert's value type.

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 initial_core $ge\ v\ \overrightarrow{v}$. The $v$ is a value, typically a function pointer, while $\overrightarrow{v}$ 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 $ge$. at_external interrogates core state $c$ to determine whether $c$ is "blocked" at an external function call interaction point. When at_external succeeds, it does so with the name of the external function being called (of type $\mathcal{F}$) 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 $c$ 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 $ge \vdash c, m \longmapsto c', m'$ 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.

[2] Our model is designed to support shared-memory concurrency, although this paper does not yet do 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 (Appel et al. 2014, Ch. 44). Racy programs can be accommodated as long as racy loads/stores can be modelled via permission changes (future work).

*Statements*

$$
\begin{array}{rcll}
s & ::= & \mathsf{Sskip} & \text{no-op} \\
 & | & \mathsf{Sassign}\ a_1\ a_2 & \text{lval} \leftarrow \text{rval} \\
 & | & \mathsf{Sset}\ id\ a & \text{temp} \leftarrow \text{rval} \\
 & | & \mathsf{Scall}\ optid\ a\ \overrightarrow{a} & \text{function call} \\
 & | & \mathsf{Sbuiltin}\ optid\ f\ \overrightarrow{\tau}\ \overrightarrow{a} & \text{intrinsic} \\
 & | & \mathsf{Ssequence}\ s_1\ s_2 & \text{sequence} \\
 & | & \mathsf{Sifthenelse}\ a\ s_1\ s_2 & \text{conditional} \\
 & | & \mathsf{Sloop}\ s_1\ s_2 & \text{infinite loop} \\
 & | & \mathsf{Sbreak} \mid \mathsf{Sreturn}\ a_{opt} & \text{break/return} \\
 & | & \mathsf{Scontinue}\ s & \text{continue statement} \\
 & | & \mathsf{Switch}\ s \mid \mathsf{Slabel}\ l\ s \mid \mathsf{Sgoto}\ l &
\end{array}
$$

*Internal & External Functions*

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{int} \mid \mathsf{long} \mid \mathsf{ptr}\ \tau \mid \cdots & \text{C types} \\
\gamma & ::= & \cdot \mid (id, \tau), \gamma & \text{typing environments} \\
f_i & ::= & \left\{ \begin{array}{ll} \tau & \text{function return type} \\ \gamma & \text{function parameter typing} \\ \gamma_v & \text{local variable typing} \\ \gamma_t & \text{temporary variable typing} \\ s & \text{function body} \end{array} \right. & \\
f & ::= & \mathsf{Internal}\ f_i \mid \mathsf{External}\ id_f\ \overrightarrow{\tau}\ \tau &
\end{array}
$$

*Continuations*

$$
\begin{array}{rcll}
\kappa & ::= & \mathsf{Kstop} & \text{safe termination} \\
 & | & \mathsf{Kseq}\ s\ \kappa & \text{sequential composition} \\
 & | & \mathsf{Kloop}\ s_1\ s_2\ \kappa & \text{loop continuation} \\
 & | & \mathsf{Kswitch}\ \kappa & \text{catch switch break} \\
 & | & \mathsf{Kcall}\ optid\ f_i\ \rho_v\ \rho_t\ \kappa & \text{catch function return}
\end{array}
$$

*Core States*

$$
\begin{array}{rcll}
\rho_v & ::= & \cdot \mid (id, (loc \times \tau)), \rho_v & \text{addressed var. environment} \\
\rho_t & ::= & \cdot \mid (id, v), \rho_t & \text{temporaries environment} \\
c & ::= & \mathsf{State}_{\mathsf{CL}}\ f_i\ s\ \kappa\ \rho_v\ \rho_t & \\
 & | & \mathsf{CallState}\ f\ \overrightarrow{v}\ \kappa \mid \mathsf{ReturnState}\ v\ \kappa &
\end{array}
$$

**Figure 3.** Syntax and semantics of Clight (excerpts). Continuations and core states appear only in the operational semantics.

*Registers*

$$
\begin{array}{rcll}
r_i & ::= & \mathsf{EAX} \mid \mathsf{EBX} \mid \mathsf{ECX} \mid \mathsf{EDX} & \text{integer registers} \\
 & | & \mathsf{ESI} \mid \mathsf{EDI} \mid \mathsf{EBP} \mid \mathsf{ESP} & \\
r_f & ::= & \mathsf{XMM0} \mid \cdots \mid \mathsf{XMM7} & \text{floating-point registers} \\
cr_{state} & ::= & \mathsf{ZF} \mid \mathsf{CF} \mid \mathsf{PF} \mid \mathsf{SF} \mid \mathsf{OF} & \text{control register state} \\
r & ::= & \mathsf{PC} \mid \mathsf{IR}\ r_i \mid \mathsf{FR}\ r_f \mid \mathsf{ST0} \mid \mathsf{CR}\ cr_{state} \mid \mathsf{RA} & \\
rs & ::= & \cdot \mid (r, v), rs & \text{register environments}
\end{array}
$$

*Instructions*

$$
\begin{array}{rcll}
p & ::= & \mathsf{MOV}_{\mathsf{RR}}\ r_i\ r_i \mid \mathsf{MOV}_{\mathsf{RI}}\ r_i\ i \mid \cdots & \text{moves} \\
 & | & \mathsf{JMP}_{\mathsf{L}}\ l \mid \mathsf{JMP}_{\mathsf{S}}\ id \mid \mathsf{JMP}_{\mathsf{c}}\ cond\ l \mid \cdots & \text{jumps} \\
 & | & \mathsf{CALL}_{\mathsf{S}}\ id \mid \mathsf{CALL}_{\mathsf{R}}\ r_i \mid \mathsf{RET} \mid \cdots & \text{calls/return} \\
 & | & \cdots\ \text{moves with conversion, integer arithmetic, } \textit{etc.} &
\end{array}
$$

*Core States*

$$
\begin{array}{rcll}
d & ::= & \mathsf{State}_{\mathsf{ASM}}\ \tau_0\ rs & \text{normal states} \\
 & | & \mathsf{MarshallState}_{\mathsf{In}}\ id_f\ \overrightarrow{v} & \text{marshall args. in} \\
 & | & \mathsf{MarshallState}_{\mathsf{Out}}\ (id_f, \overrightarrow{\tau_0}, \tau_0)\ \overrightarrow{v}\ rs & \text{marshall args. out}
\end{array}
$$

**Figure 4.** Syntax and semantics of CompCert x86 assembly (excerpts). Core states appear only in the operational semantics. Intfloatness types $\tau_0$ (used for value decoding) are int, float, long, or single.

