Verified Separate Compilation for C

James Gordon Stewart

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Advisor: Andrew W. Appel

June 2015

© Copyright by James Gordon Stewart, 2015. All rights reserved.

Abstract

A separate compiler independently translates a program's components in a way that preserves correctness of the program as a whole. This dissertation develops techniques and tools for verified (mechanically proved) separate compilation of programs in 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.* 2006–2015 and upon which this dissertation builds, has proofs of correctness for whole programs, but its simulation relations are too weak to specify or prove separately compiled modules.

The main contributions of the dissertation are:

(i) language-independent linking, a new operational model of multilanguage module interaction that supports the statement and proof of cross-language contextual equivalence;

(ii) structured simulations, a program-equivalence proof method that enables expressive module-local invariants on the state communicated between compilation units at runtime;

(iii) the application of the above techniques to Compositional CompCert, a verified separate compiler for C. As additional validation, the dissertation demonstrates the connection of Compositional CompCert to the Verifiable C program logic.

Acknowledgments

I thank my adviser, Andrew, for innumerable enlightening conversations, both technical and otherwise. His energy, consistency, and all-around intelligence in research were always encouraging, and occasionally awe-inspiring. He taught me, more than anything, not to be afraid of big complex systems—that is where the most interesting research questions lie.

I thank the members of the PL group at Princeton over the years, including: Ryan Beckett, C.J. Bell, Lennart Beringer, Joey Dodds, Qinxiang Cao, Santiago Cuellar, Rob Dockins, Matt Meola, Chris Monsanto, Olivier Savary Belanger, Cole Schlesinger, and Dave Walker. Each and every one of you helped to make my time at Princeton both enjoyable and edifying.

I thank my research collaborators, both at Princeton and elsewhere: Anindya Banerjee and Aleks Nanevski at IMDEA Software in Madrid, Mahanth Gowda at UIUC, Xavier Leroy at INRIA, Geoff Mainland at Drexel, Bozidar Radunovic and Dimitrios Vytiniotis at MSR Cambridge, the FLINT group at Yale—especially Tahina Ramananandro and Zhong Shao—and Andrew, Lennart, Rob, Santiago, and Josh Kroll at Princeton. I am especially grateful to Lennart: this thesis would not have been possible without our close collaboration over these many years. I thank the Air Force Office of Scientific Research (award FA9550-09-1-0138) and the Defense Advanced Research Projects Agency (award FA8750-12-2-0293) for funding this research.

I thank my parents, Sarah and Gordon, for their support and encouragement. Last but not least, I thank my wife, Zori: You were always there for me, through the good times and the not so good. Graduate school is not easy; I could not have done it without you.

Contents

Contents v

List of Figures viii

1 Introduction 1
1.1 Motivation 4
1.1.1 Verifying Realistic Optimizations 4
1.1.2 Specifying and Compiling Open Programs 9
1.2 Contributions and Thesis Scope 10
1.3 Relation to Previous Work by the Author and Co-Authors 11
1.4 Related Work 11
1.4.1 Whole-Program Compilation 12
1.4.2 Compositional Compilation and Logical Relations 12
1.4.3 Verifying and Compiling Concurrency 13
1.4.4 Game Semantics for Interaction 14
1.4.5 The Bleeding Edge 14

2 The CompCert Memory Model 17
2.1 Memory Model Basics 18
2.1.1 Values, Loads, and Stores 19
2.2 Memory Model, Version 2 22
2.3 Memory Transformations 25
2.4 Validity and Reachability 28
2.5 Global Environments 30

3 Language-Independent Semantics 31
3.1 Interaction Semantics . . . 31
3.2 Examples . . . 33
3.2.1 CompCert Clight . . . 33
3.2.2 CompCert x86 Assembly . . . 38
3.2.3 Gallina Semantics . . . 44
3.2.4 Trace Semantics . . . 45
3.3 Reach-Closed and Valid Semantics . . . 48
3.3.1 Reach-Closed Semantics . . . 48
3.3.2 Valid Semantics . . . 52

4 Language-Independent Linking 55
4.1 Linking Semantics . . . 55
4.2 Contextual Equivalence . . . 63
4.3 Gallina Contexts . . . 63
4.4 Stateful Contexts . . . 64

5 Compiler Correctness 69
5.1 Whole-Program Simulations . . . 71
5.2 Corollaries . . . 75
5.2.1 Termination . . . 75
5.2.2 Safety . . . 77
5.2.3 Behavior Refinement . . . 79
5.3 Open Program Simulations . . . 81
5.3.1 Logical Simulation Relations . . . 81
5.3.2 Structured Simulations . . . 82

6 Separate Compilation 95
6.1 Vertical Composition . . . 95
6.1.1 Interpolation . . . 96
6.2 Horizontal Composition . . . 97
6.2.1 Reach-Closed Contextual Equivalence . . . 100
6.2.2 Linking Invariants . . . 104

7 Modular Verification 113
7.1 Modular C Program Logic . . . 114
7.1.1 Inference Rules . . . 114
7.1.2 Proving Whole Modules . . . 115
7.2 Juicy Memories . . . 121
7.3 Composing End-to-End . . . 129
7.3.1 Safely Linking . . . 131
7.3.2 Squeezing the (Princeton) Orange . . . 136

8 Application to CompCert **139**
8.1 Compositional CompCert 139
8.2 Anatomy of a Phase 141
8.3 Anatomy of the Proof 142

9 Conclusion **145**
9.1 What Has Been Achieved? 145
9.2 Discussion 146
9.3 Future Directions 147
9.4 Conclusions 149

Bibliography **151**

List of Figures

2.1 A CompCert memory (version 1) 18
2.2 Reachability 28

3.1 Interaction Semantics interface 33
3.2 Syntax and semantics of Clight (excerpts) 34
3.3 Syntax and semantics of Clight (continued) 35
3.4 Call rules from the operational semantics of Clight 36
3.5 Syntax and semantics of CompCert x86 assembly (excerpts) . . . 39
3.6 x86 initialization 40
3.7 Reach-Closed Semantics 49
3.8 Valid Semantics 53

4.1 Corestep relation of Linking Semantics $\mathcal{L}$: illustration 58
4.2 Corestep relation of Linking Semantics $\mathcal{L}$ 59
4.3 Interaction Semantics of program linking 61

5.1 Whole-program simulations $S \leq T$ 73
5.2 Structured Injections and the axioms they satisfy 84
5.3 Structured Simulations: Initial Core, At External, and Halted Core clauses 86
5.4 Structured Simulations: Internal and External Step cases 88
5.5 Structured simulations: additional definitions 89
5.6 Graphical representation of Structured Injection leakage 91
5.7 Block leakage 92

6.1 Interpolation lemma for composing injection phases 97

6.2 Schematic representation of the stacks-of-cores linking invariant 105

7.1 Juicy Interaction Semantics 126
7.2 Composing the proofs 130

8.1 The phases of Compositional CompCert 140
8.2 Lines of code for selected parts of the development 143

Chapter 1

Introduction

C is a language of contradictions. On the one hand, it was designed [KR98]—and excels at—giving the programmer fine-grained control over byte-level data representations as they are laid out in memory. Low-level control simplifies the construction of software components like operating systems, device drivers, and other resource-constrained systems for which a garbage-collected language is unsuitable.

At the same time, the C specification [ISO11] fights to maintain a minimum of abstraction, if only to preserve the sanity of C programmers and compiler writers. This "C-level abstraction" does more than just circumscribe control flow (to function call/return and local jumps); it imposes an abstraction layer over data as well, including:

- the distinction of pointers from integers (in particular, casting pointers to integers, and vice versa, is only implementation-defined [ISO11, 6.3.2.3]). This distinction runs counter to the intuitions of many C programmers, who often assume general pointer–integers casts are portable across implementations.
- the notion of memory *object* [ISO11, 3.15], as distinct from the underlying bit- or byte-level representation of that object. For example, pointer arithmetic and comparison in C across memory objects—the run-time representations of language-level constructs such as (addressed) variables—is undefined [ISO11, 6.5.6]. Contrast with assembly language, in which no such distinction exists.
- (weak) typing, in the form of type tags (*e.g.*, `int` or `float*`) ascribed to memory objects. Compilers require types for register allocation and stack-frame management, and take advantage in alias analyses of the

"strict aliasing" condition [ISO11, 6.5], which asserts that two pointers of incompatible types never alias.

- object lifetime. [ISO11, 6.2.4.6] Certain memory objects have lifetime that is block-scoped. Block scoping gives compiler writers the freedom to, *e.g.*, reuse an object's storage once the object's lifetime has ended. The lifetime of a `malloc'd` region extends to the point (in the program execution) at which the region is deallocated, giving the `malloc` implementation the freedom to reuse the freed region.

