Verified Compilation for Shared-memory C

Lennart Beringer[1], Gordon Stewart[1], Robert Dockins[2], and Andrew W. Appel[1]

[1] Princeton University
[2] Portland State University

Abstract. We present a new architecture for specifying and proving optimizing compilers in the presence of shared-memory interactions such as buffer-based system calls, shared-memory concurrency, and separate compilation. The architecture, which is implemented in the context of CompCert, includes a novel interaction-oriented model for C-like languages, and a new proof technique, called *logical simulation relations*, for compositionally proving compiler correctness with respect to this interaction model. We apply our techniques to CompCert's primary memory-reorganizing compilation phase, Cminorgen. Our results are formalized in Coq, building on the recently released CompCert 2.0.

1 Introduction

Shared-memory cooperation—the coordinated use of memory by several static or dynamic execution units—occurs ubiquitously in systems software. *Sequential* applications exchange pointers across module boundaries; *concurrent* threads interact via memory synchronization and by communicating pointers to shared data; nearly all programs communicate via memory with libraries and make pointer-valued system calls. Correct compilers—that preserve program safety and functional specifications—must respect a program's effects on memory.

Yet optimizing compilers for system languages such as C routinely perform code transformations that *alter* memory behavior. They relocate or eliminate load and store operations, they coalesce allocation events (especially as local variables are formulated into a stack frame), and they delete and insert loads, stores, stack allocations, and stack frees. For example, consider the CompCert verified optimizing C compiler [Ler11]. To limit pointer aliasing and perform efficient register allocation, CompCert identifies in an early compiler phase all local variables whose addresses are not taken. These unaddressed variables are shifted from in-memory blocks in function stack frames to register-allocated compiler temporaries. The variables are thus "removed from memory" (though some may later be spilled back into memory after register allocation).

Optimizing transformations are important for generating efficient code, yet also complicate the compiler's specification as it relates to memory. Correctness of any phase that adjusts the memory layout must preserve the program's memory behavior. However, it is not clear what "preservation of memory behavior" means when the compiler may introduce or remove memory effects. The difficulties are even more acute when translation units may be *separately compiled*, since

To appear in ESOP'14: 23rd European Symposium on Programming, April 2014, Springer LNCS 8410.

a pointer passed as an argument between modules may need to be translated depending on the (intermediate) languages in which the modules are expressed.

To address these issues, we present a novel framework for specifying C-like languages—imperative languages with low-level memory models—and their translations. The framework consists of two major components.

First, we develop a new interaction model, *core semantics*, that describes communication between execution threads with pointer exchange. A thread, or *core*, might represent true concurrency, or a sequential call to an external function. Crucially, our model is *language-independent,* separating thread-local data such as the control stack and local environment from global data such as shared memory. The caller of an external function thus need not know in which language the invoked function is implemented—a necessary precondition even for the *specification* of separate compilation and linking.

Second, we introduce *logical simulation relations (LSRs)*, a notion of compiler correctness that supports the core semantics model. Critically important—and a major contribution of our work—is a proof that LSRs compose transitively. Transitivity is essential for compositional verification of multiphase compilers.

We develop our framework in the concrete setting of CompCert. CompCert is an ideal testbed for three reasons: (i) it is the only publicly available optimizing C compiler that is equipped with a formal specification and correctness proof; (ii) CompCert provides a uniform memory model across all intermediate languages, a prerequisite for meaningful pointer communication; and (iii) CompCert punts on shared-memory cooperation, by disallowing communication of pointers to dynamically allocated data. CompCert's transitivity proof is dependent on this restriction, but in consequence, even basic interactions with system calls such as Linux `read` and `write` cannot be validated. Our framework reformulates CompCert's correctness theorem to lift these restrictions.

Contributions and Outline

i. We provide a detailed analysis of the tensions that result when proving compiler correctness in the shared-memory setting (§2). Our analysis is conducted in the concrete setting of CompCert and its memory model.
ii. We present *core semantics*, a new execution model for C-like languages (§3). Core semantics capture the interactions between a running thread, or core, and its environment via calls to external functions. Unlike in current CompCert, our execution model enables pointer sharing at interaction points.
iii. We develop (§4) a language-independent notion of shared-memory compiler correctness, called *logical simulation relations (LSRs)*, that is compatible with all three classes of memory transformations employed by CompCert. Our model of compiler correctness is *transitively composable*, a result that is necessary for the verification of multiphase compilers.
iv. Our approach requires minimal changes to CompCert's existing machine-checked correctness proofs. We demonstrate a proof adaptation for the hardest case, Cminorgen (§6).

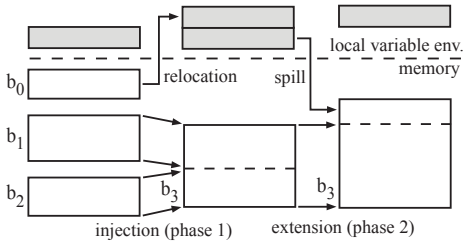


Fig. 1. CompCert block-level memory transformations

v. Our operational-semantic model supports the soundness proof of expressive program logics. For example, we have formalized the soundness proof of a step-indexed program logic for C light [A+14], but could support XCAP-like syntactic models [NS06] just as well. Section 5 briefly describes the connection of the C light logic to the techniques of this paper.

2 Technical Challenges and Approach

The technical challenges inherent in adapting CompCert to support shared memory lie in three major areas: the CompCert memory model, the CompCert correctness proofs, and the compiler's specification of external functions. This section motivates, with examples, the main technical challenges in each of these three areas, and outlines our solutions.

2.1 The CompCert C Memory Model

The semantics of pointer arithmetic, pointer comparison, and other "messy" features of C led Leroy, Blazy *et al.* to strike a balance in the design of CompCert's memory model [LB08,L+12] between concreteness and abstraction. Concreteness is necessary to model C's low-level memory behavior, such as aliasing and partial overlap of word-sized loads due to pointer arithmetic. Abstraction is needed for high-level reasoning. These competing requirements have led to a memory model—for use in operational semantics—that is elegant, yet inherently complex.

CompCert models memory as a set of *blocks*, the sizes of which are fixed at allocation time. Addresses are pairs of a block-number and an *offset*, which is an integer pointing to a particular location within the block. Pointer arithmetic is allowed within blocks but not across blocks. CompCert allocates a fresh block for each local variable, thus permitting pointer arithmetic *within* a local (array or struct) variable, but not across them. A stack-allocated `char` array of size n would be allocated an n-byte block, whereas a local integer variable occupies a block of size 4 (on 32-bit architectures).

This memory model is used in all operational semantics from C, through several intermediate languages, to assembly language. Because CompCert may alter a program's memory layout during compilation, the model must also support *memory transformations*. The transformations include (1) removal from function

activation records of scalar local variables that are never addressed with the & operator, and (2) spilling of local variables that could not be register-allocated. Figure 1 depicts these transformations schematically. In CompCert's variable relocation phase (labeled *injection* in the diagram), local variables that are never addressed (here, block 0) are moved from memory into a local variable environment. Additionally in this phase, CompCert coalesces the distinct memory blocks of the local variables for each function activation (here, blocks 1 and 2) into a single block representing the entire activation record (block 3). It is sound to merge blocks—thus permitting more pointer arithmetic in the target program than in the source program—because we assume that the source program did not go wrong: *i.e.*, did not do forbidden cross-variable pointer arithmetic.