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 $\mathsf{State}_{\mathsf{CL}}\ f_i\ s\ k\ \rho_v\ \rho_t$ 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 $f_i$, the function body $s$, the control continuation $\kappa$, and two environments, $\rho_v$ for mapping address-taken stack variables to their locations in memory, and $\rho_t$ for mapping temporary variables to their values. $\mathsf{CallState}\ f\ \overrightarrow{v}\ \kappa$ models Clight programs that are about to call function $f$ (either internal or external) with arguments $\overrightarrow{v}$ and continuation $\kappa$. $\mathsf{ReturnState}\ v\ \kappa$ 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:

$$
\begin{array}{l}
\mathsf{cl\_after\_external}\ v_{opt}\ c\ \triangleq \\
\quad \textbf{case}\ c\ \textbf{of}\ \mathsf{CallState}\ f\ \overrightarrow{v}\ \kappa\ \rightarrow \\
\qquad\quad \textbf{case}\ f\ \textbf{of}\ \mathsf{Internal}\ \_\ \rightarrow\ \mathsf{None} \\
\qquad\qquad \mid \mathsf{External}\ id_f\ \overrightarrow{\tau}\ \tau\ \rightarrow \\
\qquad\qquad\quad \textbf{case}\ v_{opt}\ \textbf{of} \\
\qquad\qquad\qquad \mid \mathsf{None}\ \rightarrow\ \mathsf{Some}\ (\mathsf{ReturnState}\ v_{undef}\ \kappa) \\
\qquad\qquad\qquad \mid \mathsf{Some}\ v\ \rightarrow\ \mathsf{Some}\ (\mathsf{ReturnState}\ v\ \kappa) \\
\qquad \mid \_\ \rightarrow\ \mathsf{None}
\end{array}
$$

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 $v_{undef}$ (whenever $v_{opt}$ was None) and $v$ (whenever $v_{opt}$ was Some $v$). In all other cases, we just return None.

The definition of initial_core $ge\ v\ \overrightarrow{v}$ 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 $f_i$, check that the arguments $\overrightarrow{v}$ are defined and match $f_i$'s type signature, then introduce state CallState (Internal $f_i$) $\overrightarrow{v}$ Kstop, which immediately steps to the body of function $f_i$ with the initial local variable environment $\rho_v$ that maps the function's formal parameters to its arguments $\overrightarrow{v}$. 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 assembly 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 $id_f$ with arguments $\overrightarrow{v}$, we produce

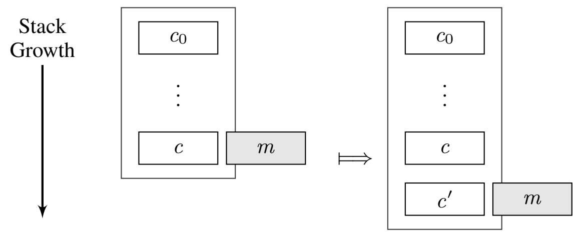



**Figure 5.** CALL case of the linking corestep relation. The outer boxes are "stacks-of-cores." Core state $c$ is at_external calling function $id_f$. Assuming $l$.plt $id_f$ $=$ Some $idx$ and $ge\,id_f$ $=$ Some $b_f$, initializing module $idx$ with function pointer Vptr $b_f$ results in initial core $c'$ being pushed onto the stack.

state MarshallState$_{\mathsf{In}}$ $id_f$ $\overrightarrow{v}$, 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 $\overrightarrow{v}$, in right-to-left `cdecl` order as expected by CompCert and `gcc`. (MarshallState$_{\mathsf{Out}}$ 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 $\mathcal{L}([\![S_0]\!], [\![S_1]\!], \ldots, [\![S_{N-1}]\!])$ over interaction semantics that defines the linked behavior of a *set* of interacting modules, as given by a multimodule program $P = S_0, S_1, \cdots, S_{N-1}$.

As input, $\mathcal{L}$ 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 $\mathcal{L}$ is a new interaction semantics $[\![P]\!] = \mathcal{L}([\![S_0]\!], [\![S_1]\!], \ldots, [\![S_{N-1}]\!])$ 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 initial_core); returning from such a function pops the top core from the stack and injects the return value into the state of the caller, using after_external.

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

$$\mathsf{Modsem} \triangleq \left\{ \begin{array}{lll} \mathsf{F},\ \mathsf{V},\ \mathsf{C} & : & \mathsf{Type} \\ \mathsf{ge} & : & \mathsf{Genv\ F\ V} \\ \mathsf{sem} & : & \mathsf{Semantics\ (Genv\ F\ V)\ C\ mem} \end{array} \right.$$

In this dependently typed record, the types of ge and 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 unit); 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 ge that was statically initialized for this module. It maps addresses to global variables and function-bodies.[3]

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

The output of $\mathcal{L}$ is an interaction semantics in the LinkedState "language." LinkedState is parameterized by $modules$, 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.

$$\mathsf{Core}\ (N : \mathsf{pos})\ (modules : I_N \to \mathsf{Modsem}) \triangleq \left\{ \begin{array}{lll} \mathsf{idx} & : & I_N \\ \mathsf{core} & : & \mathsf{C}\ (modules\ \mathsf{idx}) \end{array} \right.$$

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:

$$\mathsf{LinkedState}\ (N : \mathsf{pos})\ (modules : I_N \to \mathsf{Modsem}) \triangleq \left\{ \begin{array}{lll} \mathsf{plt} & : & \mathsf{ident} \to \mathsf{option}\ I_N \\ \mathsf{stack} & : & \mathsf{Stack}\ (\mathsf{Core}\ N\ modules) \end{array} \right.$$

The two fields of LinkedState are: the procedure linkage table 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 plt as a field in the LinkedState record, as opposed to deriving it from $N$ and $modules$, to retain flexibility to do dynamic linking in the future. The stack is always nonempty; all cores except the topmost one are at_external ($\forall c \in$ (pop stack). at_external $c =$ Some $-$).

Figure 6 gives the step relation. There are three rules. The STEP rule deals with the case in which the topmost core on the call stack ($c =$ peek $l$.stack) takes a normal internal step ($ge_c \vDash c, m \longmapsto c', m'$). $ge_c$ 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 $l$ **with** {stack := push $c'$ (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, CALL, handles the case in which the topmost core on the stack is at_external (at_external $c =$ Some $(id_f, \overrightarrow{v})$) making a cross-module function call. In this case, we use the initial_core function of the module semantics that defines function $id_f$ ($l$.plt $id_f$ $=$ Some $idx$) to initialize a new core state to handle the function call (initial_core $(modules\ idx)$ (Vptr $b_f$) $\overrightarrow{v}$ $=$ Some $c'$). The core $c'$ is then pushed onto the stack ($l$ **with** {stack := push $c'$ $l$.stack}) to become the new running core.

The RETURN rule models external function returns. Here, the core state $c$ is halted with return value $v$ ($l$ **with** {stack := push $c'$ $l$.stack}). To resume execution, we use the after_external function exposed by the caller's semantics $c' =$ peek (pop $l$.stack) to inject the return value $v$ (after_external (Some $v$) $c'$ $=$

[3] All the inputs to $\mathcal{L}$ must have ge functions that map exactly the same global addresses (modules that fail to declare some unused external global variables or functions can always be made to do so, by safety monotonicity).