These abstractions—pointers/integers, objects, weak typing, lifetime, and others—are more than just convenience. They fundamentally enable compilers to do their work. In addition to strict aliasing, which facilitates alias analysis, the correctness of compiler phases that reorganize memory layout, such as stack-frame allocation, register allocation/spilling, function inlining, and stack reuse optimization, depends deeply on whether the program contexts—in which the compiled code will run—respect the C language abstraction. I illustrate this point with multiple examples in the second half of the introduction.

Well-defined C programs are of course valid program contexts (they are, by definition, C-abstraction-preserving). But a compiler, and its correctness proof if one exists, should generalize beyond pure C. Real software systems like operating systems contain components written in multiple languages (*e.g.*, C and assembly), in order to perform tasks at varying levels of abstraction. For example, an OS's scheduler might be written in C while its process switcher and interrupt handlers are written in assembly language. While some assembly-language modules will respect C-level abstractions (at the points of interaction), others will not. **The first question** this thesis answers is,

> *Under which program contexts (assembly or otherwise) is an optimizing C compiler guaranteed to preserve program behavior?*

Specific related technical questions include:

> *How to give semantics to open modules (those that call functions defined in other translation units)?* Answering this question, and the related *How does one reason about equivalence of open modules?* is necessary to state (and prove) correctness of a separate compiler.

> *How to achieve language independence?* Our compiler-correctness theorems should apply regardless of the language in which program contexts are implemented, assuming some basic semantic conditions on context behavior.

Do the techniques scale to the complex features of languages such as C? For example, can we handle addressed stack-allocated local variables? Section 1.1 explains why such features do not mix smoothly with proofs of compiler optimizations that transform program memory layout.

Do the techniques scale to real systems? Compatibility with existing verified compilers for C such as CompCert is an important aspect of this work. Achieving compatibility means significant proof engineering to validate the techniques against the actual compiler transformations performed by CompCert.

The solution to the "which program contexts" question (Chapter 6) is not a new type system or syntactic device, but a set of semantic restrictions that circumscribe the behaviors of valid program contexts in a manner that is independent of the particular language in which the contexts are implemented. In the second half of this chapter, I further motivate by presenting a number of failure cases: what goes wrong when compiled code is linked with contexts that break C-level abstractions, in sometimes subtle ways.

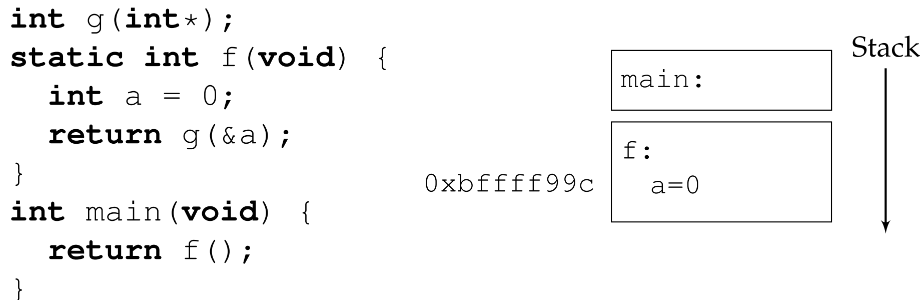
In order to achieve language independence, it was necessary first to define what it means for program modules in C and assembly (and perhaps other languages) to interact. I solve this problem with *interaction semantics* (Chapter 3), which defines the interface of sequential (and well-synchronized concurrent) threads in a language-independent manner, and *language-independent linking* (Chapter 4), which gives the overall semantics of linked programs from the interaction semantics of the underlying modules.

The final contribution of the thesis is the application of interaction semantics, language-independent linking, and the semantic restrictions on program contexts that I develop in Chapter 6 to Compositional CompCert, a verified compositional C compiler. [SBCA14] Compositional CompCert extends the correctness specification/proof of Leroy *et al.*'s CompCert verified C compiler—which dealt only with whole programs—to separate compilation. The major additional technical advance in the proof itself is *structured simulations* (Chapter 5), an extension of Leroy's simulation proofs to support both rely-guarantee reasoning (about the program properties that are assumed and preserved by compilation) and fine-grained invariants on program state that distinguish, *e.g.*, compiler-managed spilled registers from programmer-managed stack-allocated local variables.

1.1 Motivation

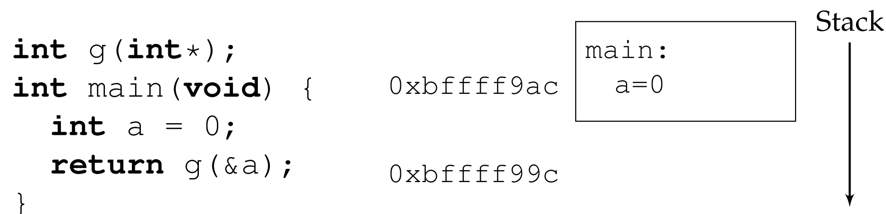
1.1.1 Verifying Realistic Optimizations

The correctness of common compiler optimizations like constant propagation, spilling,[1] dead-code elimination, and function inlining is sensitive to the program contexts in which the compiled code is executed. As example, consider the following C program fragment:



The code on the left starts from `main`, which calls internal function `f`, which in turn calls external function `g(&a)` (declared in this module but defined by another translation unit), passing the address of the stack-allocated local variable `a` as argument. To the right of the code, I give a schematic representation of the memory state at the point at which `g` is called: The stack grows downward; the outlined blocks are the activation records for the calls to `main` and `f` respectively.

So far, so good. But consider for a moment how the picture changes if the compiler decides to inline `f`:



In the code on the left, the (static) function `f` has been removed entirely (it has internal linkage, and therefore could not have been called from an external translation unit). In `main`, the body of `f` has been inlined at what

[1]Spilling, which is performed after register allocation, moves temporaries that cannot be allocated in registers into function activation records.

was previously the call point. The variable a is now declared and initialized in main rather than f.

In the memory diagram to the right, the two activation records for main and f have been coalesced into a single, slightly larger stack frame for main. Because a is stack-allocated at function entry for main, instead of f, it is placed in memory at a different location than it was previously, before function inlining was performed.

The problem here is that the function g is now passed a different pointer than it was before (the pointer references the same memory object, containing the value of variable a, but the object itself has been allocated at a different address). A craftily constructed implementation of g, such as the following (bad) C module:

```
int g(int* p) {
  return ((uintptr_t)p==0xbffff99c);
}
```

could in principle distinguish the two program fragments, pre- and post-function inlining, by cleverly choosing the integer 0xbffff99c to equal the address at which a is allocated before inlining has been performed. When this implementation of g is linked to the program fragment above, C compilers such as gcc and CompCert produce programs that generate different return values, depending on whether function inlining is enabled or not. This behavior does *not* indicate the presence of a bug in the compilers. Instead, it is evidence that—from the perspective of the compiler writers—g is an overly sensitive program context; the result of executing g depends too much on implementation details of the translation unit(s) it is linked with.

Constant Propagation. There is not much a compiler writer can do if program contexts like g have the power to interrogate memory at arbitrary addresses. Such contexts break all abstraction, and therefore rule out most, if not all, program optimizations that reorganize memory in nontrivial ways.

There are more subtle abstraction-breaking behaviors, however. Consider the following C program fragment:

```
void g(int*);
int f(void) {
  int a; int b = 3;
  g(&a);
  return b;
}
```

Function f declares two local integer variables, a and b. It then calls external function g, passing the address of variable a as argument.

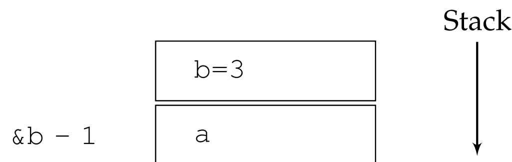
What value does function f return? The answer depends, of course, on the implementation of g. For example, if g is the following (bad) C program:

```
void g(int* p) {
  *(p + 1) = 4;
}
```

and we compile and execute with gcc[2] at optimization level 0, we get result:

```
> gcc -O0 f.c g.c; ./a.out; echo $?
> 4
```

At this optimization level, gcc stack-allocates both a and b, in the following configuration:



The write to *(p + 1) in g becomes a write to &a + 1 == &b, which overwrites the value of b to 4.

But now consider what happens if, viewing f in isolation, we apply a standard program optimization like constant propagation, resulting in the new program:

```
int f'(void) {
  int a;
  g(&a);
  return 3;
}
```

The optimized f' clearly has different behavior, when linked with g, than the original f (it returns 3 instead of 4). This new behavior can be demonstrated by compiling the original program at optimization level 1:

```
> gcc -O1 f.c g.c; ./a.out; echo $?
> 3
```

[2]Version 4.7.2.

The problem, again, is not the compiler (soundness of basic optimizations like constant propagation is uncontroversial) but the context `g`. By writing to address `&a + 1`, `g` breaks the C-language abstraction. Variables `a` and `b` represent two distinct runtime objects; pointer arithmetic between them is undefined.