To model phase (1), CompCert introduces a generic form of memory embedding called *memory injection*: a block-wise partial function of the form $\mathcal{B} \rightharpoonup \mathcal{B} \times \mathcal{Z}$. Here $\mathcal{B}$ is a countable set of blocks. The second component $\mathcal{Z}$ of the result pair is an integer offset that is applied, in the resulting memory, uniformly to every address in the mapped block. For example, assuming four byte blocks in the source language, the memory injection in phase (1) would be specified as, $b_0 \mapsto \mathsf{None}$ $b_1 \mapsto \mathsf{Some}\ (b_3, 4)$ $b_2 \mapsto \mathsf{Some}\ (b_3, 0)$. Block b_0 is unmapped because it is not addressed, block b_1 is mapped to block b_3 at offset 4 and block b_2 is mapped to block b_3 at offset 0. Thus a load from block b_1 at offset 0 becomes a load from block b_3 at offset 4 after the transformation.

Variables in the local variable environment do not have addresses, but may be spilled back into memory in phase (2) (labeled *extension* in Figure 1) after register allocation. Spilling requires that certain (stack-frame) blocks be *extended* (here, block 3) to make room for the spilled variables. Extension of a block is modeled by a change in *memory permissions*, which record the level of access allowed (read, write, etc.) at a particular memory location.

CompCert 1.x's[3] memory injections and memory extensions did not compose. This was not a problem for CompCert 1.x because memory was not exposed at external calls. We discuss the solution for CompCert 2.x in Section 4.1.

Permission Changes. A different limitation of CompCert 1.x is its operational model of memory-access permissions: at each abstract block number, a range $lo \ldots hi$ of addresses could be read or written. Calling a function would allocate a new (stack) block, returning would deallocate (without reusing block numbers).

Consider a source-level program logic for shared-memory concurrency, such as Concurrent Separation Logic [O'H07], in which we prove that a synchronization operation (lock acquire/release) adds or removes write or read permission to

[3] We describe as **CompCert 1.x** early versions of CompCert dating from 2006 in which Leroy *et al.* focused on whole-program single-threaded execution. Certain releases between 1.0 and 2.0 have incorporated several technical suggestions that resulted from the work reported in this paper and from discussions with Leroy. We describe as **CompCert 2.x** the 2.0 release, incorporating these changes, and near-future CompCert versions in which other adjustments to the specification may be made to improve compositionality of shared-memory interaction. Of course, between CompCert 1.0 and 2.0, Leroy *et al.* made many other unrelated enhancements.

some set of addresses. To communicate the *result* of the program-logic reasoning to CompCert, without embedding the entire program logic into CompCert, we now use a finer-grain permission structure in the operational semantics: byte by byte, read or write [L+12]. External functions (such as lock-acquire, lock-release) may change the permissions in arbitrary and nondeterministic ways.[4] Reasoning in the program logic ensures that the source program (with its operational permission changes) is safe. These permissions in the operational semantics will not exist physically when executing the compiled program.

CompCert 1.x could not permit this; it could not even permit general system calls such as `brk` to change memory permissions; `malloc` and `free` could not be modeled as system calls, so had to be special built-ins. The new permission model allows for expressive proofs about correct compilation of synchronized shared-memory programs.

2.2 The CompCert Correctness Proofs

The correctness proofs of the CompCert phases generally take the form of *forward simulations* to deterministic languages. By proving receptiveness of the source language, CompCert recovers event trace equivalence from the forward simulation proofs, for a limited class of events not containing pointers to stack or heap data. For shared-memory interaction, the events in CompCert 1.x's event traces are simply too inexpressive; but we *will* use forward simulations.[5]

A forward simulation proof consists of a *measured simulation relation* between the states of the source and target languages and a proof that the simulation relation can be re-established over execution steps. For example, in the proof of CompCert's variable relocation phase, the simulation relation asserts that the values of variables that have been removed from memory match the contents of the blocks from which the variables were relocated. That is, a four byte load from block b_0 in Figure 1 should produce the same value as evaluation of the associated local variable in the environment that results after the injection. "Measured" means the simulations allow multiple source language steps to correspond to zero target language steps, as long as there is a well-founded order on source language states that decreases at each step.

CompCert 1.x permits calls to *external functions* that take numeric parameters and do not access shared memory. For whole-program embedded appli-

[4] CompCert need not know which external functions do what; any external call may change memory permissions. Standard optimizing compilers such as `gcc` behave consistently with similar conservative assumptions. In principle, one could tell CompCert that lock-acquire only increases permissions, and lock-release only decreases them; then CompCert could hoist loads/stores past these calls (down, and up, respectively).

[5] Forward simulations are adequate because: We determinize multithreading with an external oracle (schedule), we transform racy loads/stores into "external calls" (which the compiler cannot remove or reschedule). Nonracy loads/stores are thread-modular by Dijkstra/Hoare (Pthreads) locking, modeled by (implicit virtual) permission transfers at acquire/release events; load/store without permission is "stuck" in the source program. In summary, no unmatched target behaviors are possible.

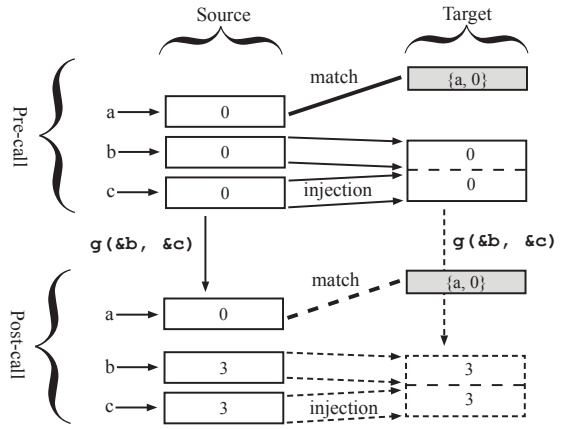


Fig. 2. Schematic CompCert simulation diagram for `f`'s call to external function `g`. Compilation goes left-to-right; execution goes top-to-bottom.

cations without an operating system, such an external function could drive an actuator or read a scalar value from a sensor. But more generally in the C language, external functions are just those that are declared within the current module (as `extern`) but defined, or implemented, in another.

```
//Module A
extern void g(int *d, int *e);

int f(void) {
  int a=0, b=0, c=0;
  g(&b, &c);
  return a + b + c;
}
```

```
//Module B

void g(int *d, int *e) {
  *d = 3;
  *e = 3;
}
```

For example, module `A` above declares an external function `g` taking two integer pointers as parameters and returning `void`. It also defines an *internal* function `f` calling `g` with the addresses of `f`'s local variables `b` and `c` as arguments. Module `B` defines `g` to perform side effects on the pointers that are passed.

Now imagine CompCert compiles module `A` through its variable relocation phase, with external module `B` remaining uncompiled. The memory state before and after the memory injection, just before external function `g` is called, is shown in the top half of Figure 2. The simulation invariant, pre-call, says that the value in memory at address `&a` in the source (*i.e.*, before variable relocation) equals the value of the variable `a` in the local variable environment in the target, after variable relocation. In the diagram, we depict this constraint as a bold line labeled "match" connecting the two boxes along the compilation axis.