$$\boxed{ge \vDash c, m \Longrightarrow c', m'}$$

$$\frac{c = \mathsf{peek}\ l.\mathsf{stack} \qquad ge_c \vDash c, m \longmapsto c', m'}{ge \vDash l, m \Longrightarrow l\ \textbf{with}\ \{\mathsf{stack} := \mathsf{push}\ c'\ (\mathsf{pop}\ l.\mathsf{stack})\}, m'}\ (\textsc{Step})$$

$$\frac{\begin{array}{c} c = \mathsf{peek}\ l.\mathsf{stack} \qquad \mathsf{at\_external}\ c = \mathsf{Some}\ (id_f, \overrightarrow{v}) \\ l.\mathsf{plt}\ id_f = \mathsf{Some}\ idx \qquad ge\ id_f = \mathsf{Some}\ b_f \\ \mathsf{initial\_core}\ (modules\ idx)\ (\mathsf{Vptr}\ b_f)\ \overrightarrow{v} = \mathsf{Some}\ c' \end{array}}{ge \vDash l, m \Longrightarrow l\ \textbf{with}\ \{\mathsf{stack} := \mathsf{push}\ c'\ l.\mathsf{stack}\}, m}\ (\textsc{Call})$$

$$\frac{\begin{array}{c} \mathsf{size}\ l.\mathsf{stack} > 1 \qquad c = \mathsf{peek}\ l.\mathsf{stack} \\ \mathsf{halted}\ c = \mathsf{Some}\ v \qquad c' = \mathsf{peek}\ (\mathsf{pop}\ l.\mathsf{stack}) \\ \mathsf{after\_external}\ (\mathsf{Some}\ v)\ c' = \mathsf{Some}\ c'' \end{array}}{ge \vDash l, m \Longrightarrow l\ \textbf{with}\ \{\mathsf{stack} := \mathsf{push}\ c''\ (\mathsf{pop}\ (\mathsf{pop}\ l.\mathsf{stack}))\}, m}\ (\textsc{Return})$$

**Figure 6.** Corestep relation of program linking semantics $\mathcal{L}$.

Some $c''$). State $c$ is then popped from the stack, and state $c'$ is updated to $c''$ ($l$ **with** $\{\mathsf{stack} := \mathsf{push}\ c''\ (\mathsf{pop}\ (\mathsf{pop}\ l.\mathsf{stack}))\}$).

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. Different modules may or may not share a "real" activation-record stack.

The final piece of linking semantics is the definition of the interface 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 none 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 $v_{opt}$ $c$ = Some $c'$). Finally, a linking semantics is halted when the stack contains a singleton halted core state (halted $c$ and size $l.\mathsf{stack} = 1$), *i.e.*, the topmost core is halted but has no return context.

***Contextual Equivalence.*** Linking semantics leads to a natural notion of semantic context: Take program contexts $C$ to be arbitrary module semantics Modsem. Then the application of a program context to an (open) multimodule program $P$ is just the semantics that results from linking the program with that context: $\mathcal{L}(C, [\![P]\!])$!

This notion of context-as-interaction-semantics is quite general: it supports the definition of program contexts in arbitrary languages, *e.g.*, Clight 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 includes one such example.)

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

**Definition 1** (Contextual Equivalence)**.**

$$P_S \sim P_T \triangleq \forall C.\, \mathcal{L}(C, [\![P_S]\!]) \Downarrow \iff \mathcal{L}(C, [\![P_T]\!]) \Downarrow$$

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 (Beringer et al. 2014), and then show our new enhancement, *structured simulations.*

LSRs established compiler correctness by showing that compilation preserved the protocol structure of interaction semantics, using CompCert's original match relations $\sim_f$ with memory injections $f$ to relate source and target states. For internal execution steps, they followed CompCert's forward simulation proofs. For external calls, they asserted that the two modules call the same function with related arguments, and that the simulation relation is reestablished at return points whenever the environment provides related return values, subject to a few constraints on how memory could evolve over the external calls.

The two most crucial of these constraints were that (1) in the source execution, external calls did not modify any memory region the compiler wished to remove;[4] and that (2) in the target execution, external calls did not modify locations that were unreachable from the source memory (an "unreachable" target location is one that does not correspond to a readable location in the source memory). Condition (2), in particular, enabled the proof of compiler phases such as spilling, which introduces new unreachable spill locations into a target program's stack frames. A deficiency of CompCert's simulation proofs and of LSRs was that they assumed conditions (1) and (2) at external calls, but did not prove that these properties were preserved by compilation.

Directly imposing constraints (1) and (2) onto the simulation clauses for internal steps does not work, however. A compiled function should be allowed to write to its *own* spill locations—just not to those of its caller.

To capture the difference in perspective between caller and callee, we make three adjustments to the LSR framework. First, to index the match relation $\sim$, we use *structured injections* $\mu$ instead of CompCert's original injections $f$. The additional structure in $\mu$ maintains the block-level ownership information necessary to tell a callee's (or other environmental) blocks apart from caller blocks. Second, we decorate the internal step relation of interaction semantics with *modification effects* $E$ such that locations not contained in $E$ are guaranteed not to be modified (*i.e.* written to, or freed) by the step in question. Third, we impose a *restriction* axiom onto $\sim$ that ensures compilation invariants depend only on memory regions either allocated by the module being compiled, or leaked to it via pointers returned from external calls. The details are as follows.

***Structured Injections.*** In CompCert, memory is allocated in regions, or *blocks*. Within each block, memory bytes are addressed using integer offsets (pointer arithmetic is allowed only within blocks). CompCert's memory injections $f$ : block $\rightarrow$

[4] For 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.

$$
\begin{array}{l}
\mathsf{Ownership} \triangleq \mathsf{Priv} \mid \mathsf{Pub} \mid \mathsf{Frgn} \mid \mathsf{Invis} \mid \mathsf{None} \\
\mu \in \mathsf{StructuredInjection} : \mathsf{Type} \triangleq \\
\quad \left\{ \begin{array}{l}
\mathsf{own}_S : \mathsf{block} \to \mathsf{Ownership} \\
\mathsf{own}_T : \mathsf{block} \to \mathsf{Ownership} \\
\mathsf{f_{us}} : \mathsf{block} \to \mathsf{option}\ \ (\mathsf{block} \times \mathcal{Z}) \\
\mathsf{f_{them}} : \mathsf{block} \to \mathsf{option}\ \ (\mathsf{block} \times \mathcal{Z})
\end{array} \right.
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{owned}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Priv}, \mathsf{Pub}\}\}, i \in \{S, T\} \\
\mathsf{shared}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Pub}, \mathsf{Frgn}\}\}, i \in \{S, T\} \\
\mathsf{vis}_i & \triangleq & \mathsf{owned}_i \cup \mathsf{shared}_i, i \in \{S, T\} \\
f\lfloor_X & \triangleq & \lambda b.\ \textbf{if}\ b \in X\ \textbf{then}\ f\ b\ \textbf{else}\ \mathsf{None} \\
\mu\lfloor_X & \triangleq & \mu\ \textbf{with}\ \{\mathsf{f_{us}} := \mathsf{f_{us}}\lfloor_X\}\{\mathsf{f_{them}} := \mathsf{f_{them}}\lfloor_X\}
\end{array}
$$

**Figure 7.** Structured injections.

option (block $\times$ $\mathcal{Z}$) relate source and target memories. For example, the memory injection that maps $b$ to Some $(b', \delta)$ associates source address $(b, 8)$ with target address $(b', 8 + \delta)$.