Stack-Reuse Optimization. Compiler optimizations can take advantage of the C object abstraction in even subtler ways. Consider this C program:

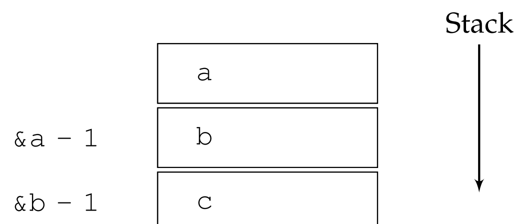
```
int g(int*, int*);
void f(void) {
  int a;
  {
    int b;
    printf("%d,", g(&b,&a));
  }
  {
    int c;
    printf("%d\n", g(&c,&a));
  }
}
```

in which `f` allocates three local variables: `a`, which has function scope, and `b` and `c`, which have (nonintersecting) block scope. The function `g` returns the result of the pointer comparison `(p+1)==q`:

```
int g(int* p, int* q) {
  return (p+1)==q;
}
```

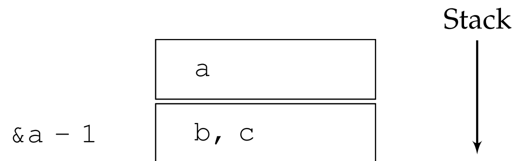
This program prints varied results at multiple different optimization levels. For example, at optimization level 0 the result is "`1,0`". At level 1 it is "`1,1`". At levels 2 and 3 it is "`0,0`".

At optimization level 0, `gcc` allocates three stack slots for the three distinct variables `a`, `b`, `c`, resulting in stack configuration:



In this configuration, `g(&b,&a)== 1`, because `&b + 1 == &a`, whereas `g(&c,&a)== 0`.

At optimization level 1, the compiler reuses `b`'s stack slot for `c` (the two variables have nonintersecting scope), resulting in configuration:



in which variables `b` and `c` have the same address. The fact that `&b==&c` gives result "`1,1`". The final output, "`0,0`", results from the fact that at higher optimization levels, `a` and `b,c` are allocated on the stack in reverse order.

The naive solution to these problems is to place limits on the compiler, *e.g.*, by disallowing compiler optimizations that rearrange memory. But this is too expensive. Many standard compiler optimizations would be ruled out, including function inlining, dead-code elimination, frame-pointer optimization, constant propagation, stack-reuse optimization, *etc.* A second potential solution is to prohibit address-taken local variables (globals, which are typically not rearranged in memory by compilers, would still be addressable). But then we no longer have C. Restricting the language is also too syntactic: it does not easily generalize to other programming languages besides C.

The semantic restrictions on program contexts that I define in this thesis rule out overly concrete contexts like the `g`s above, while still enabling interesting programs. The details of the semantic restrictions are discussed in Chapter 6. At a very high level, they ensure that programs

- treat pointers abstractly, by not comparing pointers with fixed integers, as in the first `g` above;
- respect the C memory and object model, by distinguishing pointers from integers, and by further distinguishing pointers to distinct memory regions/objects;
- respect the interaction model imposed by the external function call protocol (no control flow aside from function call/return across module boundaries).

The advantage of stating and enforcing these conditions semantically, as opposed to syntactically—*e.g.*, via a type system or by restricting in which languages program contexts may be implemented—is the flexibility to model program contexts in a variety of languages: from C (Section 3.2.1) and x86 assembly (Section 3.2.2) to Coq's Gallina (Section 3.2.3).

1.1.2 Specifying and Compiling Open Programs

A presumption of the preceding is that we at least *have* a specification of multilanguage programs. By multilanguage, I mean programs in which some components are written in a language like C while others are written in assembly, or possibly a third language. As Perconti and Ahmed [PA14] have also observed, multilanguage semantics is useful not only for program understanding, but also as a mechanism for stating cross-language contextual equivalence—the compiler correctness criterion I employ in this thesis. What are the difficulties here?

Consider first the whole program case, construed broadly: Imagine P_S is a source program (*e.g.*, in C or some other source language) and P_T the assembly code produced by compiling P_S. Then whole-program compiler correctness states that P_S and P_T have the same observable behavior.[3] How does one express "same observable behavior"? Because P_S and P_T are whole programs, we can prove, *e.g.*, with respect to the big-step semantics of the source and target languages, that $P_S \Downarrow v \iff P_T \Downarrow v$, for all v. Or, in a small-step semantics, we might show that P_S and P_T produce corresponding traces of observable events.

As Benton and Hur [BH10] and Perconti and Ahmed [PA14] have both remarked, things become more difficult when we move from (closed) whole programs to consider open modules (those that call functions declared but not defined in the current translation unit). In this more general setting, multimodule source programs P_S, P'_S are separately compiled to multimodule targets P_T, P'_T. To state correctness, we must say that the semantics of P_S and P_T correspond in some way, and likewise for P'_S and P'_T. However, we cannot simply apply the usual notions of program equivalence here, as we did in the whole-program case. Because the modules are program fragments and not complete programs, we cannot "execute" them in any meaningful sense.

The other force at play is the need for compositionality: Correctness of the translation of one unit should compose with the correctness proofs of other units to yield correctness of the whole program translation. In other words, from (independent) proofs that $P_S \cong P_T$ and $P'_S \cong P'_T$, it should be possible to deduce $P_S \bowtie P'_S \cong P_T \bowtie P'_T$, for some suitable notion of linking $\bowtie$ and program equivalence $\cong$. In the most general case, we will support source programs containing multiple compilation units, each written in a different source language (C, x86 assembly, ML, etc.), each

[3] Or that every behavior of P_T is a possible behavior of P_S, if P_S is nondeterministic (refinement).

calling functions defined either in the other compilation units or by external entities such as an operating system.[4]

I address these issues in Chapters 3, 4, 5, and 6. Chapters 3 and 4 provide a solution to the basic problem of how to specify open modules, both in isolation and in interaction, in the form of interaction and linking semantics, respectively. Chapter 5, on structured simulations, shows how to specify the correctness of compiler transformations on single translation units in a modular way—without reference to the whole program. Chapter 6 proves that structured simulations compose both vertically (*i.e.*, transitively) and horizontally. By horizontal composition, I mean that the structured simulations induced by independently compiling the individual translation units of a multimodule program compose to yield correctness of the whole-program transformation.

1.2 Contributions and Thesis Scope

In summary, the specific contributions of this dissertation are:

- a semantic characterization of the program contexts for which an optimizing C compiler is sound (Chapter 6), answering the question *For which contexts is an optimizing C compiler sound?*
- interaction and language-independent linking semantics (Chapters 3 and 4), which facilitate the statement and proof of cross-language program equivalences (Chapter 6), answering the question *How to achieve language-independence?*
- structured simulations (Chapter 5), a novel extension of CompCert's forward simulation proof method that composes both transitively and horizontally, across program modules, answering the question *How to reason about equivalence of open modules?*
- the application of the above techniques to Compositional CompCert (Chapter 8 and [SBCA14]), the first verified separate compiler for C, answering the question *Do the techniques scale to languages like C, and to existing verified C compilers such as CompCert?*

To substantiate the utility of my approach, I show (Chapter 7) how to connect the Verifiable C program logic [ADH+14] to Compositional CompCert. The result is a system in which program properties can be be proved mod-

[4] A distinct but equally important notion is vertical (*i.e.*, transitive) compositionality of the proofs of distinct compiler phases. Vertical compositionality (*cf.* Section 6.1 of Chapter 6) is required to prove correctness of any realistic multiphase compiler.

ularly at the source level, even of multilanguage programs, and yet are shown to be preserved by separate compilation.

The Coq Proof Development. Except where otherwise indicated in the text, the theorems in this thesis have been proved and machine-checked in Coq. The development is divided between two GitHub repositories:

The Compositional CompCert repository contains the bulk of the development and proofs (corresponding to Chapters 3–6). It is available at: `https://github.com/PrincetonUniversity/compcomp`.

The Verified Software Toolchain (VST) repository contains the proofs that connect the Verifiable C logic to the compiler (Chapter 7): `https://github.com/PrincetonUniversity/VST`.

The thesis text provides pointers into the developments, where appropriate.

1.3 Relation to Previous Work by the Author and Co-Authors

Much of the material in Chapters 2 through 6 is based on previous work by myself and co-authors. Interaction semantics and logical simulation relations (the precursor to the structured simulations of Chapter 5) first appeared in ESOP'14 [BSDA14]. Lennart Beringer did the initial work on structured simulations and adapted many of CompCert's compiler phases. He also proved that structured simulations compose transitively. Language-independent linking and structured simulations are the topic of a paper presented at POPL'15 [SBCA15]. Juicy memories are briefly described in a chapter, which I co-authored, of *Program Logics for Certified Compilers* [ADH+14]. A second co-authored chapter of the same book gives preliminary advice on "How to specify a compiler." Much of the material in Chapter 2 of this dissertation has appeared before, some of it verbatim, in Chapter 32 of [ADH+14], of which I am a co-author.