To preserve semantics, CompCert needs to show that at the point of the external call, `g` will succeed in the injected memory state assuming it succeeded before memory injection, and that the "match" relation can be re-established after the call, assuming it held initially. This proof corresponds to the completion

of the lower half of the diagram in Figure 2. We must find a state that (1) results from running `g` with injected arguments; (2) is the injection of the lower left state; and (3) satisfies the match invariant with respect to the lower left. But an issue arises when we attempt to establish (3). To retain expressiveness, we must give external functions the freedom to mutate memory. On the other hand, simulation proofs for modules such as `A` should be able to safely assume that certain portions of memory remain unchanged by external calls. For example, the value in memory at address `&a` above, which is private to module `A` because the location has not been leaked as a parameter to `g`, should match the value assigned to `a` in the local variable environment both before and after the call to function `g`. How to reconcile these opposing concerns?

2.3 External Functions

CompCert 1.x maintains a distinction between "public" and "private" memory, but does so in a way that restricts the kinds of external functions that may be defined. To see why, consider CompCert 1.13's specification of external calls, modeled as a relation on the function arguments, the initial memory, the return value, and the final memory. We call this relation ec_sem, for external-call semantics. The axiom for external calls is: Suppose

- ec_sem $ge\ \overrightarrow{v_1}\ m_1\ rv_1\ m_1'$; inject $j\ m_1\ m_2$ (we use notation $m_1 \rightarrowtail_j m_2$ for this);
- block validity, and permissions (see below) are suitably preserved; we use notation forward $m_1\ m_1'$ for this; and
- val_list_inject $j\ \overrightarrow{v_1}\ \overrightarrow{v_2}$ (notation: $\overrightarrow{v_1} \rightarrowtail_j \overrightarrow{v_2}$);

that is, in the *source language,* in global environment ge, calling a particular function with parameters $\overrightarrow{v_1}$ and memory m_1 yields return-value rv_1 and memory m_1'; and there is a source-to-target memory injection j injecting $\overrightarrow{v_1}$ into $\overrightarrow{v_2}$, and m_1 into m_2. Then there must exist a post-call injection j' extending j (notation $j \leq j'$), and rv_2 and m_2' such that:

- ec_sem $ge\ \overrightarrow{v_2}\ m_2\ rv_2\ m_2'$; forward $m_2\ m_2'$; $rv_1 \rightarrowtail_{j'} rv_2$; $m_1' \rightarrowtail_{j'} m_2'$;
- unchOn (loc_unmapped j) $m_1\ m_1'$; and unchOn (loc_out_of_reach $j\ m_1$) $m_2\ m_2'$.

That is, at every external function call (ec_sem $ge\ \overrightarrow{v_1}\ m_1\ rv_1\ m_1'$) one can complete the simulation diagram (ec_sem $ge\ \overrightarrow{v_2}\ m_2\ rv_2\ m_2'$) for compiler phases that adjust the memory representation via a memory injection. (CompCert imposes a similar restriction in the memory extension case.) Also, on locations such as `&a` that are unmapped by the memory injection, the memory remains unmodified (unchOn, "unchanged on") in the pre-transformation execution. Memory locations in the post-transformation states that have empty permission initially, before the injection is applied (loc_out_of_reach $j\ m_1$), must remain unmodified by the external function call.

There are two problems with these restrictions. The first is that they impose a big-step semantics on external calls. Clause ec_sem $ge\ \overrightarrow{v_2}\ m_2\ rv_2\ m_2'$ requires that the external function terminate, in one step, when executed in m_2. This

requirement is incompatible with external functions implemented by potentially nonterminating code, or that might block in a concurrent setting.

The second problem lies with the restrictions on how external function calls may mutate memory (clauses beginning unchOn...). The unchOn $P\ m\ m'$ clauses have two effects. They ensure that (1) external calls do not modify, in the pre-compilation memories m_1 and m_1', locations which are unmapped by the memory injection j (loc_unmapped); and (2) they ensure that external calls do not modify locations in m_2 which were unreachable in m_1 under j (loc_out_of_reach). The problem here is that CompCert is using injections both to specify the memory transformations performed by the compiler, and to axiomatize the behavior of external function calls. In other words, restrictions on which locations external calls may mutate are keyed to the compiler transformations themselves. (Locations mapped by a memory injection are made public, whereas unmapped locations, *e.g.*, `&a` in the example above, remain private.) Unfortunately, this dual-purpose use of memory injections (and extensions) fails to account for situations in which the external function is *itself* code, perhaps a second CompCert translation unit, that is compiled independently from the calling module.

To illustrate the issues that arise when external functions are compiled, consider the case in which the function `g` of module `B` *itself* contains an unaddressed local variable, say `h`, which is relocated out of memory by CompCert's variable relocation phase.

```
void g(int *d, int *e) {      //Module B'
  int h = <expr>; *d = h; *e = <expr>;
}
```

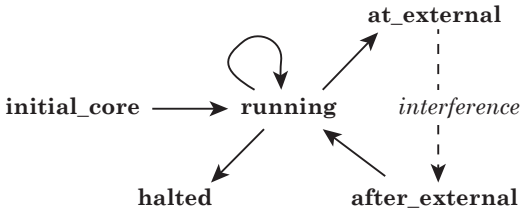
By the above axiom, module `B'` must not mutate memory blocks that are subsequently removed from memory by compilation (here, the block containing variable `h`). Yet this is exactly what module `B'` does when it assigns to `h`. Indeed, since `h` is a private local variable, the modification is perfectly acceptable behavior.

3 Core Semantics

The first major part of our solution is to define a uniform, protocol-oriented interface to languages that interact with their environments. Imagine a multithread shared-memory execution. One can spawn a new thread; a thread may yield (or block on a synchronization) and perhaps later resume; eventually a thread may exit. We use this model not only for concurrency but also for sequential calls to separately compiled functions (spawn a new "thread" to run the call, block until it returns) and for a single thread running in an operating-system context with system calls. When a thread yields (or calls a sequential external function), its local state including stack and registers will be preserved until it resumes, but the state of most of memory may have changed arbitrarily upon resumption.

Core semantics (Figure 3) are a general formulation of a thread protocol. At a high level, a core semantics (G, C, M) is a partitioning of a thread's state into a *global environment* (G), a *local* part (C), which we call the *core state*, or *core*,

and which typically includes both the control continuation and local variable environment, and a shared part (M), which we typically identify with shared memory. $\mathcal{V}$ is the type of values, and $\mathcal{F}$ is the type of external function names.



$$\begin{aligned}
\mathsf{initial_core} &: G \to \mathcal{V} \to \mathsf{list}\ \mathcal{V} \to \mathsf{option}\ C \\
\mathsf{at_external} &: C \to \mathsf{option}\ (\mathcal{F} \times \mathsf{list}\ \mathcal{V}) \\
\mathsf{after_external} &: \mathsf{option}\ \mathcal{V} \to C \to \mathsf{option}\ C \\
\mathsf{halted} &: C \to \mathsf{option}\ \mathcal{V} \\
\mathsf{corestep} &: G \to C \to M \to C \to M \to \mathbb{T}
\end{aligned}$$

Fig. 3. Core 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 values, and $\mathbb{T}$ is Coq's type of propositions, Prop. The names initial_core, at_external, after_external, halted are not constructors, but are (proved) disjoint predicates.

With this partitioning comes a step relation (corestep) on core states and memories that defines the small-step operational model of the core semantics. We will often write the corestep relation as $ge \vdash \langle c, m\rangle \longmapsto \langle c', m'\rangle$. The global environment ge maps functions to their definitions and does not vary.