Structured injections $\mu$ (Figure 7) strengthen CompCert's memory injection relations with additional ownership structure. They have four components: Two *ownership* functions $\mathsf{own}_S, \mathsf{own}_T$ : block $\to$ 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: $\mathsf{f_{us}}$ and $\mathsf{f_{them}}$. $\mathsf{f_{us}}$ records the source–target mapping of blocks that were allocated by the current module; $\mathsf{f_{them}}$ 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 $\mathsf{own}_S(b)$ (resp. $\mathsf{own}_T(b)$) is either Pub or Priv. *External* blocks in source and target are those mapped by $\mathsf{own}_{\{S,T\}}$ 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 $\mathsf{vis}_S \triangleq \mathsf{owned}_S \cup \mathsf{shared}_S$ (and likewise for $\mathsf{vis}_T$). We use notation $\mathsf{foreign}_{\{S,T\}}$ and $\mathsf{public}_{\{S,T\}}$ to denote the 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) enforce that $\mathsf{f_{us}}$ and $\mathsf{f_{them}}$ (1) operate exclusively on blocks of appropriate ownership (*i.e.* $\mathsf{f_{us}}$ only maps owned blocks, to owned blocks, and similarly for $\mathsf{f_{them}}$ and external blocks); and (2) are total on their portion of shared blocks: $\mathsf{f_{us}}$ must map all $\mathsf{Pub}_S$ blocks, and must map them to $\mathsf{Pub}_T$ blocks, and similarly for $\mathsf{f_{them}}$ 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 injections 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 $\mathsf{vis}_S$ is always closed under pointer dereferencing and pointer arithmetic.

***Simulation Structures.*** Figure 8 presents the two core clauses of structured simulations $\preceq$, 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). In the Figure, $\sim_\mu$ is existentially quantified. There is no single definition of $\sim_\mu$, but instead, one per compilation phase—Figure 8 defines the laws that each such $\sim_\mu$ relation must satisfy.[5]

*Internal Steps*

$$
\begin{array}{l}
\langle c, m\rangle \sim_\mu \langle d, tm\rangle \ \wedge\ ge_S \vdash c, m \xrightarrow{E_S} c', m' \to \\
\exists d'\ tm'\ \mu'. \\
\quad (1)\ \mu \sqsubseteq_{\mathsf{us}} \mu' \ \wedge \\
\quad (2)\ \mathsf{separated}\ \mu\ \mu'\ m\ tm \ \wedge \\
\quad (3)\ \mathsf{locally\_allocated}\ \mu\ \mu'\ m\ tm\ m'\ tm' \ \wedge \\
\quad (4)\ \langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle \ \wedge \\
\quad (5)\ \exists E_T.\ ge_T \vdash d, tm \xrightarrow{E_T}{}^{+} d', tm' \ \wedge \\
\quad (6)\ E_S \subseteq \mathsf{vis}_S\ \mu \to \\
\qquad (6a)\ E_T \subseteq \mathsf{vis}_T\ \mu \ \wedge \\
\qquad (6b)\ \forall b_t\ z_t.\ (b_t, z_t) \in E_T \ \wedge\ b_t \notin \mathsf{owned}_T\ \mu \to \\
\qquad\qquad \exists b_s\ \delta.\ \mathsf{f_{them}}(b_s) = \mathsf{Some}\ (b_t, \delta) \ \wedge\ (b_s, z_t - \delta) \in E_S
\end{array}
$$

*External Steps*

*(at-external)*

$$
\left(\begin{array}{l}
\langle c, m\rangle \sim_\mu \langle d, tm\rangle \ \wedge \\
\mathsf{inject}\ \mu\ \overrightarrow{v_s}\ \overrightarrow{v_t} \ \wedge\ \mathsf{inject}\ \mu\ m\ tm \ \wedge \\
\mathsf{at\_external}\ c = \mathsf{Some}\ (id_f, \overrightarrow{v_s}) \ \wedge \\
\mathsf{at\_external}\ d = \mathsf{Some}\ (id_f, \overrightarrow{v_t}) \ \wedge \\
\nu \triangleq \textbf{leak\_out}\ \mu\ \overrightarrow{v_s}\ \overrightarrow{v_t}\ m\ tm
\end{array}\right) \to
$$

*(environment)*

$$
\begin{array}{l}
\forall \nu'\ v_s\ v_t\ m'\ tm'. \\
\left(\begin{array}{l}
\nu \sqsubseteq_{\mathsf{them}} \nu' \ \wedge \\
\mathsf{separated}\ \nu\ \nu'\ m\ tm \ \wedge\ \mathsf{valid}\ \nu'\ m'\ tm' \ \wedge \\
\mathsf{inject}\ \nu'\ v_s\ v_t \ \wedge\ \mathsf{inject}\ \nu'\ m'\ tm' \ \wedge \\
\mathsf{forward}\ m\ m' \ \wedge\ \mathsf{forward}\ tm\ tm' \ \wedge \\
\textbf{unchanged\_on}\ \{(b, z) \mid \mathsf{own}_S\ \nu\ b = \mathsf{Priv}\}\ m\ m' \ \wedge \\
\textbf{unchanged\_on}\ (\textbf{local\_out\_of\_reach}\ \nu\ m)\ tm\ tm' \ \wedge \\
\mu' \triangleq \textbf{leak\_in}\ \nu'\ v_s\ v_t\ m'\ tm'
\end{array}\right) \to
\end{array}
$$

*(after-external)*

$$
\begin{array}{l}
\exists c'\ d'.\ \mathsf{after\_external}\ v_s\ c = \mathsf{Some}\ c' \\
\quad \wedge\ \mathsf{after\_external}\ v_t\ d = \mathsf{Some}\ d' \\
\quad \wedge\ \langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle
\end{array}
$$

**Figure 8.** Structured simulations, internal and external step cases.

The structure of the internal diagram (which is simplified in the figure to elide stuttering source steps) is familiar from traditional forward simulation proofs: Assume we are in matching initial states $\langle c, m\rangle \sim_\mu \langle d, tm\rangle$ and we take a source step $ge_S \vdash c, m \xrightarrow{E_S} c', m'$ with effect $E_S$. Then there exists a matching $d'$, $tm'$, and Kripke-extended structured injection $\mu'$ such that $ge_T \vdash d, tm \xrightarrow{E_T}{}^{+} d', tm'$ and $\langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle$.Clause (1) (Kripke extension, $\mu \sqsubseteq_{\mathsf{us}} \mu'$) 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 $\mathsf{vis}_T\ \mu$, assuming that $E_S \subseteq$

[5] In most Compositional CompCert phases, the $\preceq_\mu$ relation is equality up to the injection of memory regions, the addition, removal, and merging of certain memory regions, and invariants on private memory regions. As in original CompCert, we say that nonpointer values such as integers are related only if they are actually equal.

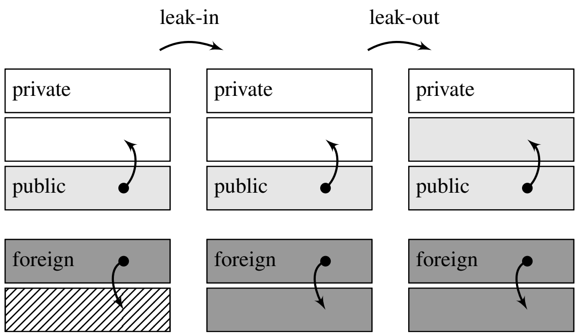



**Figure 9.** Graphical representation of the structured injection leakage operations. The thick black arrows are pointers in memory. The white private and light gray public boxes are owned ("us") blocks. The dark gray boxes are foreign ("them") blocks. The striped box is an Invis memory region that was allocated by another module but not yet leaked in. The leak-in operation marks the reachable invisible region as foreign. The leak-out operation marks as public a private region reachable from a public pointer.