1.4 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 [MP67]. For a comprehensive survey up to the year 2003, see [Dav03]. Here I focus on the most closely related work.

1.4.1 Whole-Program Compilation

Moore [Moo89] 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 [Ler09], upon which Compositional CompCert is based. Chlipala has also built verified compilers in Coq—first, from lambda calculus to idealized assembly language [Chl07], and later for an impure functional language [Chl10]. But both Chlipala and Leroy's compilers were limited to whole programs—they did not provide correctness guarantees, as I do in this work, about the behavior of separately compiled multimodule programs. More recently, Dockins [Doc12] completed an in-depth study of notions of operational refinement for whole programs, with applications to CompCert and compiler correctness more generally.

1.4.2 Compositional Compilation and Logical Relations

Benton and Hur were two of the first explicitly to do compositional specification of compilers and low-level code fragments, first for a compiler from a simply typed functional language to a variant of Landin's SECD machine [BH09], then for a functional language with polymorphism [BH10]. This initial 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 [HD11], and more recently, to relation transition systems (RTSs) [HDNV12] and the related parametric bisimulations [HNDV13]. 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 they focus on typed higher-order functional languages with only limited forms of shared memory (mutable references), some of the techniques used by Benton, Dreyer, Hur, and their collaborators draw interesting parallels in our own work. Our "us vs. them" protocol (Chapter 5) 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 simula-

tion's view of the memory regions allocated by external functions—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 memory allocation.

An alternative to language-independent interaction semantics is *multi-language semantics* [AB11], which combines several languages of a compiler into a single host language via syntactic boundary casts in the style of Matthews and Findler [MF07]. 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 [PA14]. 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 [McK14].

1.4.3 Verifying and Compiling Concurrency

Liang *et al.*'s work [LFF12] on verifying concurrent program transformations inspired my use of a rely-guarantee discipline, but the complexity of stack frame management, spilling, and block coalescing in Compositional CompCert made it difficult to apply their ideas directly in our setting. Ley-Wild and Nanevski's subjective concurrent separation logic (SCSL) [LWN13] 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 [NLWSD14]. 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).

Also related is verified compilation of concurrent programs. Lochbihler verified a whole-program compiler for multithreaded Java [Loc12]. Sevcík *et al.* built CompCertTSO [SVN+13], which adapted CompCert's correctness proofs to the x86 TSO weak memory model, in order to reason about compilation of racy C code. Mansky's PTRANS framework [Man14] 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 my work. For example, both CompCertTSO and PTRANS lift program refinements from individual threads to whole programs, as I do for interacting modules, under certain noninterference conditions on shared state. One difference is that PTRANS and CompCertTSO both state the noninterference conditions in a "large footprint" way, as global whole-system invariants. My 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 future work to investigate whether the compositional compilation approach I advocate in this thesis could be applied to verified compilation with weak memory models.

1.4.4 Game Semantics for Interaction

The system-level semantics of Ghica and Tzevelekos [GT12] extended standard game models of programs to the more general situation in which the moves of the opponent (environment) are constrained by semantic rather than combinatorial or syntactic restrictions. The key semantic constraint, which Ghica and Tzevelekos call "epistemic" (the environment may only update memory locations it learned about at interaction points), is similar in some ways to the "reach-closed" restrictions I impose on source modules in Theorem 5. In this dissertation, the restriction to reach-closed interaction semantics was a natural side condition of the proof: The kinds of transformations present in Compositional CompCert are just not sound in the context of "omniscient" program contexts that may write to arbitrary memory locations, even those—like return addresses and spills in function stack frames—that are managed by the compiler. One major difference to the work of Ghica and Tzevelekos is that I apply the techniques to the two-program setting of compositional compilation; Ghica and Tzevelekos were concerned primarily with modeling the interactions of a single program module with its environment.

1.4.5 The Bleeding Edge

A number of research groups have recently begun working on compositional compilation for realistic languages. Tahina Ramanandro, along with colleagues at Yale, has proposed a new separate compilation framework [RSW+15] for CompCert that compares favorably in many respects to the approach I describe in this thesis. For example, in Ramanandro's approach, linking is defined semantically on module behaviors, in mixed

big-step/small-step style—which parallels my small-step semantic linking operator $\mathcal{L}$ (Chapter 4). This approach leads to an elegant specification (and proof) of soundness of syntactic linking with respect to the semantic linking semantics, when modules are all implemented in the same language. I do not yet have such a proof (though see Chapter 9 for further discussion). One apparent disadvantage of Ramanandro's approach is that it requires a stronger notion of memory transformation (essentially, bijection) between source and target memories over each compiler transformation. To construct such a bijection for CompCert's Cminorgen phase, it was necessary to add "tags" to memory regions, and to update the semantics of the CompCert languages (*e.g.* Csharpminor and Cminor). Bijective relations are not necessary in Compositional CompCert. Also, my colleagues and I have verified a complete compiler (*cf.* Chapter 8); Ramanandro *et al.* have so far verified only a few of CompCert's (admittedly more difficult) phases.

Very recently, Chung-Kil Hur and Jeehon Kang completed a proof of separate compilation for the most recent version of CompCert (at the time of writing, version 2.4) [Com]. The theorem proved is more limited than that of this thesis; in particular, Hur and Kang's proof does not say anything about linking with modules that were not compiled by CompCert from source modules in CompCert C. At the same time, the Hur and Kang proof is an elegant piece of engineering: it manages to factor the proof to use many of CompCert's forward simulations intact. In private communication, Hur has mentioned that next steps will include the application of recent (unpublished) work, by Hur and others, on *parametric inter-language simulations* [PIL] to CompCert, in order to extend their proof to support linking with more general program contexts.

Wang, Cuellar, and Chlipala, in recent work at OOPSLA [WCC14], showed how to connect verified multilanguage programs to a verified compiler for a small C-like language (Cito). Their approach builds the axiomatic specifications of external functions, as Hoare-style pre/post-conditions on abstract data types, into the operational semantics of their source language. Compositional CompCert avoids tying axiomatic specifications, and thus the details of the program logic, to compiler correctness.

Chapter

2 The CompCert Memory Model

In the semantics of imperative languages, a *memory model* defines the meaning of the memory-manipulating operations supported by the language, such as memory load and store. For toy imperative languages, this model may be as simple as a partial map from locations to values.

$$\mathsf{mem} \triangleq \mathsf{loc} \rightharpoonup \mathsf{val}$$

In a language like C, the memory model is significantly more complicated. It must specify, among other things,

- the C object model, in order to define which pointer arithmetic and comparison operations are valid;
- the byte- vs. word-level representation of data, to handle standard library functions such as `memcpy` [ISO11, 7.24.2.1] and their interaction with normal loads and stores;
- data alignment, *e.g.*, to word boundaries;
- and—for Pthreads-style shared-memory concurrency and certain constant propagation optimizations—permissions on memory values that restrict or enable access to parts of the memory state.

In this chapter, I give background on CompCert's memory model [LABS14] and describe how it handles the above. The first section introduces CompCert memory-model basics, such as the addressing model and value types. These aspects are mostly unchanged even from the earliest versions of CompCert (up to version 1.10, which I collectively call version 1 [LB08]). In Section 2.2, I describe several enhancements to the memory model which I contributed to, including the addition of memory permissions. The original

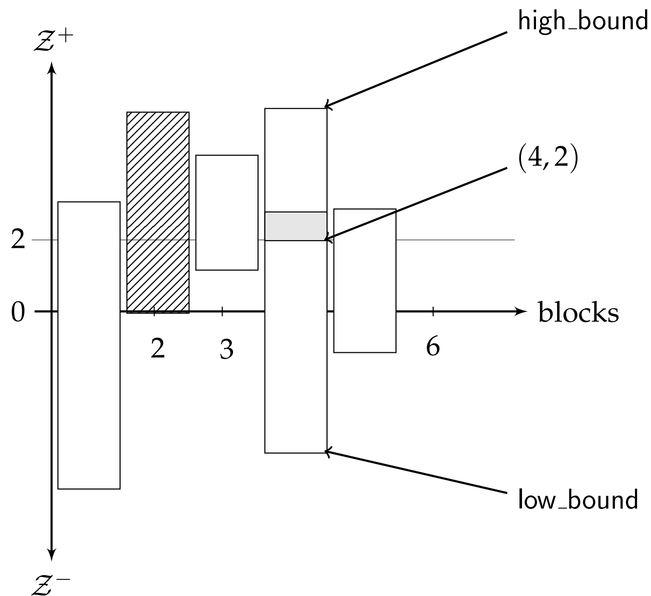


Figure 2.1: A CompCert memory (version 1). The hatched region (block 2) has been deallocated. Address $(b, z) = (4, 2)$ contains an abstract "byte" (in gray). Block 6 is nextblock, the unreserved region to be returned by the next allocation operation.

motivation for permissions was shared-memory concurrency; however, the permissions turned out to be useful in proofs of separate compilation as well (Chapters 5 and 6). Permissions are also used, in standard CompCert, to reason about optimization of read-only global variables.