In a (concurrently or sequentially) multithreaded system, different cores could have different core types (C) and different corestep relations. This permits interoperation of modules written in different languages. But such a surrounding system, modeling (respectively) a scheduler or a linker, is not needed for specifying compilation. This is an important separation of concerns.

To enforce the protocol described above, we divide core states into the five *lifetime* stages. *Initial* cores result directly from the creation of the thread or initialization of the program using initial_core. Typically, an initial core contains an empty local environment, together with a control continuation consisting of a single function call (the $\mathcal{V}$ parameter in the definition), with arguments (list $\mathcal{V}$). For a standalone program, this function is `main`; for a thread, it is the function that was forked; for a call to a separately compiled module, it is the called function. At_external cores are those initiating an external function call. In C terminology, external functions are just functions that are declared within the current translation unit or module but which are defined elsewhere (*e.g.*, in a module that is later linked to the current one). After_external cores result from resumption of the thread or program after an external call. In the transition from after_external to a running state, a core is expected to incorporate the return value (option $\mathcal{V}$) into its local variables (in its own language-dependent way). Halted cores are just that: threads or programs that have terminated normally, yielding an optional return value (option $\mathcal{V}$). Finally, *running* cores are neither blocked on an external function call nor halted.

Statements:

s ::= Sskip — no-op
| Sassign a_1 a_2 — lval ← rval
| Sset id a — temp ← rval
| Scall $optid$ a $\overrightarrow{a}$ — function call
| Sbuiltin $optid$ f $\overrightarrow{\tau}$ $\overrightarrow{a}$ — intrinsic
| Ssequence s_1 s_2 — sequence
| Sifthenelse a s_1 s_2 — conditional
| Sloop s_1 s_2 — infinite loop
| Sbreak | Sreturn a_{opt} — break/return
| Scontinue s — continue stmt.
| Switch s | Slabel l s | Sgoto l

Internal Functions:

F_i ::= { returnType τ; — fun. ret. type
params $\overrightarrow{(id, \tau)}$; — params./types
locals ρ_v; — local var. env.
temps ρ_t; — temp. env.
body s } — function body

Internal & External Fun. Definitions:

τ ::= int | float | long | single
F ::= Internal F_i
| External f $\overrightarrow{\tau}$ τ

Fig. 4. Syntax of C light

Preemption. The core semantics protocol is nonpreemptive—it does not *directly* model thread preemption that may result from interrupt handling. We can make this simplification—even though the underlying machine-language system may do preemption—because we write well-synchronized *race-free* code. Our source-level program logic verifies a stronger property than race-freedom: every memory access is performed with permission, and synchronization ensures that no two threads have conflicting permissions to a given address. Race-free programs running in a nonpreemptive semantics soundly approximate race-free programs in an interleaving semantics. Moreover, we are interested ultimately in proving correctness of the logic and compilation toolchain with respect to weak memory models—on such machines, *interleaving* is not even the right model. Race-free programs have sequentially consistent behavior on all well-behaved weakly consistent machines.[6] In our approach, the compiler correctness theorems can be oblivious of preemption and weak cache coherence—they just follow the rules of operational memory permissions.

3.1 Example: C light

As an example of a core semantics, we show CompCert C light. This high-level subset of C is the target of CompCert's first translation phase (from the full CompCert C language). It serves as a natural interface between CompCert, user-level program logics, and verified static analyses.

Figure 4 gives the syntax of C light. The syntax of expressions a is standard. In the statement syntax, `for` and `while` loops have already been translated (in an earlier compiler phase) to combinations of the more primitive Sloop and Sbreak constructs. The details of local control flow (loop, if, break, continue, switch, goto) are standard CompCert 1.13 Clight, and not relevant to (or changed by) our work on external interaction.

[6] This is not a theorem, it's better than a theorem: it is the *definition* of "well-behaved" for weakly consistent memory models.

$$\frac{\begin{array}{c} ge \vdash a \Downarrow_{\rho_v,\rho_t,m} v_f \qquad ge \vdash \overrightarrow{a} \Downarrow_{\rho_v,\rho_t,m} \overrightarrow{v} \qquad ge[v_f] = \mathsf{Some}\ (\mathsf{Internal}\ F_i) \\ \mathsf{typeOf}\ F_i = \mathsf{Tfunction}\ \overrightarrow{\tau}\ \tau \qquad \mathsf{allocVars}\ \rho_\emptyset\ m\ (\mathsf{locals}\ F_i) = (\rho'_v, m') \\ \mathsf{bindParams}\ (\mathsf{params}\ F_i)\ \overrightarrow{v}\ (\mathsf{initTempEnv}\ (\mathsf{temps}\ F_i)) = \mathsf{Some}\ \rho'_t \end{array}}{\begin{array}{l} ge \vdash \langle \mathsf{RunState}(\rho_v, \rho_t, \mathsf{Scall}\ optid\ a\ \overrightarrow{a} \cdot \kappa), m\rangle \longmapsto \\ \qquad \langle \mathsf{RunState}(\rho'_v, \rho'_t, \mathsf{body}\ F_i \cdot \mathsf{Sreturn}\ \mathsf{None} \cdot \mathsf{Kcall}\ optid\ F_i\ \rho_v\ \rho_t\ \kappa), m'\rangle \end{array}}\ (\textsc{ScallInternal})$$

$$\frac{ge \vdash a \Downarrow_{\rho_v,\rho_t,m} v_f \qquad ge \vdash \overrightarrow{a} \Downarrow_{\rho_v,\rho_t,m} \overrightarrow{v} \qquad ge[v_f] = \mathsf{Some}\ (\mathsf{External}\ f\ \overrightarrow{\tau}\ \tau)}{\begin{array}{l} ge \vdash \langle \mathsf{RunState}(\rho_v, \rho_t, \mathsf{Scall}\ optid\ a\ \overrightarrow{a} \cdot \kappa), m\rangle \longmapsto \\ \qquad \langle \mathsf{ExtCallState}(f, \overrightarrow{\tau}, \tau, \overrightarrow{v}, optid, \rho_v, \rho_t, \kappa), m\rangle \end{array}}\ (\textsc{ScallExternal})$$

Fig. 5. Internal and external call rules from the operational semantics of C light

Functions F are either internal (defined in the current translation unit) or external (declared here but defined elsewhere). Internal functions comprise a record containing the function return type, a list of function parameters with their types, a local variable environment for address-taken variables, a temporaries environment for the rest of the function variables, and the function body. External functions comprise an external function identifier f, a list of argument types $\overrightarrow{\tau}$ and a return type τ, where τ is int, float, long, or single (single-precision floats). External functions do not contain a function body. The core semantics for C light will stop at external calls and yield control to the execution environment.

Semantics. Figure 5 shows our reformulation of the internal and external function call rules of the C light operational semantics. The operational semantics is a three-place relation on global environments $ge : G$, initial configurations $\langle c, m\rangle$ and final configurations $\langle c', m'\rangle$. Here c is a core state; m is a CompCert memory. The relation $ge \vdash a \Downarrow_{\rho_v,\rho_t,m} v$ denotes big-step evaluation of expression a to value v in global environment ge, local variable environment ρ_v, temporaries environment ρ_t, and memory m.