$\mathsf{vis}_S\ \mu$. In other words, the compiler preserves the property of "writing to, and freeing, only visible locations."

Clause (6b) guarantees that writes to (and frees of) memory locations in the target that are not owned by $\mu$ ($b_t \notin \mathsf{owned}_T\ \mu$) can be "tracked back" to corresponding writes and frees in the source ($\exists b_s\, \delta.\, \mathsf{f}_{\mathsf{them}}(b_s) = \mathsf{Some}\ (b_t, \delta)$ and $(b_s, z_t - \delta) \in E_S$). 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, $ge_S \vdash c, m \overset{E_S}{\longmapsto} c', m'$ 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 intensionally 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 effects no longer "decompose": 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. Decomposition is used in the internal step case of the proof that structured simulations compose vertically.

We track only write and free effects, and not read effects, because compiler correctness invariants do not, in general, depend on which locations another module merely reads. Proving more general program refinements (*e.g.*, between multiple implementations of an ADT), or that the compiler does not introduce additional memory reads—a property useful in security contexts—would most likely require a generalization to read effects.

The external step diagram occupies the bottom half of Figure 8. It relates an $\mathsf{at\_external}$ source–target configuration pair $\langle c, m\rangle \sim_\mu \langle d, tm\rangle$ with the $\mathsf{after\_external}$ configuration pair $\langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle$ 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 $\nu'$ 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 $tm'$ ($\langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle$). The $\nu \sqsubseteq_{\mathsf{them}} \nu'$ is dual to the $\sqsubseteq_{\mathsf{us}}$ condition used in the internal step diagram. It says that $\nu'$ may 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.

*Reachability*

$$\begin{array}{l}
\mathsf{reach}\ :\ \mathsf{mem} \to \mathsf{set\ block} \to \mathsf{list\ block} \to \mathsf{set\ block}\\
\mathsf{reach}\ m\ R\ \mathsf{nil} \triangleq R\\
\mathsf{reach}\ m\ R\ ((b', z') :: L) \triangleq\\
\quad \{b \mid b' \in \mathsf{reach}\ m\ R\ L \wedge \mathsf{perm}\ m\ b'\ z' = \mathsf{Readable}\\
\qquad \wedge\, \exists z.\ m(b', z') = \mathsf{Vptr}\ (b, z)\ \}\\
\mathsf{REACH}\ m\ R \triangleq \{b \mid \exists L.\ b \in \mathsf{reach}\ m\ R\ L\}
\end{array}$$

*Export/Import*

$$\begin{array}{l}
\mathsf{export}_i\ \mu\ B\ :\ \mathsf{StructuredInjection} \triangleq \mu\ \mathbf{with}\ \{\mathsf{own}_i :=\\
\quad \lambda b.\, \mathbf{if}\ b \in B\ \mathbf{then}\ \mathsf{Pub}\ \mathbf{else}\ \mathsf{own}_i\ \mu\ b\}, i \in \{S, T\}\\
\mathsf{import}_i\ \mu\ B\ :\ \mathsf{StructuredInjection} \triangleq \mu\ \mathbf{with}\ \{\mathsf{own}_i :=\\
\quad \lambda b.\, \mathbf{if}\ b \in B\ \mathbf{then}\ \mathsf{Frgn}\ \mathbf{else}\ \mathsf{own}_i\ \mu\ b\}, i \in \{S, T\}
\end{array}$$

*Leakage*

$$\begin{array}{l}
\mathsf{leak_out}\ \mu\ \overrightarrow{v_s}\ \overrightarrow{v_t}\ m\ tm\ :\ \mathsf{StructuredInjection} \triangleq\\
\quad \mathbf{let}\ L_S \triangleq \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ \overrightarrow{v_S} \cup \mathsf{shared}_S\ \mu)\\
\qquad\qquad\quad \cap\ \mathsf{owned}_S\ \mu\\
\qquad\quad L_T \triangleq \mathsf{REACH}\ tm\ (\mathsf{blocksOf}\ \overrightarrow{v_T} \cup \mathsf{shared}_T\ \mu)\\
\qquad\qquad\quad \cap\ \mathsf{owned}_T\ \mu\\
\quad \mathbf{in}\ \mathsf{export}_T\ (\mathsf{export}_S\ \mu\ L_S)\ L_T\\
\mathsf{leak_in}\ \mu\ v_s\ v_t\ m\ tm\ :\ \mathsf{StructuredInjection} \triangleq\\
\quad \mathbf{let}\ L_S \triangleq \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ [v_s] \cup \mathsf{shared}_S\ \mu)\\
\qquad\qquad\quad \cap\ \overline{\mathsf{owned}_S\ \mu}\\
\qquad\quad L_T \triangleq \mathsf{REACH}\ tm\ (\mathsf{blocksOf}\ [v_T] \cup \mathsf{shared}_T\ \mu)\\
\qquad\qquad\quad \cap\ \overline{\mathsf{owned}_T\ \mu}\\
\quad \mathbf{in}\ \mathsf{import}_T\ (\mathsf{import}_S\ \mu\ L_S)\ L_T
\end{array}$$

**Figure 10.** Reachability and leakage.

The conditions listed in bold together compose the **structured simulation rely**. The predicate $\mathsf{unchanged\_on}\ U\ m\ m'$ specifies that memories $m$ and $m'$ are equal (same contents and permissions) at the locations in set $U$. In the source execution, we use $\mathsf{unchanged\_on}\ \{(b, z) \mid \mathsf{own}_S\, \nu\, b = \mathsf{Priv}\}\ m\ m'$ 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 $\mathsf{unchanged\_on}\ (\mathsf{local\_out\_of\_reach}\ \nu\ m)\ tm\ tm'$ 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 $\mathsf{unchanged\_on}$ here, we stipulate the nonmodification conditions of the rely extensionally.

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

The leak_in function used to define $\mu'$ at the end of the external step diagram plays a role analogous to that of leak_out, except that here, we are leaking into $\mu'$ new *foreign* blocks reachable from the return value $v_i$ 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 block set $B$ to Frgn as opposed to Pub.

***Restriction.*** The final consistency condition is: the simulation relation $\sim_\mu$ is independent of the Invis (and None) blocks. This condition is used, *e.g.*, in the proof of vertical composition (transitivity, Theorem 1). Technically, we enforce Invis-independence by requiring that $\sim_\mu$ be closed under restrictions to reach-closed supersets of the visible blocks, an operation defined in Figure 7 as $\mu\downharpoonright_X$ (with $X$ a block set). $\mu\downharpoonright_X$ denotes the structured injection obtained by restricting the maps $\mathsf{f}_{\mathsf{us}}$ and $\mathsf{f}_{\mathsf{them}}$ to the domain $X$. The closure condition says that if $\langle c, m\rangle \sim_\mu \langle d, tm\rangle$, then $\langle c, m\rangle \sim_{\mu\downharpoonright_X} \langle d, tm\rangle$ for any block set $X$ that contains at least $\mathsf{vis}_\mu$ 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 (Beringer et al. 2014). 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 $m_2'$ in the intermediate execution between source and target.