2.1 Memory Model Basics

The CompCert memory model is block- (or region-) structured. Addresses (b, z) are pairs of a block/region identifier b (a positive number) and an integer offset z. In CompCert's high-level C-like languages (*e.g.*, CompCert C and Clight), blocks are allocated one per global variable, one per call to `malloc`, and—per function invocation—one per addressed local variable. Pointer arithmetic

$$(b, z) + n \triangleq (b, z + n)$$

is then allowed only within, not between blocks. This regime—modeling globals, addressed locals, and `malloc`'d regions as distinct blocks—prevents pointer arithmetic across, *e.g.*, distinct addressed locals or distinct globals, which is undefined in C.

Each address (b, z) specifies an abstract "byte", at offset z in block b. Offsets may be either positive or negative. In the first version of the CompCert memory model, blocks were bounded above and below by two functions: $\mathsf{low_bound}\ m\ b$, which gives the low bound of block b, and $\mathsf{high_bound}\ m\ b$, which gives the upper bound of the block. Loads and stores succeeded only to offsets below the high bound and above the low bound.

Figure 2.1 depicts this situation for a representative CompCert memory. There are 5 allocated blocks, numbered $1 - 5$. Block 2 has been allocated but then freed (indicated by black–white hatching). The CompCert memory allocation model assumes an infinite number of memory regions; it will never reuse block 2. The gray box is an (abstract) bytes at address $(4, 2)$, block 4 at offset 2.

Block 6 is the next free memory region, called nextblock in CompCert. At the next allocation operation

$$\begin{aligned}&\mathsf{alloc} : \mathsf{mem} \to \mathcal{Z} \to \mathcal{Z} \to \mathsf{mem} \times \mathsf{block}\\&\mathsf{alloc}\ m\ lo\ hi\ = (m', \mathsf{nextblock}\ m)\end{aligned}$$

the CompCert memory model returns $(m', \mathsf{nextblock}\ m)$, where $\mathsf{nextblock}\ m$ is the block number of the newly allocated region (= 6 in Figure 2.1 above) and m' is the updated memory state that records the allocation (*e.g.*, by incrementing nextblock). The low–high boundaries of newly allocated memory regions are given—at block allocation time—by the integers lo, hi.

2.1.1 Values, Loads, and Stores

Values. CompCert's intermediate languages share a common value type val, defined by the following inductive data type:

$$\begin{aligned}&\textbf{Inductive}\ \mathsf{val} : \mathsf{Type} \triangleq\\&\quad \mid \mathsf{Vundef} : \mathsf{val}\\&\quad \mid \mathsf{Vint} : \mathsf{int} \to \mathsf{val}\\&\quad \mid \mathsf{Vlong} : \mathsf{int64} \to \mathsf{val}\\&\quad \mid \mathsf{Vfloat} : \mathsf{float} \to \mathsf{val}\\&\quad \mid \mathsf{Vptr} : \mathsf{block} \to \mathsf{int} \to \mathsf{val}.\end{aligned}$$

Vundef is the undefined value, associated to uninitialized local variables. Vint i is an integer value, with i a 32-bit machine integer. Vlongs represent 64-bit machine integers. Vfloats are floating-point numbers. Vptr $b\ i$ is a pointer value, addressing block b and (machine-integer) offset i. To convert from a Vptr to an actual CompCert memory location, one must convert i from type int to type $\mathcal{Z}$, a lossless operation. (Converting from $\mathcal{Z}$ to int can overflow, however.)

Loads. Memory loads in CompCert have the following shape:

$$\text{load} : \text{mem} \to \text{memory_chunk} \to \text{block} \to \mathcal{Z} \to \text{option val}$$
$$\text{load}\ m\ ch\ b\ z = \text{Some}\ v$$

Loading from memory m at address (b, z), with *chunk type* ch, gives value v. The chunk types ch : memory_chunk specify the size, type, and alignment constraints that apply to a given load or store operation.

```
Inductive memory_chunk : Type ≜
  | Mint8signed (* 8−bit signed integer *)
  | Mint8unsigned (* 8−bit unsigned integer *)
  | Mint16signed (* 16−bit signed integer *)
  | Mint16unsigned (* 16−bit unsigned integer *)
  | Mint32 (* 32−bit integer, or pointer *)
  | Mint64 (* 64−bit integer *)
  | Mfloat32 (* 32−bit single−precision float *)
  | Mfloat64 (* 64−bit double−precision float *)
  | Mfloat64al32. (* 64−bit double−precision float, 4−aligned *)
```

For example, a load in C from the address of a 32-bit single-precision float variable, as in the program fragment:

```
float f; *&f;
```

is modeled as load with chunk type Mfloat32. Each chunk type has a natural size $|ch|$ in bytes and a natural alignment $\langle ch \rangle$. For example, |Mfloat64al32| equals 8 while ⟨Mfloat64al32⟩ equals 4 (4-aligned 64-bit double-precision float).

A memory load will fail with None (return type option val) when it is either (i) to a misaligned address (for the given chunk type) or (ii) because the load violates the bounds of the addressed block. The rules, for a load (b, z) at chunk ch, are:

- **Alignment:** $\langle ch \rangle$ divides z;
- **Bounds-Checking:** $\text{low_bound}\ m\ b \leq z \wedge z + |ch| \leq \text{high_bound}\ m\ b$.

Stores. Storing value v in memory m at address (b, z), with chunk type ch, gives new memory m'.

$$\text{store} : \text{mem} \to \text{memory_chunk} \to \text{block} \to \mathcal{Z} \to \text{val} \to \text{option mem}$$
$$\text{store}\ m\ ch\ b\ z\ v = \text{Some}\ m'$$

Stores are partial, like loads. Stores also respect the same alignment and bounds-checking rules as memory loads (stores must be aligned, with respect to the alignment of the memory chunk, and must be within bounds).

The interactions of loads and stores (as well as loads and allocations and loads and frees) are governed by a number of rules, described in detail in Chapter 44 of Appel *et al.*'s new book [ADH+14]. I briefly summarize them here.

The common case is load-after-store, in which we load from the same address (same block and offset) that we previously stored.

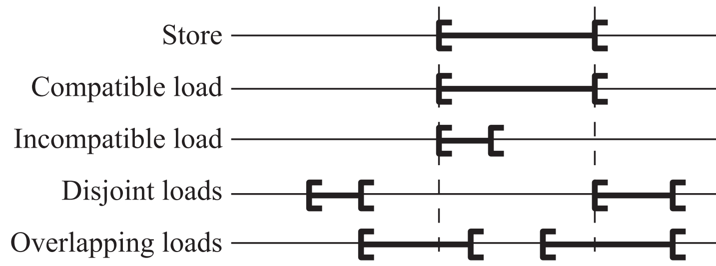
$$\begin{aligned} \mathsf{store}\ m\ ch\ b\ z\ v &= \mathsf{Some}\ m' \\ \mathsf{load}\ m'\ ch'\ b\ z &= \mathsf{Some}\ (\mathsf{convert}\ ch'\ v), \quad \text{if } |ch| = |ch'| \end{aligned}$$

The C standard [ISO11, 6.5] states that load-after-store should succeed only if chunk ch' (representing a C type) is *compatible* with ch, the chunk type at which the address was last assigned. Compatibility means that two types differ only in qualifiers and signedness. CompCert models compatibility by:

- $|ch| = |ch'|$;
- implicitly casting v to type ch at store-time;
- converting v to type ch' at loads.

Think of convert as a C cast. For example, if we attempt to load an integer Vint i at chunk type Mint8signed, convert will return the 8-bit sign extension of i.

The other load-after-store cases are disjoint load–stores, loads that "overlap", and loads which use incompatible types. These cases are summarized in the following figure from Chapter 32 of [ADH+14].



In the "incompatible" and "overlapping" cases, we attempt to load with an incompatible chunk type (*e.g.*, of the wrong size) or from a memory region that only partially overlaps (byte-for-byte) the stored region. Since the first version of the CompCert memory model did not expose the underlying byte representation of values, such "bad" loads just returned None.