We instantiate the type C of the core semantics interface to C light as follows.

$$\begin{array}{rll} c \in C ::= & \mathsf{RunState}\ \rho_v\ \rho_t\ \kappa & \text{“running” states} \\ \mid & \mathsf{ExtCallState}\ f\ sig\ \overrightarrow{v}\ optid\ \rho_v\ \rho_t\ \kappa & \mathsf{at\_external}\ \text{states} \end{array}$$

RunStates are normal execution states. ExtCallStates are calls to an external function f, with arguments $\overrightarrow{v}$. Parameter *optid* is an optional return value variable (= None when the function has `void` return type). The control continuation κ is a stack of suspended commands and function activations. In the ExtCallState constructor, *sig* is the external function type signature.

Next, we define the at_external function of the core semantics interface as a straightforward match on a core state c, returning Some $(f, \overrightarrow{v})$ when c is an ExtCallState, and None otherwise.

After_external takes as arguments an optional return value v_{ret} (again, None is used for `void` functions) and a core state c. If c is an ExtCallState and the return value is not None, then the temporary environment is updated to reflect the new return value. After_external will return None if c is not a proper external call state or if the return value and return variable are incompatible.

Readers familiar with CompCert 1.x will observe the proximity of our definition to Leroy *et al.*'s presentation: our adaptation removes the memory components from the two state constructors RunState and ExtCallState and adds the definitions of after_external and so on. The operational semantics (not shown) arises by refactoring the existing definition in accordance with these state representation changes and removing the rule for external function calls: such calls are now handled by the generic core-semantics interface.

4 Logical Simulation Relations

To adapt compiler correctness to the core semantics of Section 3 in a composable way, we take inspiration from the well established notion of type-indexed logical relations. Given a pair of core semantics, our notion of compiler correctness takes the form of a forward simulation, a correspondence relation between cores that is structure-preserving in source-to-target direction. Given the absence of sufficiently expressive type structure, *preservation of structure* in our case amounts to compatibility with the cores' lifetime stages.

CompCert distinguishes among three kinds of translations: *memory equality passes* leave memory unaffected but may modify the representation or operational behavior of cores; *memory extension passes* may enlarge existing memory blocks (by increasing the block size during allocation) and increase the definedness of memory-held values, but do not add or remove blocks; *memory injection passes* may discard or merge blocks by eliminating or coalescing allocation instructions. Our simulation relation accordingly defines distinct clauses for the three cases. Definition 1 below details the clause for injection passes, where $j, j', \ldots : \mathcal{B} \rightharpoonup \mathcal{B} \times Z$ indicate block relocations (*e.g.*, $j\ b = (b', z)$ relocates block b to a contiguous region in block b', starting at offset z), $j \leq j'$ indicates inclusion of relocations, $j \bowtie_{m_1;m_2} j'$ denotes that for any entry $j'\ b_1 = (b_2, z)$ not present in j, blocks $b_{\{1,2\}}$ must be unallocated in $m_{\{1,2\}}$, and $m_1 \rightarrowtail_j m_2$ indicates that m_2 is m_1's image under j (and similarly for $\overrightarrow{v_1} \rightarrowtail_j \overrightarrow{v_2}$).

Definition 1 (Measured Forward Simulation (Injection Case)). *Let M be the type of CompCert memories; $L_1 = (G_1, C_1, M)$ be the source core semantics; $L_2 = (G_2, C_2, M)$ be the target core semantics; $ge_1 : G_1$ be some source global environment; $ge_2 : G_2$ be some target global environment.*

Then we say there is a measured forward simulation for injections *from L_1 to L_2 (notation $L_1 \preceq_{\mathsf{Inj}} L_2$) if there exist a well-founded-order $<$ and a family of relations $(\sim_j) : C_1 \to M \to C_2 \to M \to \mathbb{T}$ on cores and memory states, indexed by memory injections, such that the following hold.*

1. *If* $\mathsf{initial\_core}\ ge_1\ u_1\ \overrightarrow{v_1} = \mathsf{Some}\ c_1$*;* $\mathsf{entryPoints}\ u_1\ u_2\ sig$*;* $m_1 \rightarrowtail_j m_2$*; and* $\overrightarrow{v_1} \rightarrowtail_j \overrightarrow{v_2}$ *then there exists* c_2 *such that* $\mathsf{initial\_core}\ ge_2\ u_2\ \overrightarrow{v_2} = \mathsf{Some}\ c_2$ *and* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$*.*
2. *If* $\mathsf{halted}\ c_1 = \mathsf{Some}\ v_1$ *and* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$ *then there exists* v_2 *such that* $\mathsf{halted}\ c_2 = \mathsf{Some}\ v_2$*,* $v_1 \rightarrowtail_j v_2$*, and* $m_1 \rightarrowtail_j m_2$*.*
3. *If* $ge_1 \vdash \langle c_1, m_1\rangle \longrightarrow \langle c_1', m_1'\rangle$ *then for all* c_2*,* j*,* m_2 *such that* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$*, there exist* c_2'*,* m_2'*,* j' *for which* $j \leq j'$*;* $j \bowtie_{m_1;m_2} j'$*; and* $\langle c_1', m_1'\rangle \sim_{j'} \langle c_2', m_2'\rangle$*; and either*
   - $ge_2 \vdash \langle c_2, m_2\rangle \longrightarrow^+ \langle c_2', m_2'\rangle$*; or*
   - $ge_2 \vdash \langle c_2, m_2\rangle \longrightarrow^* \langle c_2', m_2'\rangle$ *and* $c_1' < c_1$*.*
4. *If* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$ *and* $\mathsf{at\_external}\ c_1 = \mathsf{Some}\ (f, \overrightarrow{v_1})$ *then* $m_1 \rightarrowtail_j m_2$*, and there exists* $\overrightarrow{v_2}$ *with* $\overrightarrow{v_1} \rightarrowtail_j \overrightarrow{v_2}$ *and* $\mathsf{at\_external}\ c_2 = \mathsf{Some}\ (f, \overrightarrow{v_2})$*.*
5. *If* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$ *and* $\mathsf{at\_external}\ c_1 = \mathsf{Some}\ (f, \overrightarrow{v_1})$*, then for all* m_1'*,* m_2'*,* j'*,* v_1'*,* v_2' *with* $j \leq j'$*;* $j \bowtie_{m_1;m_2} j'$*;* $m_1' \rightarrowtail_{j'} m_2'$*;* $v_1' \rightarrowtail_{j'} v_2'$ *and*
   - $\mathsf{forward}\ m_1\ m_1'$*;* $\mathsf{forward}\ m_2\ m_2'$*;*
   - $\mathsf{unchOn}\ (\mathsf{loc\_unmapped}\ j)\ m_1\ m_1'$*;* $\mathsf{unchOn}\ (\mathsf{loc\_out\_of\_reach}\ j\ m_1)\ m_2\ m_2'$

   there exist c_1'*,* c_2' *such that*
   - $\mathsf{after\_external}\ (\mathsf{Some}\ v_1')\ c_1 = \mathsf{Some}\ c_1'$*;*
   - $\mathsf{after\_external}\ (\mathsf{Some}\ v_2')\ c_2 = \mathsf{Some}\ c_2'$*; and* $\langle c_1', m_1'\rangle \sim_{j'} \langle c_2', m_2'\rangle$*.*

The definition contains one clause for each protocol stage of core semantics:

Initial Cores. Clause 1, the base case, requires L_2 to match any L_1-initial core, given matching memories and arguments and related entry points.