**Theorem 1** (Transitivity)**.** *Let $L_1$, $L_2$, and $L_3$ be effect-annotated interaction semantics. If $L_1 \preceq L_2$ is a structured simulation from $L_1$ to $L_2$ and $L_2 \preceq L_3$ a structured simulation from $L_2$ to $L_3$, then there exists a structured simulation $L_1 \preceq L_3$ from $L_1$ to $L_3$.*

***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) multimodule source and target programs. We give the theorem statement first, then explain some of the subtleties, in particular, the restriction to *reach-closed* source semantics, which enforces the single-program conditions corresponding to the structured simulation guarantees of Section 4, and to *valid* target semantics (a technical property related to the CompCert memory model, explained below).

**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 $[\![S_i]\!] \preceq [\![T_i]\!]$ for each source–target pair, and each of the $T_i$ is* ***valid****, then*
- *there is a simulation relation $\mathcal{L}([\![P_S]\!]) \leq \mathcal{L}([\![P_T]\!])$ 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. Whole program simulations are as in Figure 8, but without clauses (1-3) in the internal step diagram, without clauses for at_external and after_external, and without effects. As Corollary 1 will show, establishing $\leq$ is sufficient for proving contextual equivalence of open multimodule programs. A *valid* semantics is one that never stores *invalid* pointers into memory. Invalid pointers, in CompCert parlance, are those that refer to memory regions that have not yet been allocated (freed pointers are never invalid). All CompCert x86 programs are valid in this way. Validity is required, in the proof of Theorem 2, to maintain the invariant that the set of valid target blocks is reach-closed.

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

```
//Module A                      //Module B
void g(void);                     .globl _g
void h(int* x) {};              _g:
void f(void) {                    pushl %ebp
  int a=0;                        movl  %esp, %ebp
  if (a) {h(&a);}                 movl  42,(0x28ac1c)
  g();}                           popl  %ebp
void main(void){f();}             ret
```

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 memory 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 previously 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`, without 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 -O0` 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 depend only on translation-unit-local analyses. Correctness of optimizations 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 either from `g`'s initial arguments, from global variables, or from the return values of external calls `g` may have made previously.[6] This condition—no writes or frees to locations that are not "visible"—is the analogue of the $E_S \subseteq \mathsf{vis}_S\,\mu$ in clause (6) of Figure 8, but stated as a single-program property, independent of any particular structured injection $\mu$. We formalize the notion of a semantics that respects this characterization of visible locations as an extension of interaction semantics called *reach-closed semantics*, defined by the existence of an invariant $\mathcal{R}$ satisfying the laws in Figure 11. From the perspective of compiler correctness proofs, the

[6] 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.

*Reach-Closed Invariant*

$$\mathcal{R} : C \to \mathsf{mem} \to \mathsf{set\ block} \to \mathsf{Prop}$$

*Reach-Closed Initial Core*

$$\mathsf{initial\_core}\ ge\ v\ \overrightarrow{v} = \mathsf{Some}\ c\ \to \\ \forall m.\ \mathcal{R}\ c\ m\ (\mathsf{blocksOf}\ \overrightarrow{v})$$

*Reach-Closed Step*

$$\mathsf{roots}\ (ge : G)\ (B : \mathsf{set\ block})\ \triangleq\ \mathsf{globalBlocks}\, ge\ \cup\ B$$

$$\begin{aligned}&\mathcal{R}\ c\ m\ B\ \wedge\ ge \vdash c, m\ \overset{E}{\longmapsto}\ c', m'\ \to\\ &(1)\ E \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\, ge\ B) \wedge\\ &(2)\ \mathcal{R}\ c'\ m'\ (\mathsf{REACH}\ m'\ (\mathsf{freshblks}\ m\ m'\\ &\qquad\qquad \cup\ \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B)))\end{aligned}$$

*Reach-Closed After External*

$$\begin{aligned}&\mathcal{R}\ c\ m\ B \wedge\\ &\mathsf{at\_external}\ c = \mathsf{Some}\ (id_f, \overrightarrow{v}) \wedge\\ &\mathsf{after\_external}\ v_{opt}\, c = \mathsf{Some}\ c'\ \to\\ &\mathbf{let}\ B'\ \triangleq\ \mathbf{case}\ v_{opt}\ \mathbf{of}\\ &\qquad\qquad \mid \mathsf{None} \to B\\ &\qquad\qquad \mid \mathsf{Some}\ v \to \mathsf{blocksOf}\ (v :: \mathsf{nil})\ \cup\ B\\ &\mathbf{in}\ \mathcal{R}\ c'\ m'\ B'\end{aligned}$$

**Figure 11.** Reach-closed semantics maintain an additional internal invariant $\mathcal{R}$ on states $c$, memories $m$, and block sets $B$ that satisfies the laws above. The definitions are parameterized by types $G$ and $C$, by a global environment $ge : G$, and by an interaction semantics of type Semantics $G$ $C$ mem that defines step relation $\longmapsto$ and functions after_external and initial_core (at_external and halted are elided).

restriction to reach-closed contexts is what *enables* program transformations: It would be unsound, for example, to remove a dead memory allocation if the larger program context depended on it, as in the example program above.

The $\mathcal{R}$ invariant of reach-closed semantics quantifies over: Core states of the argument semantics $c : C$, the memory $m$ : mem, and a set $B$ that records the blocks exposed to the semantics at interaction points (via pointers in the initial argument list, in the return values of external function calls, and by local allocation). We use the notation $sem_{\mathsf{rc}}$, for *reach-closed semantics*, to denote a semantics $sem$ that exhibits such an $\mathcal{R}$.

The roots of a block set $B$ and global environment $ge$ are the union of $B$ and the global blocks of $ge$. The operative conditions of reach-closed semantics are those that characterize the reach-closed step relation (clauses 1 and 2). In particular, clause (1), which ensures that reach-closed semantics satisfy the structured simulation guarantees of Section 4, instruments the step relation of the underlying semantics with the additional condition that the effects $E$ produced by the step above clause (1) are a subset of the locations reachable in $m$ from the current roots.

Clause (2) asserts that the invariant can be reestablished after the step for: the blocks reachable (in $m'$) from newly allocated blocks (freshblks $m$ $m'$), if any, as well as from the blocks that were originally reachable in $m$ (REACH $m$ (roots $ge$ $c$)). This last condition ensures that the reachable set grows monotonically at each step, by not "forgetting" locations that were previously reachable.

The other interface laws modify $B$ as specified above. For example, the clause for after_external asserts that $\mathcal{R}$ can be reestablished for $B'$ equal to $B$ union the blocks exposed by the return values of external calls (blocksOf ($v$ :: nil)). initial_core asserts that the invariant can be established initially, with $B$ equal to the blocks exposed in the initial arguments. at_external and halted (not shown) assert that the arguments to external calls and return values, respectively, are not undefined.

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)**.**

$$\begin{aligned}&P_S \sim_{\mathsf{rc}} P_T\ \triangleq\\ &\quad \forall C_{\mathsf{rc}}.\ \ C \preceq C\ \wedge\ \mathsf{det}\ C\ \wedge\ \mathsf{valid}\ C\ \wedge\ \mathsf{safe}\,\mathcal{L}(C, [\![P_S]\!])\ \to\\ &\qquad\quad \mathcal{L}(C, [\![P_S]\!])\Downarrow\ \Longleftrightarrow\ \mathcal{L}(C, [\![P_T]\!])\Downarrow\end{aligned}$$

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

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

*for reach-closed source modules* $S_0, S_1, \cdots, S_{N-1}$ *and valid deterministic target modules* $T_0, T_1, \cdots, T_{N-1}$*. If for each* $i$*,* $[\![S_i]\!] \preceq [\![T_i]\!]$*, then* $P_S \sim_{\mathsf{rc}} P_T$*.*

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). $C$ must also be valid. 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 4 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).