2.2 Memory Model, Version 2

Version 2 of the CompCert memory model improved upon version 1 in two major ways. The first was to expose the byte-level representation of values in memory, in order to give semantics to operations like `memcpy` and the overlapping loads I described in the previous section. The second innovation was to add *permissions* to the memory model, which replaced the low_bound and high_bound functions of version 1 and prepared the CompCert compiler and its memory model for connection to a concurrent separation logic (as described briefly in Chapter 44 of [ADH+14]). As it turned out, however, the permissions added in version 2 were also convenient for stating the separate compilation invariants of Compositional CompCert. In this section, I briefly describe the version 2 memory model's support for byte-level data representations. Then I introduce the permission model.

Byte-Level Data Representation. Version 1 of the CompCert memory model gave accurate semantics to most memory loads and stores (and allocations and frees) but hid the underlying byte-level representation of values. In version 2, each location is mapped to a memval, an abstraction of a single byte of data. Memory loads (and stores) then decode (encode) sequences of memvals as values.

The memval$s$ themselves are defined inductively:

```
Inductive memval : Type ≜
  | Undef : memval
  | Byte : byte → memval
  | Pointer : block → int → nat → memval.
```

Undef memvals represent undefined bytes, as might be associated with the values of stack-allocated uninitialized local variables (four Undefs compose a single Vundef).

Bytes are 8-bit machine integers in the range 0...255, which compose larger values such as integers, floats, or longs, taking aspects like endianness and the IEEE encoding of floats into account.

Pointer memvals are abstractions of the bytes that compose pointer values. Pointer b i n is the n-th chunk of a pointer value Vptr b i. The intent is to encode pointers in a way that does not expose their underlying representation (*e.g.*, as bytes, which would be too concrete), but still supports operations like `memcpy`. In the most recent versions of CompCert ($\geq$ 2.4), Pointers are generalized to memory Fragments,

which allow uninterpreted encodings of data values other than just pointers.

Along with the memval interpretation of bytes in memory, CompCert defines operations that encode/decode values to/from sequences of memvals. More details are available in [ADH+14].

Permissions. The second innovation in version 2 was to add permissions to the model, associated with each byte in memory.[1] The permissions are:

Freeable	top permission: can compare, read, write, and free
Writable	can compare, read, and write but not free
Readable	can compare and read but not write or free
Nonempty	can only compare

"Compare" is the ability to compare a pointer to the given location with other pointers. We say a pointer $\mathsf{Vptr}(b, z)$ is valid for pointer comparison if the location (b, z) has at least Nonempty permission.

Permissions accumulate: having permission p implies having all permissions $p' < p$, where the permission order is defined

$$\mathsf{Nonempty} < \mathsf{Readable} < \mathsf{Writable} < \mathsf{Freeable}$$

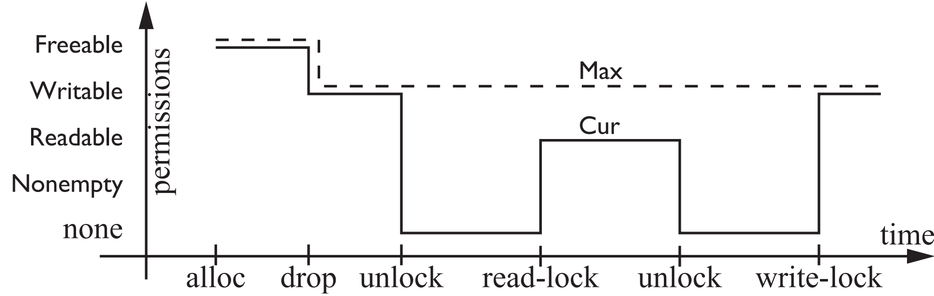
A memory location may have no permission at all. In this case, we say that the location is *empty*. This is typically the case for locations that have not yet been allocated, or which have already been freed.

Every byte location is associated not to one, but to two permissions: the *current* permission and the *max* permission. Throughout an execution, the current permission is always less than or equal to the max permission. The max permission evolves predictably over a location's lifetime: when the location is allocated, it has max permission Freeable; this permission can later be lowered by a `drop_perm` operation;[2] finally, freeing the location removes all its max permissions, making the location empty. The max permission can only decrease once the location has been allocated. In contrast, the current permission can decrease or increase (without ever exceeding

[1] The material in this subsection has significant overlap, some of it verbatim, with [ADH+14, Chapter 32], of which I am a co-author. See that work for further details.

[2] `drop_perm` was added in version 2 and is used to model, *e.g.*, the lowering of a constant global's permission from Freeable to Readable.

the max permission) during the lifetime of the location. For example, in a (proposed) extension to shared memory concurrency, an `unlock` operation temporarily drops current permissions, which can be recovered by a subsequent `lock` operation. The following figure (also from [ADH+14, Chapter 32]) illustrates:



The association of permissions to locations is defined by the predicate:

$$\mathsf{perm} : \mathsf{mem} \to \mathsf{block} \to \mathcal{Z} \to \mathsf{perm_kind} \to \mathsf{permission} \to \mathsf{Prop}$$

where perm_kind is the inductive Max | Cur. The proposition perm $m\ b\ z\ k\ p$ means memory state m at location (b, z) has k-permission at least p. The cumulativity of permissions, and the fact that current permissions are never above max permissions, are expressed by the following implications:

$$\begin{aligned} \mathsf{perm}\ m\ b\ z\ k\ p \wedge p' \leq p &\implies \mathsf{perm}\ m\ b\ z\ k\ p' \\ \mathsf{perm}\ m\ b\ z\ \mathsf{Cur}\ p &\implies \mathsf{perm}\ m\ b\ z\ \mathsf{Max}\ p \end{aligned}$$

In version 1 of the memory model, load and store operations checked that offsets were within bounds. In version 2, load and store instead check that the accessed locations have current permissions at least Readable (Writable for store). Likewise, free checks that the affected locations have current permissions at least Freeable. Defining

$$\begin{aligned} &\mathsf{range_perm}\ (m : \mathsf{mem})\ (b : \mathsf{block})\ (lo\ hi : \mathcal{Z}) \\ &\qquad (k : \mathsf{perm_kind})\ (p : \mathsf{permission}) : \mathsf{Prop} \triangleq \\ &\forall z.\ lo \leq z < hi \implies \mathsf{perm}\ m\ b\ z\ k\ p. \end{aligned}$$

as the predicate that is true iff all offsets between lo and hi have permission at least p, we get the following access conditions for the various memory operations:

Operation...	succeeds if and only if...
load $m\ ch\ b\ z$	range_perm $m\ b\ z\ (z + \lvert ch \rvert)$ Cur Readable
store $m\ ch\ b\ z\ v$	range_perm $m\ b\ z\ (z + \lvert ch \rvert)$ Cur Writable
free $m\ b\ l\ h$	range_perm $m\ b\ l\ h$ Cur Freeable
drop_perm $m\ b\ l\ h\ p$	range_perm $m\ b\ l\ h$ Cur Freeable

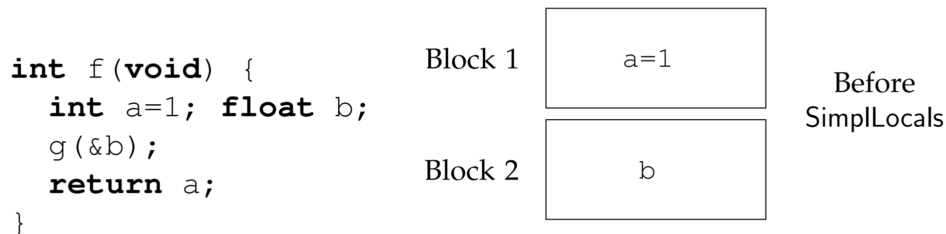
In general, permissions are preserved by operations over memory states, with the following exceptions:

- alloc $m\ l\ h = (m', b)$ sets Max and Cur to Freeable in range (b, l) to $(b, h-1)$.
- free $m\ b\ l\ h$ drops all permissions to empty in (b, l) to $(b, h-1)$.
- drop_perm $m\ b\ l\ h\ p$ sets Max and Cur in range $(l, h-1)$ to p.

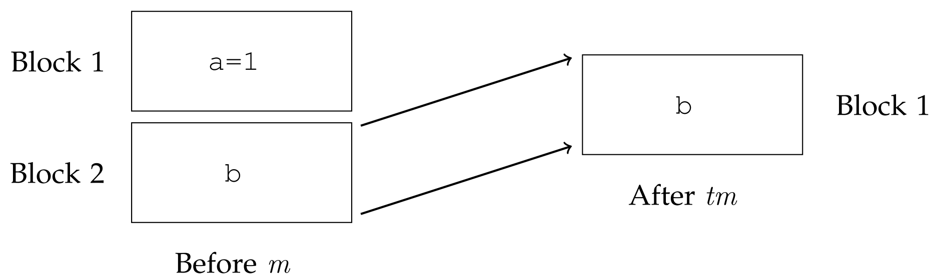
2.3 Memory Transformations

To be useful in compiler proofs, a C memory model must also support the memory transformations performed by an optimizing compiler: spilling and reloading, dead-code elimination, function inlining, *etc.* Operations like load and store should be insensitive to the memory transformations performed by these optimizations. Otherwise, the optimizations may change the observable behavior of the program (by causing, *e.g.*, a store that succeeded before compilation to fail afterward).