Halted Cores. Symmetrically, clause 2 propagates termination from L_1 to L_2 for any $\sim_j$-related states, guaranteeing correspondence with respect to j for final memories and return values.

Core Steps. Clause 3 handles core steps, following the pattern of CompCert 1.x's forward simulations. An L_1 step may be matched by empty or nonempty sequences of L_2 core steps. In order to prevent infinite stuttering, the well-founded measure $<$ over core states c must decrease each time a possibly empty sequence is chosen. Since c_1 and c_2 may allocate new blocks during execution, resulting in larger memories m_1' and m_2', the relocation map j may also be extended to j' (notation $j \leq j'$) to account for the new blocks (under condition $j \bowtie_{m_1;m_2} j'$).

External Steps. The most interesting clauses concern the interaction of a core semantics with its environment. Clause 4 requires that L_2 match any call performed by L_1 with a call to the same function, with corresponding arguments.

Function Returns (Clause 5). In contrast to formulations using logical relations or $\top\top$-closure, we do not explicitly impose a simulation relation on environments. Instead, we require that the cores be ready to accept (and to re-establish the match relation on) nearly any pair of memories and return values the environments happen to return.[7] As a consequence, a compilation is considered correct *independent* of the termination behavior of its environment.

[7] This is an important point! The authors of a compiler such as `gcc` or CompCert make few assumptions about the environment, or about separately compiled modules. They do not want their reasoning about compiler correctness entangled with specifications of the programs to be linked with.

More precisely, given a call in L_1 (and, necessarily, a corresponding call in L_2, by Clause 4), we mandate that the match relation $\sim$ be re-established (and the resumption of normal execution succeed in both languages) whenever the environments yield back with return values v_1' and v_2' and updated memories m_1' and m_2' that are related by a relocation map j'. Here j' is an extension of the relocation map j provided at the time of the calls, meaning it agrees with j wherever j was defined, but may relocate new, freshly allocated blocks.

In accordance with the restrictions on external calls in CompCert, however, we assume that the evolution of memories across calls satisfies some basic conditions: $\mathsf{forward}\ m\ m'$ requires that an evolution $m \rightsquigarrow m'$ does not invalidate (*i.e.*, return to the allocation pool) any block that was previously allocated, and at most decreases the maximum permissions of the block's individual locations.[8] Blocks may of course be freed, but in CompCert's memory model, freed blocks are never re-allocated (each new allocation takes a fresh block-number from a countable set of positive numbers). The unchOn conditions impose a frame discipline, by confining the effects of the commands to addresses specifiable using j. In particular, $\mathsf{unchOn}\ (\mathsf{loc\_unmapped}\ j)\ m_1\ m_1'$ requires that m_1' contain identical values as m_1 in all blocks b outside the preimage of j, while $\mathsf{unchOn}\ (\mathsf{loc\_out\_of\_reach}\ j\ m_1)\ m_2\ m_2'$ imposes preservation of values at m_2 address whose preimage under j has empty Max permission.

In addition to the clauses in Definition 1, our formal definition imposes some structural conditions on the relation $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$, such as a constraint that global environments be suitably preserved (notation $\mathsf{preservesGlobals}\ ge_1\ j$), and that all blocks mentioned by j be valid in the respective memories. We omit the details of these clauses from our presentation.

We denote simulations for extension and equality passes by $L_1 \preceq_{\mathsf{Ext}} L_2$ and $L_1 \preceq_{\mathsf{Eq}} L_2$, respectively—the definitions of these notions mirror that of $L_1 \preceq_{\mathsf{Inj}} L_2$, but we omit the details. We write $. \preceq .$ for the union of all three relations.

4.1 Transitive Composition of Simulations

In order to verify a multiphase compiler in a modular way, it is critically important to *transitively compose* correctness proofs of individual compiler phases. That is, we would like to prove that $L_1 \preceq L_3$ holds whenever $L_1 \preceq L_2$ and $L_2 \preceq L_3$. In the following, we summarize our Coq proof of this result.

A *cooperative core semantics* is a core semantics such that $ge \vdash \langle c, m\rangle \longmapsto \langle c', m'\rangle$ implies $\mathsf{forward}\ m\ m'$.

Theorem 1. *For cooperative L_1, L_2, L_3, suppose $L_1 \preceq L_2$ and $L_2 \preceq L_3$. Then there exists a simulation $L_1 \preceq L_3$.*

The proof of this result proceeds by case distinction on $L_1 \preceq L_2$ and $L_2 \preceq L_3$, yielding nine cases. The resulting simulation is of type $. \preceq_{\mathsf{Inj}} .$ whenever at least one hypothesis is an injection, is of type $. \preceq_{\mathsf{Eq}} .$ if both hypotheses are equalities,

[8] The current permission at each memory location may fluctuate arbitrarily so long as it does not exceed the Max permission.

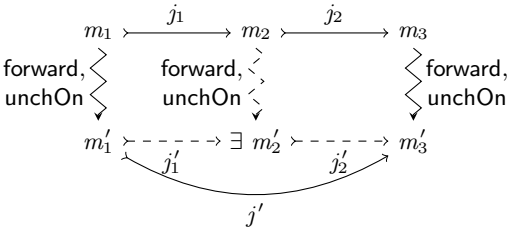


Fig. 6. Interpolation lemma for composing injection phases $L_1 \preceq_{\mathsf{Inj}} L_2$ and $L_2 \preceq_{\mathsf{Inj}} L_3$. Solid lines represent assumptions; dashed lines represent constraints that the constructed m_2' has to satisfy. Similar lemmas have been validated in Coq for all combinations of injection and extension passes.

and is of type . $\preceq_{\mathsf{Ext}}$. otherwise. Each case consists of subgoals according to the clauses in Definition 1 (or the similar clauses in case of . $\preceq_{\mathsf{Ext}}$. and . $\preceq_{\mathsf{Eq}}$.).

The most interesting subgoals are those for the after_external-clauses.[9] In order to establish the desired relation $\langle c_1', m_1' \rangle \sim_{j'} \langle c_3', m_3' \rangle$ between the return states in languages L_1 and L_3, one would like to appeal to the corresponding relations that are inductively given for $L_1 \preceq L_2$ and $L_2 \preceq L_3$. However, in order for these induction hypotheses to apply, one must provide a suitable intermediate state $\langle c_2', m_2' \rangle$, and in particular the memory m_2'. Figure 6 depicts this situation for the case in which both compiler phases are injection passes. As illustrated in the figure, we require the existence of a post-call memory m_2' in L_2 such that m_1' can be injected to m_2' (via an extension j_1' of j_1) and m_2' can be injected to m_3' via j_2', such that $j' = j_2' \circ j_1'$ ($j_2 \circ j_1$ defines injection composition). This is assuming j_1 injects m_1 to m_2, j_2 injects m_2 to m_3, and j' injects m_1' to m_3'.