Nor is the reach closure condition imposed above an unrealistic proof obligation. One can show, for example, that all Clight programs satisfy the restrictions imposed in Figure 11.

**Theorem 3** (Safe Clight Programs are Reach-Closed)**.** *There exists an* $\mathcal{R}$*, specialized to Clight states* $c$ *and the Clight step relation, that satisfies the laws given in Figure 11.*

The proof of this (perhaps counterintuitive) theorem relies on the fact that Clight programs never fabricate nonnull pointers, *e.g.*, by casting an integer to a pointer and then dereferencing it. (Even in standard C, casting an integer to a pointer, or vice versa, is only implementation defined, except when the pointer is `null`. See, *e.g.*, the C11 standard (C11 2011, 6.3.2.3).) □

The main difficulty in proving Theorem 2 (and Corollary 1) is in devising a simulation invariant to relate the stacks-of-cores runtime states of the linked programs $P_S$ and $P_T$. 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.

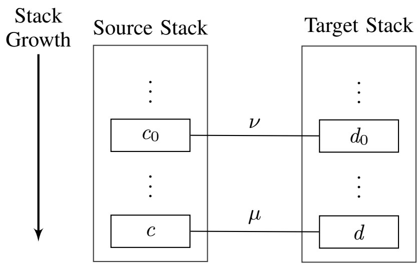



**Figure 12.** Schematic representation of the stacks-of-cores linking invariant. The inner white boxes are core states. Source core $c$ and target core $d$ are callees at the bottom of the $\mathsf{LinkedState}$ callstack, related by structured injection $\mu$ (memory is elided). Cores $c_0$ and $d_0$ are caller cores related by $\nu$.

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 $\nu$:

$$\mathsf{foreign}_S\,\mu \,\cap\, \mathsf{owned}_S\,\nu \subseteq \mathsf{public}_S\,\nu \qquad (1)$$

From the fact that source modules are reach-closed

$$E_S \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \qquad (2)$$

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 $\mathsf{vis}_S$ used in the simulation relations. We do this by maintaining the following two invariants:

$$\mathsf{roots}\ ge\ B \subseteq \mathsf{vis}_S\ \mu \qquad (3)$$

$$\mathsf{REACH}\ m\ (\mathsf{vis}_S\ \mu) \subseteq \mathsf{vis}_S\ \mu \qquad (4)$$

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, for incoming block set $\mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ (\mathsf{blocksOf}\ \overrightarrow{v}))$, 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 (2) and monotonicity of the REACH relation, imply that $E_S$ is a subset of $\mathsf{vis}_S\ \mu$. This fact, together with condition (1) above, is sufficient to prove the unchanged_on relies of Figure 8 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 includes three additional RTL-level optimizations (common subexpression elimination, constant propagation, and function inlining);[7] the adaptation of their proofs is ongoing work. The toplevel theorems we prove are the following.

[7] Inlining, which merges function stack frames, requires a slightly more general simulation relation than that of Figure 8 (without separated). We have done this generalization in a way that supports inlining and all previously proved phases but have not yet completed this proof in Coq.

**Theorem 4** (Compiler Correctness)**.** *Let* CompCert *denote the compilation function that composes the phases in Figure 13 in order. If* $\mathsf{CompCert}(S) = \mathsf{Some}\ T$*, for Clight module* $S$ *and x86 module* $T$*, then* $[\![S]\!] \preceq [\![T]\!]$.

*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 Clight modules such that* $\mathsf{CompCert}(S_i) = \mathsf{Some}\ T_i$ *for each* $i$*. Then* $S_0, S_1, \ldots, S_{N-1} \sim_{\mathsf{rc}} T_0, T_1, \ldots, T_{N-1}$.

*Proof.* By Corollary 1, Theorem 3, Theorem 4, and determinism and validity of CompCert x86 assembly. ☐

The process of making the proof of a transformation phase compositional typically proceeded as follows: We refined CompCert's internal match-state notion $\sim_f$ (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 passes 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: $\mathsf{vis}_\mu$ is closed under reachability, and the relation $\sim_\mu$ is closed under restriction to the visible set ($\mu\ \lfloor_{\mathsf{vis}_\mu}$). To ensure that global blocks were always mapped by each compiler phase, we treated them as Frgn to all modules. While the addition of these extra invariants proceeded in a mostly uniform manner across all phases, the refinement of $\sim_f$ to $\sim_\mu$ was phase-by-phase, due to the considerable internal differences between the various CompCert passes.

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 `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 inlineable, on the other. To sharpen this distinction, we modified the definition of the CminorSel 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. Much of this time was spent at the "boundaries" of the proof, updating the interfaces that connected our linking semantics and proofs to structured simulations. In general, the porting time decreased as 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, but also to the accumulation of a library of general-purpose lemmas that will remain useful as we continue to adapt the last few optimization passes.

As another measure of effort, we give lines-of-code for representative files in the development (Figure 14). Proofs of individual phases ("new") were on the order of 5klocs. By contrast, CompCert 2.1's ("old") proofs are about 2× smaller. The increase in proof lines is due mostly to the additional invariants we prove. However, we have not yet applied much proof automation at all, so we believe there is room for improvement. The increase in specification size is due to the use of duplicate language definitions: In order to add effects to the CompCert languages we duplicate the step relation of each semantics (once with, and once without, effects), then prove that the two semantics coincide. This results in specification counts that are larger than necessary.

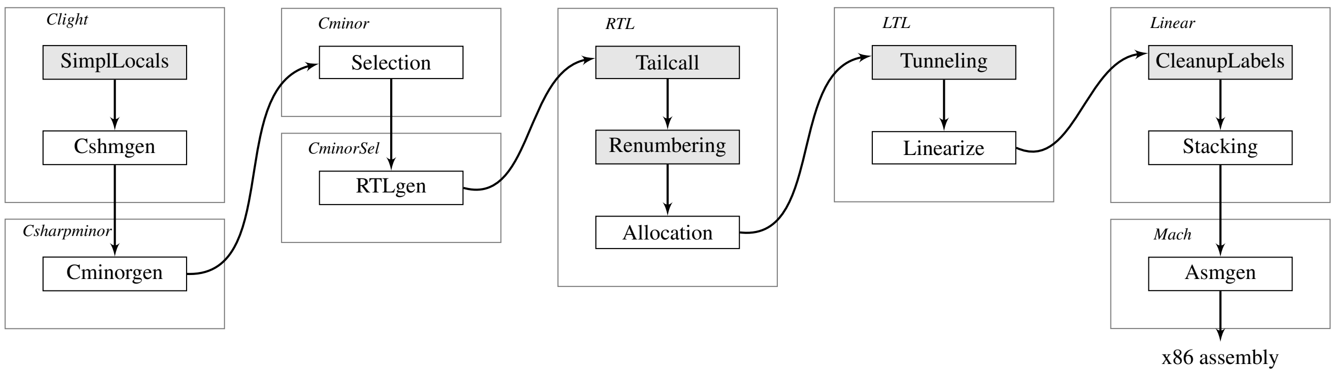



**Figure 13.** The phases of Compositional CompCert. Boxes in gray are optimization passes. Outer boxes indicate source languages.