Consider CompCert's SimplLocals phase, which pulls unaddressed local variables—such as a in the following C program—out of memory and into a temporaries (register) environment.



In CompCert's highest-level languages, invoking f will generate a memory state that contains two blocks for f's locals, one for a (call it block 1) and another for b (call it block 2). SimplLocals will detect that a is never addressed, however, which means it can promote a to a register.



The memory before the transformation is m, the memory after tm. After SimplLocals, `f` allocates just one block (for `b`) instead of two blocks. The identifiers assigned to blocks have also changed: `b`'s block (formerly block 2) is renamed block 1.

To model transformations of this form, CompCert uses a generalization of block renaming called *memory injection*:

$$\begin{array}{l} f : \mathsf{block} \rightarrow \mathsf{option}\ (\mathsf{block} \times \mathcal{Z}) \\ f\ b = \mathsf{Some}\ (b', \delta) \end{array}$$

A memory injection f maps a subset of the blocks $b \in \mathsf{dom}(m)$ to new blocks $b' \in \mathsf{dom}(tm)$, at offset δ. For example, the SimplLocals transformation described above is modeled by the memory injection

$$\begin{array}{rcl} f\ (\text{Block } 1) & = & \mathsf{None} \\ f\ (\text{Block } 2) & = & \mathsf{Some}\ (\text{Block } 2, 0) \end{array}$$

that maps block 1 to None and block 2 to block 1, at offset 0.

We extend the relation f to memvals as follows:

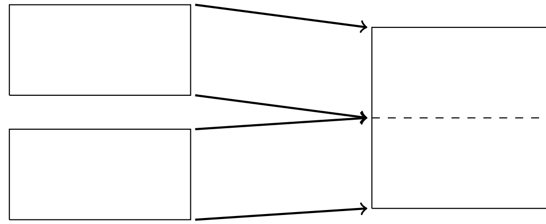
$$\begin{array}{l} \mathsf{memval_inject}\ f\ \mathsf{Undef}\ mv \\ \mathsf{memval_inject}\ f\ (\mathsf{Byte}\ n)\ (\mathsf{Byte}\ n) \\ \mathsf{memval_inject}\ f\ (\mathsf{Pointer}\ b\ z\ n)\ (\mathsf{Pointer}\ b'\ z'\ n) \\ \quad \textit{iff}\ f\ b = \mathsf{Some}\ (b', \delta) \wedge z' = z + \delta \end{array}$$

The relation

$$\mathsf{val_inject}\ f\ v\ v'$$

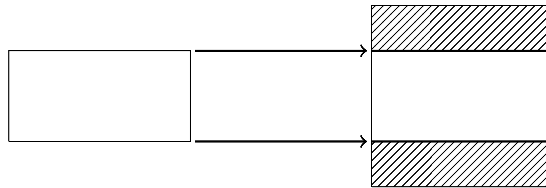
defines the analogous lifting to vals (pointer values are injected; Vundef values are refined to arbitrary values; otherwise, *e.g.*, on integers, val_inject is the identity relation). We use notation $\mathsf{vals_inject}\ f\ \vec{v}\ \vec{v}'$ to denote the pairwise application of $\mathsf{val_inject}\ f$ to the sequences of values $\vec{v}$ and $\vec{v}'$.

Memory injections support more complicated memory transformations as well. For example, CompCert's Cminorgen phase coalesces the multiple blocks allocated at a given function invocation into a single "Cminor" stack block, to facilitate the final layout of stack frames in memory in the Stacking phase. This transformation has the following general form:



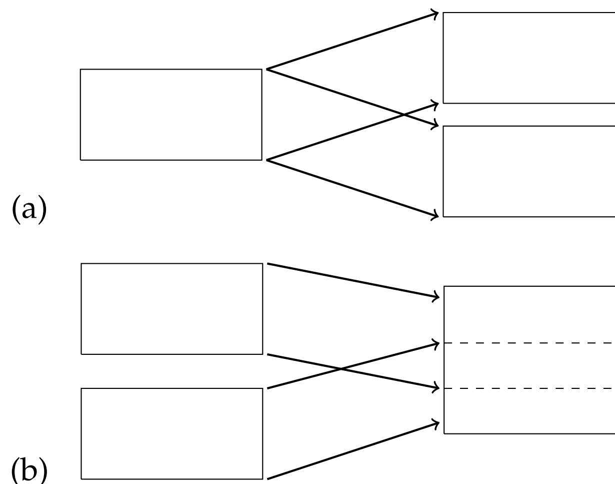
The two blocks on the left are mapped into a single larger block on the right, with the upper-left block being transposed at a nonzero positive offset into the block on the right.

Passes like CompCert's spilling phase may also *extend* memory blocks, in order to accommodate, *e.g.*, registers that have been spilling into stack frames. These transformations have form:



in which the block on the left has been injected into the block at the right (at a zero or nonzero offset). The block on the right contains fresh locations (hatched regions) not present in the left block.

Some memory transformations are not expressible as memory injections. Consider the following two diagrams in which (a) a block is related simultaneously to two blocks; and (b) two blocks are simultaneously mapped to overlapping offsets of a single target region.



Memory injections are functions, ruling out (a). CompCert memory injections prohibit overlap as in (b). Neither (a) nor (b) is needed in CompCert.

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

Figure 2.2: Reachability

However, it's possible that a weakening of memory injections to support (a) and (b) could be useful for future compiler optimizations.

In general, we say a memory injection f injects memory m to tm, written

$$\mathsf{inject}\ f\ m\ tm$$

if (1) permissions at locations in m are preserved in tm under the transformation f; and (2) each readable byte value mapped by f in m is related by memval_inject to the corresponding byte in tm.

Memory injections—which are key for proving correctness of optimizations that reorganize memory layout—behave the same in versions 1 and 2 of the CompCert memory model. However, memory injections are not quite expressive enough to support full separate compilation. Among other things, they do not cleanly distinguish "private" memory regions (the compiler has freedom to optimize these regions more aggressively) from public regions to which pointers may have been leaked. Structured simulations (Chapter 5) enrich memory injections with additional structure to support such fine-grained invariants.

2.4 Validity and Reachability

In addition to structured injections and simulations, which we'll first present in Chapter 5, we require a few additional memory-model-related definitions, some of which are not present in standard CompCert.

We say a memory region b is *reachable* in memory m from a set of root blocks R (REACH $m\ R\ b$, Figure 2.2) when there is a path L of *readable* pointers starting from a root region in R and ending at b. A pointer Vptr (b, z) is readable when location (b, z) has at least Readable permission.

Reachability will play an important role in Chapters 5 and 6. We calculate reachability on memory regions b rather than locations (b, z) because

pointer arithmetic is always allowed within (readable) regions. Hence an entire region is reachable once any location within it is reachable.

The following definitions (already present in standard CompCert) will also prove useful in later chapters. We say a memory region b is valid in memory m

$$\mathsf{valid}\ m\ b\ \triangleq\ b < \mathsf{nextblock}\ m$$

when its index (a natural number) is less than that of the first region in m's free list (CompCert models allocation deterministically, via a pointer nextblock to the next free region in m). Once a region has been allocated, it remains valid for the duration of an execution, even after the region is freed.

We say an entire memory m is valid

$$\mathsf{mem_valid}\ m\ \triangleq\ \mathsf{inject}\ (f_{\mathsf{id}}\ m)\ m\ m$$

when every readable pointer value in m points to a valid (*i.e.*, allocated) block. The definition mem_valid is stated somewhat technically, as the equivalent proposition that all regions in $\mathsf{dom}(m)$ are mapped by the identity injection on m ($f_{\mathsf{id}}\ m$) to themselves. This, in particular, implies that no pointer references a region outside the domain of m.

We say a memory m' evolves *forward*, via one or more execution steps, from an initial memory m when the following holds.

$$\begin{array}{l} \mathsf{forward}\ m\ m'\ \triangleq \\ \quad \forall b.\ \mathsf{valid}\ m\ b\ \implies\ \mathsf{valid}\ m'\ b \\ \wedge\ \forall z.\ \mathsf{max_perm}\ m'\ b\ z\ \sqsubseteq_{\mathsf{perm}}\ \mathsf{max_perm}\ m\ b\ z \end{array}$$

Forward captures the minimal properties that should hold over any sequence of execution steps in any language: (1) valid blocks in m should remain valid in m'; and (2) execution steps should only decrease max permissions (*e.g.*, via drop_perm operations).