Prior to CompCert 2.0, memory injections did not compose, *i.e.* $m_1 \rightarrowtail_{j_2 \circ j_1} m_3$ did not follow from $m_1 \rightarrowtail_{j_1} m_2$ and $m_2 \rightarrowtail_{j_2} m_3$. Because the simulations did not expose memory, transitive compiler correctness did not require this property to hold. In CompCert 2.0, Leroy respecified injections to facilitate such composition, based on a suggestion of Tahina Ramananandro. The interpolation lemma provides the counterpart to this composition, by guaranteeing that the post-call injection $m_1' \rightarrowtail_{j'} m_3'$ can be split into some m_2', j_1', and j_2' with $m_1' \rightarrowtail_{j_1'} m_2'$ and $m_2' \rightarrowtail_{j_2'} m_3'$. Moreover, these items can be constructed in such a way that the evolution $m_2 \rightsquigarrow m_2'$ inherits the appropriate forward and unchOn properties from the extremal evolutions $m_1 \rightsquigarrow m_1'$ and $m_3 \rightsquigarrow m_3'$. Our proofs of the interpolation lemmas suggested a handful of additional alterations to the memory model, which we communicated to Leroy. These included a subtle refinement to the treatment of permissions across external calls and a tweak to the definition of unchOn. Leroy installed these modifications in CompCert 2.0, and we formally validated the interpolation lemma in Coq. That is, we have proved that intermediate memories m_2', and injections j_1' and j_2' with the required properties can indeed be constructed.[10] Having proved similar lemmas for the cases where

[9] Indeed, a principal result of this paper is that one *can* reason about the interaction between the memory manipulations of the compiler and the memory effects of external function calls.

[10] Differences between injections and extensions mean that the intermediate memories differ slightly between these cases, although several auxiliary lemmas are shared. As a

one or both of the phases are memory extension translations, we combined the interpolation lemmas to a Coq proof of Theorem 1.

The evolutions $m_i \rightsquigarrow m_i'$ in Figure 6 are stated purely extensionally, in terms of forward and unchOn. The alternative would be a requirement to preserve sequences of memory operations—but the question of *which sequences* we'd want to preserve would take us back exactly to square one: simply requiring *all* sequences to be preserved prevents the compiler from optimizing redundant loads/stores and from reordering loads/stores once we refine external calls into (compilable) code. On the other hand, any other equivalence would itself require extensional justification, so nothing would be gained by considering such sequences.

5 Semantics Preservation

As Section 4 showed, LSRs support phase-by-phase verification of a compiler such as CompCert. But what end-to-end result do LSRs guarantee?

This section answers this question, by stating and proving a strong semantics preservation theorem implied by LSRs. In order to simplify the presentation of this theorem, we deal here only with closed programs, *i.e.*, those whose calls to external functions have been fully resolved after linking. But LSRs imply a strong semantics preservation theorem for open programs as well.

We prove semantics preservation as a corollary of safety preservation, under the following definition of program safety.

Definition 2 (Safety). *A program* $\langle c, m\rangle$ *is* safe for n steps *with postcondition* Q *and in global environment* ge *when either*

- $n = 0$, *or*
- $n \neq 0$ *and if* halted $c =$ Some v *then* $Q(v, m)$ *else there exist* c', m' *such that* $ge \vdash \langle c, m\rangle \longmapsto \langle c', m'\rangle$ *and* $\langle c', m'\rangle$ *is safe for* $n - 1$ *steps.*

A program $\langle c, m\rangle$ *is* safe *iff it is safe for all* n.

In particular, if $\langle c, m\rangle$ is safe with postcondition Q, then $\langle c, m\rangle$ will either infinite loop or halt in a state (return value and memory) satisfying Q.

We state safety preservation in terms of this definition of safety as follows.

Theorem 2 (Safety Preservation). *Let* M *be the type of CompCert memories;* $L_1 = (G_1, C_1, M)$ *be the source semantics;* $L_2 = (G_2, C_2, M)$ *be the target semantics;* $\langle c_1, m_1\rangle$ *be a configuration of* L_1*; and* $\langle c_2, m_2\rangle$ *be a configuration of* L_2*. Assume that* L_1 *and* L_2 *are deterministic, and that* $L_1 \preceq_{\mathsf{Inj}} L_2$ *holds with* $\langle c_1, m_1\rangle \sim_j \langle c_2, m_2\rangle$ *for some injection* j*. Then for all postconditions* Q *up to injection, if* $\langle c_1, m_1\rangle$ *is safe for* Q *then* $\langle c_2, m_2\rangle$ *is safe for* Q.

consequence of our work, it became apparent that under certain conditions, namely the absence of pointers to previously unallocated blocks, memory extensions are special cases of memory injections. This opens the potential to unify the two notions across the entire CompCert development, and hence to coalesce all interpolation lemmas. The price is that all languages would have to (be proven to) preserve the absence of such wild pointers throughout execution. At present, it is unclear which way CompCert will eventually go, so we adopt the status quo for the time being.

By "postconditions Q up to injection," we mean the set of predicates on return values and memories that remain true under injection of their arguments. Formulating Theorem 2 with respect to $. \preceq_{\mathsf{Inj}} .$ is appropriate since this is the type of simulation one obtains when composing all translation phases of CompCert. In the Verified Software Toolchain [App11], a proof of $\{P\}c\{Q\}$ in our C light program logic [A$^+$14] and soundness of the logic together give us the required safety theorem for source-language C light configurations.

As a corollary of Theorem 2 and from termination preservation, we get the following semantics preservation result.

Corollary 1 (Semantics Preservation). *For any execution of L_1 starting in an initial state (module entry point) and resulting in a halted state, and for any observation Q up to injection one could make of that halted state, there is a unique execution starting from the initial state of L_2 that terminates in a unique halted state also satisfying the predicate Q.*

These top-level theorems concern fully linked programs, but our results on LSRs allow the extensions of these theorems to the situation in which a thread interacts with its environment using shared-memory interaction, provided external functions are equipped with suitable up-to-injection specifications. To get an even stronger result regarding fully separate compilation, some additional constraints need to be imposed on CompCert's specification: that all assumptions made in the *External Steps* clause are established by the *Core Steps* clause.

6 Backwards Compatibility

Although CompCert 2.0's top-level correctness theorems still say very little about the memory transformations performed by the compiler, many of the compiler's internal invariants—established in proofs of individual compiler phases—make more precise statements about these memory transformations. In order to maximize the reuse of CompCert's proof infrastructure, it is desirable to preserve as many of these internal invariants as possible.

To evaluate our approach, we adapted the proof of one of the trickiest compilation passes, Cminorgen, to the simulation structures of Section 4. The main task of Cminorgen is to remove from activation records any local variables that are not address-taken (these local variables are allocated in registers, or occasionally spilled back into activation records after CompCert's register allocation pass). Because the Cminorgen pass significantly reorganizes memory, CompCert 2.0 proves Cminorgen as an injection pass.

First, we refactored the source and target languages of the transformation, Csharpminor and Cminor, as described for C light in Section 3.1. This involved isolating the memory component from the core data and giving definitions for the core semantics interface from Figure 3. Next, we adapted the proof, by exposing the memory injection in a match relation that described the evolution of the call stack, and reassembling the main inductive argument. While most instruction forms were rather easy to adapt, the rule for internal function calls required a

slight strengthening of invariants in order to establish the $j \bowtie_{m_1;m_2} j'$ condition of clause 3 from Definition 1. In contrast, the case for external functions, which in CompCert 1.x and 2.0 was rather involved, disappeared completely since external function calls are now handled by the core semantics interface.

That the adaptation of the Cminorgen proof, one of CompCert's most complicated phases, to our setting was reasonably straightforward indicates that our proof techniques will scale to the remainder of CompCert. Indeed, many of CompCert's phases make no adaptations to the memory layout at all. For these phases, all that is needed is to adapt the source and target languages to the core semantics interface of Section 3, and its strengthening to cooperative semantics.