| | Specs. | | Proofs | |
|---|---|---|---|---|
| *Compiler Phases:* | old | new | old | new |
| SimplLocals | 725 | 1004 | 2168 | 4572 |
| Cminorgen | 1619 | 1695 | 2796 | 5018 |
| RTLgen | 961 | 1355 | 1475 | 4883 |
| Tailcall | 441 | 637 | 628 | 1769 |
| Stacking | 712 | 1679 | 2906 | 6657 |
| *Theories:* | | | | |
| Structured Injs. (§4) | | 55 | | 2051 |
| Structured Sims. (§4) | | 763 | | 7967 |
| Transitivity (§5) | | 105 | | 5274 |
| Linking (§3, §5) | | 1651 | | 8459 |

**Figure 14.** Lines of code for selected parts of the development.

## 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 (McCarthy and Painter 1967). We cannot hope to give a complete survey here (see (Dave 2003)). Instead, we focus on the most closely related work.

***Verified Whole-Program Compilers.*** Moore (Moore 1989) was one of the first to mechanically verify a programming language implementation (a compiler for a language called Piton). The most well-known work in this vein since Moore is Leroy's CompCert C compiler in Coq (Leroy 2009), upon which Compositional CompCert is based. Chlipala has also built verified compilers in Coq—first, from lambda calculus to idealized assembly language (Chlipala 2007), and then, later, for an impure functional language (Chlipala 2010). 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 (Benton and Hur 2009), then for a functional language with polymorphism (Benton and Hur 2010). 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 biorthogonality). The refinements included extensions to step-indexed *Kripke* logical relations, for dealing with state in the context of more realistic ML-like languages (Hur and Dreyer 2011), and more recently, to relation transition systems (RTSs) (Hur et al. 2012) and the related parametric bisimulations (Hur et al. 2013). 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, whereas in RTSs the local vs. global distinction is really about different notions of term equivalence. Also, our "them" invariants—which encapsulate one structured simulation's view of the memory regions allocated by external funcions—are not quite "global" in the same sense as Hur *et al.*'s global knowledge. Perhaps more fruitfully, one can view interaction semantics—and the structured simulations that are "indexed to" interaction semantics—as an analogue of the type structure used to index standard logical relations, but here applied to imperative languages with impoverished type systems: C, x86, and the other languages of CompCert. As in Kripke logical relations, structured simulations use Kripke-style possible worlds to model, *e.g.*, memory allocation.

An alternative to language-independent interaction semantics is *multi-language semantics* (Ahmed and Blume 2011), which combines several languages of a compiler into a single host language via syntactic boundary casts in the syle of Matthews and Findler (Matthews and Findler 2007). This makes it possible to state the correctness of a separate compiler as contextual equivalence in the combined language, as Perconti and Ahmed have recently done for a two-phase compiler from System F with existential and recursive types (Perconti and Ahmed 2014). But where Perconti and Ahmed define contexts syntactically, as one-hole terms in the combined language, we define contexts semantically, as interaction semantics. McKay's variation of Perconti and Ahmed's approach replaces explicit boundary conversion with programmatic conversion expressed as terms of the combined language, but considers only a single transformation, closure conversion (McKay 2014). Recently, Wang *et al.* (**?**) built a compositional compiler from a restricted C-like language, Cito, to Bedrock. In contrast to our work, compiler correctness in (**?**) is tied to the specifics of the Cito program logic.

***Concurrency.*** Liang *et al.*'s work (Liang et al. 2012) on verifying concurrent program transformations inspired our use of a rely-guarantee discipline, but the complexity of stack frame management, spilling, and block coalescing in CompCert made it difficult to apply their ideas directly in our setting. Ley-Wild and

Nanevski's SCSL (Ley-Wild and Nanevski 2013), which we mentioned in the introduction, used subjective rely-guarantee invariants on auxiliary state to verify coarse-grained concurrent programs, such as parallel increment. Later work by Nanevski *et al.* extended the techniques to support verification of fine-grained concurrent programs (Nanevski et al. 2014). These subjective invariants made their proofs robust to the thread structure of the environment. Our "us vs. them" invariants serve a similar purpose—to prevent module-local structured simulations from being sensitive to the exact composition of their environment (other modules).

Lochbihler verified a whole-program compiler for multithreaded Java (Lochbihler 2012). Sevcík *et al.* built CompCertTSO (Sevcík et al. 2013), which adapted CompCert's correctness proofs to x86-TSO in order to reason about compilation of racy C code. Mansky's PTRANS framework (Mansky 2014) models optimizations as rewrite operations on parallel control flow graphs, specified using temporal logic formulae. While all three of these projects are whole-program, there are some similarities with our work. For example, both CompCertTSO and PTRANS lift program refinements from individual threads to whole programs, as we do for interacting modules, under certain noninterference conditions on shared state. A difference from our work is that PTRANS and CompCertTSO state the noninterference conditions as whole-system invariants. Our horizontal composition results instead rely only on a module-local characterization of noninterference, in the form of reach-closed semantics. That said, it would be interesting to investigate whether the compositional compilation approach we advocate could be applied to compilation with weak memory models.

## 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]." (Leroy 2014) We overcome this restriction.

## Acknowledgments

We thank the POPL anonymous reviewers, the members of the Princeton and Yale PL groups, Amal Ahmed, Robert Dockins, Dan Ghica, Robbert Krebbers, Xavier Leroy, and David Pichardie for valuable feedback. We especially thank Tahina Ramananandro for numerous insightful conversations. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0293. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

## References

A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *ICFP'11: The 16th ACM SIGPLAN International Conference on Functional Programming*, 2011.

A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.

N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP'09: The 14th ACM SIGPLAN International Conference on Functional Programming*, 2009.

N. Benton and C.-K. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, 2010.

L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP'14: The 23rd European Symposium on Programming*, 2014.

C11. C11 draft standard. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`, April 2011.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI 2007: Programming Language Design and Implementation*, 2007.

A. Chlipala. A verified compiler for an impure functional language. In *POPL'10: The 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

M. A. Dave. Compiler verification: A bibliography. *SIGSOFT Software Engineeering Notes*, 28(6), 2003.

D. Ghica and N. Tzevelekos. A system-level game semantics. In *MFPS'12: The 28th Conference on the Mathematical Foundations of Programming Semantics*, 2012.

C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL'12: The 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2012.

C. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. Parametric bisimulations: A logical step forward. In *POPL'13: The 40th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2013.

C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL'11: The 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011.

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

X. Leroy. The CompCert C compiler website. `http://compcert.inria.fr/compcert-C.html`, 2014.

R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL'13: The 40th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2013.

H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL'12: The 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2012.

A. Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institut für Technologie, July 2012.

W. E. Mansky. *Specifying and Verifying Program Transformations with PTRANS*. PhD thesis, University of Illinois, 2014.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL'07: The 34th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2007.

J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 1, 1967.

M. McKay. Compiler correctness via contextual equivalence. Undergraduate thesis, Carnegie Mellon University, May 2014.

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

A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP'14: The 23rd European Symposium on Programming*, 2014.

J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP'14: The 23rd European Symposium on Programming*, 2014.

J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.