7 Related Work

Some of the most appealing treatments of compiler correctness to-date have been developed in the setting of ML-like languages or (typed) lambda-calculi [Plo73,Rey74]. Proof techniques such as logical relations or $\top\top$-closure exploit type structure to capture the relationship between code and its execution context, supporting advanced language features such as higher-order functions, existential or recursive types, polymorphism, and references. But the property of transitive compositionality has often been difficult to obtain.

Inspired in part by Pitts and Stark's work on *Kripke logical relations* [PS98], recent years have seen progress on supporting local state in the form of mutable references [ADR09,DNB10,HDNV12,HNDV13]. State transition systems, which are similar in some respects to our protocol-oriented semantics, figure prominently in many of the recent approaches as a means of specifying invariants on local state. However, this work has all been done in the context of strongly typed functional languages, *e.g.*, System F extended with recursive types and mutable references. Our context and goals are different: we apply logical simulation relations to the problem of verified separate compilation of a weakly typed language (C), in the context of a realistic optimizing compiler (CompCert). The application to CompCert is one of the major contributions of our work.

Arguably, the most closely related work to ours is Hur *et al.*'s integration of bisimulations and Kripke logical relations [HDNV12]. Hur *et al.* achieve compositionality for the rich setting of $F^{\mu!}$, but employ a highly nonstandard construction: the cardinalities of the syntactic categories for types and values, and of the semantic interpretation of the type **nat**, are exploited to construct "bad" values that have little motivation from a typed perspective (these values occur in the logical relation at function type position, despite being integers) and are then used to "artificially" block certain executions in the intermediate language L_2. In contrast, although being far from trivial, the proofs of our interpolation lemmas have significantly more constructive content.

A strength of Hur *et al.*'s contribution is to capture the intricate interactions between global (shared) and local knowledge. Hur *et al.*'s analysis is formulated using *relation-transition systems* (RTS's), an evolution of the authors' earlier work on using state transition systems to index Kripke-worlds in

step-indexed logical relations [ADR09,DNB10]. Our protocol-oriented interaction model shares many of the features of RTS's but is used to specify the "operational ground truth".

Progress has also been made in extending logical relations to low-level code, and to compilation [BH09,BH10]. One of the challenges is to transform high-level type structure into well-behavedness at the low level even in the presence of more fine-grained observation contexts. Again, the situation for our work is different, as the C language does not provide us with much high-level type structure to start with. It is indeed the memory model, not type structure, that constitutes the *lingua franca* between *C* modules, and the use of a *uniform* memory model across all stages of compilation is a crucial feature of CompCert.

Recently, Liang *et al.* [LFF12] have explored applications of rely-guarantee reasoning [Jon83] to proving the correctness of concurrent program transformations. In that work, rely-guarantee conditions were used to model the interactions of a program thread with its concurrent context. There are natural extensions of these ideas to separate compilation in the CompCert setting. Indeed, we are actively exploring a rely-guarantee simulation proof method that would allow separate compilation of linked modules even in the presence of mutually recursive inter-module dependencies. In the CompCert setting, rely conditions correspond to the assumptions CompCert currently makes about the behaviors of external function calls, and which we expose in Clause 5 of measured forward simulations (Definition 1, Section 4). Adapting CompCert to support symmetric guarantees about compiled code is much trickier. Solving these issues is active research.

Our work on verified compilation of concurrent programs has goals similar to those of CompCertTSO [ŜVZN$^+$11] but with a quite different method. Instead of modeling a specific relaxed memory model, *e.g.*, x86-TSO, as CompCertTSO does, we prove—by instrumenting the CompCert languages with memory permissions—that data race freedom is preserved by compilation. For programs proved data race free in our concurrent separation logic we will therefore get correctness guarantees with respect to even *weaker* memory models than x86-TSO, *e.g.*, the POWER and ARM models. Our approach even permits CompCert to optimize nonsynchronizing loads and stores, *e.g.*, hoist loads/stores, eliminate redundant load/stores, when they do not cross synchronization operations.

8 Conclusion

Compositional compilation is not an easy problem. In this paper, we attack the problem "at scale," in the intensely practical CompCert compiler. In this setting, we show that core semantics and LSRs, together with the program logic [A$^+$14], enable end-to-end verification of C programs that interact via shared memory. But our approach to "how to specify a compiler" is significant beyond just CompCert, and will be relevant to optimizing compilation of any C-like language.

Acknowledgments. We thank the members of the Princeton programming languages group and the ESOP anonymous reviewers for their comments on earlier drafts of this paper. We are indebted to Xavier Leroy and Tahina Ramananandro for many enlightening technical conversations.

This material is based on research sponsored by the 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$^+$14. Andrew W. Appel et al., *Program logics for certified compilers*, Cambridge, 2014.

ADR09. Amal Ahmed, Derek Dreyer, and Andreas Rossberg, *State-dependent representation independence*, POPL, 2009.

App11. Andrew W. Appel, *Verified Software Toolchain*, ESOP, 2011.

BH09. Nick Benton and Chung-Kil Hur, *Biorthogonality, step-indexing and compiler correctness*, ICFP (New York), ICFP, 2009, pp. 97–108.

BH10. ———, *Realizability and compositional compiler correctness for a polymorphic language*, Tech. Report MSR-TR-2010-62, Microsoft Research, 2010.

DNB10. Derek Dreyer, Georg Neis, and Lars Birkedal, *The impact of higher-order state and control effects on local relational reasoning*, ACM SIGPLAN Notices, vol. 45, ACM, 2010, pp. 143–156.

HDNV12. Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis, *The marriage of bisimulations and Kripke logical relations*, POPL, 2012.

HNDV13. Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis, *Parametric bisimulations: A logical step forward*, draft, 2013.

Jon83. Cliff B. Jones, *Tentative steps toward a development method for interfering programs*, TOPLAS **5** (1983), no. 4, 596–619.

L$^+$12. Xavier Leroy et al., *The CompCert memory model, version 2*, Tech. Report RR-7987, INRIA, 2012.

LB08. Xavier Leroy and Sandrine Blazy, *Formal verification of a C-like memory model and its uses for verifying program transformations*, JAR **41** (2008), no. 1.

Ler11. Xavier Leroy, *The CompCert verified compiler, software & ann. proof*, 2011.

LFF12. H. Liang, X. Feng, and M. Fu, *A rely-guarantee-based simulation for verifying concurrent program transformations*, POPL, 2012.

NS06. Zhaozhong Ni and Zhong Shao, *Certified assembly programming with embedded code pointers*, POPL, 2006.

O'H07. Peter W. O'Hearn, *Resources, concurrency and local reasoning*, Theoretical Computer Science **375** (2007), no. 1, 271–307.

Plo73. Gordon Plotkin, *Lambda-definability and logical relations*, School of Artificial Intelligence, University of Edinburgh, 1973.

PS98. Andrew Pitts and Ian Stark, *Operational reasoning for functions with local state*, Higher order operational techniques in semantics (1998), 227–273.

Rey74. John Reynolds, *On the relation between direct and continuation semantics*, Automata, languages and programming (1974), 141–156.

ŜVZN$^+$11. J. Ŝevčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, *Relaxed-memory concurrency and verified compilation*, ACM SIGPLAN Notices **46** (2011), no. 1, 43–54.