Finally, we say a memory m' is *unchanged on* a set of locations L, with respect to an original memory m, when the following are true:

$$\begin{array}{l} \mathsf{unchanged_on}\ m\ m'\ L\ \triangleq \\ \quad \textit{(1) For all locations in } L,\ m \textit{ and } m' \textit{ agree on permissions.} \\ \quad \forall b\ z\ k\ p.\ (b,z) \in L\ \wedge\ \mathsf{valid}\ m\ b \\ \quad \implies\ (\mathsf{perm}\ m\ b\ z\ k\ p\ \iff\ \mathsf{perm}\ m'\ b\ z\ k\ p) \\ \\ \quad \textit{(2) For all locations in } L,\ m \textit{ and } m' \textit{ agree on contents } (\mathsf{memvals}). \\ \wedge\ \forall b\ z.\ (b,z) \in L\ \wedge\ \mathsf{perm}\ m\ b\ z\ \mathsf{Cur}\ \mathsf{Readable}\ \implies\ m(b,z) = m'(b,z) \end{array}$$

2.5 Global Environments

CompCert's global environments, defined by the following record type:

```
Record Genv (F V : Type) : Type ≜
  mkGenv {
    genv_symb : id → option block;
    genv_funs : block → option F;
    genv_vars : block → option (globvar V);
    genv_next : block;
    (* ... properties of the above projections ... *)
  }
```

will also make an appearance in later chapters. Each Genv is parameterized by a type F, of function definitions specific to a particular language (for example, in Clight, F is instantiated to the type of Clight function definitions; in x86 assembly, to assembly code sequences), and by a type V of auxiliary information associated to global variables (also language-specific). Parameterizing Genvs by F and V makes them suitable for use in each of CompCert's intermediate languages (a feature of standard CompCert).

The components of a Genv are:

A symbol table genv_symb mapping global identifiers to the memory regions in which they are allocated;

A function table genv_funs mapping memory blocks to function definitions;

A global variable table genv_vars mapping memory blocks to auxiliary data attached to global variables. globvar V is a record containing a value of type V, variable initialization data, and variable read-only and volatility status;

A pointer genv_next to the lowest-numbered memory region that does not contain global data.

We call the *domain* of a global environment ge the blocks b such that

- there exists an id for which genv_symb $ge\ id = \mathsf{Some}\ b$, or
- $b \in \mathsf{dom}(\mathsf{genv_funs}\ ge)$, or
- $b \in \mathsf{dom}(\mathsf{genv_vars}\ ge)$.

The Genv dependent record also asserts invariants on the global environment (not shown) such as "all blocks marked global by the Genv are less than genv_next."

Chapter

3 Language-Independent Operational Semantics

Interaction semantics is a language-independent model of sequential and (well-synchronized) concurrent threads. The core idea is to phrase interaction, between modules, threads, or other program fragments, as calls to *external functions*. Many kinds of interaction can be modeled in this way, including linking (Chapter 4) and well-synchronized shared-memory concurrency (a future application of the results of this thesis).

I use the term *external function* to describe functions callable but not defined by a particular program unit (declared but not defined, in C terminology). In a concurrent program, a thread might make calls to the external functions `lock` and `unlock`. In a sequential program composed of multiple translation units, one unit may call external functions defined only by another unit.

3.1 Interaction Semantics

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. This protocol models concurrency but also sequential calls to separately compiled functions (spawn a new "thread" to run the call, block until it returns) and single threads 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.

Interaction semantics (Figure 3.1) are a general formulation of this thread protocol. At a high level, an interaction 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. 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.

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

We say an interaction semantics sem is deterministic when its underlying core step relation is deterministic.

Definition 1 (deterministic sem)**.**

$$\forall m\ m'\ m''\ ge\ c\ c'\ c''.$$
$$ge \vdash c, m \longmapsto c', m' \ \wedge\ ge \vdash c, m \longmapsto c'', m'' \implies c' = c'' \wedge m' = m''$$

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, a function pointer value), with arguments (list $\mathcal{V}$). For a standalone program, this function is the entry point `main` (as initialized by the operating system/program loader); for a thread, it is the function that was forked; for a call to a separately compiled module, it is the called function. In general, each module entry point corresponds to an initial_core, at the point at which that entry point is called; internal function calls (to functions defined within the current module) do not call initial_core but instead are handled internally, by the corestep relation of the defining semantics.

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

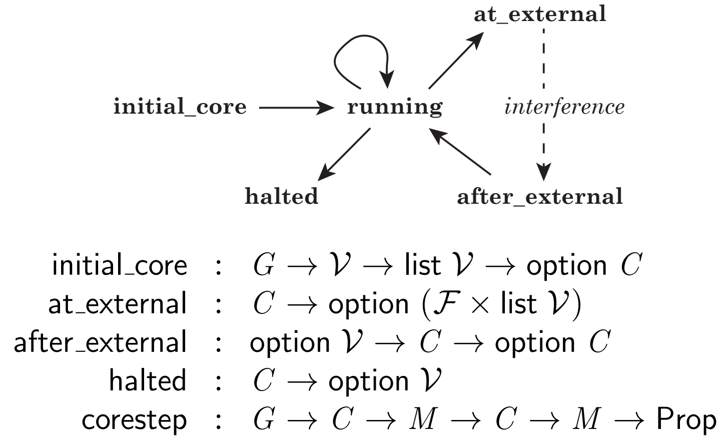


Figure 3.1: Interaction Semantics interface. The types G (global environment), C (core state), and M (memory) are parameters to the interface. $\mathcal{F}$ is the type of external function identifiers. $\mathcal{V}$ is the type of values, and Prop is Coq's type of propositions, Prop. By convention, initial_core takes a pointer of type $\mathcal{V}$ to the function to be called, rather than a function identifier $\mathcal{F}$. The names initial_core, at_external, after_external, halted are not constructors, but are (proved) disjoint predicates.

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.

3.2 Examples

3.2.1 CompCert Clight

As an example of an interaction semantics, I show CompCert Clight [BDL06]. 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.

Figures 3.2 and 3.3 give the syntax of Clight. 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.

Statements

s	$::=$	Sskip	no-op
	$\mid$	Sassign a_1 a_2	lval $\leftarrow$ rval
	$\mid$	Sset id a	temp $\leftarrow$ rval
	$\mid$	Scall $optid$ a $\vec{a}$	function call
	$\mid$	Sbuiltin $optid$ f $\vec{\tau}$ $\vec{a}$	intrinsic
	$\mid$	Ssequence s_1 s_2	sequence
	$\mid$	Sifthenelse a s_1 s_2	conditional
	$\mid$	Sloop s_1 s_2	infinite loop
	$\mid$	Sbreak $\mid$ Sreturn a_{opt}	break/return
	$\mid$	Scontinue s	continue statement
	$\mid$	Switch s	switch statement
	$\mid$	Slabel l s	introduce new label
	$\mid$	Sgoto l	unconditional jump

Internal and External Functions

τ	$::=$	int $\mid$ long $\mid$ ptr τ $\mid$ $\cdots$	C types
γ	$::=$	$\cdot \mid (id, \tau), \gamma$	typing environments
f_i	$::=$	return τ	function return type
		params γ	function parameter typing
		locals γ_v	local variable typing
		temps γ_t	temporary variable typing
		body s	function body
f	$::=$	Internal f_i $\mid$ External id_f $\vec{\tau}$ τ	

Figure 3.2: Syntax and semantics of Clight (excerpts). $optid$ in Scall and Sbuiltin statements is the (optional) variable in which to store the return value of the function (may be None if the function has `void` return type).

Continuations

κ	$::=$	Kstop	safe termination
	$\mid$	$\mathsf{Kseq}\ s\ \kappa$	sequential composition
	$\mid$	$\mathsf{Kloop}\ s_1\ s_2\ \kappa$	loop continuation
	$\mid$	$\mathsf{Kswitch}\ \kappa$	catch switch break
	$\mid$	$\mathsf{Kcall}\ optid\ f_i\ \rho_v\ \rho_t\ \kappa$	catch function return

Core States

ρ_v	$::=$	$\cdot \mid (id, (loc \times \tau)), \rho_v$	addressed var. environment
ρ_t	$::=$	$\cdot \mid (id, v), \rho_t$	temporaries environment
c	$::=$	$\mathsf{State}_{\mathsf{CL}}\ f_i\ s\ \kappa\ \rho_v\ \rho_t$	"running" states
	$\mid$	$\mathsf{CallState}\ f\ \vec{v}\ \kappa$	call (internal or external) function f
	$\mid$	$\mathsf{ReturnState}\ v\ \kappa$	return from (internal or external) function

Figure 3.3: Syntax and semantics of Clight (continued). Continuations and core states appear only in the operational semantics.

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 function records contain an external function identifier id_f, a list of argument types $\vec{\tau}$ and a return type τ, where τ is a C type int, long, ptr τ, *etc.* External functions do not contain a function body. The interaction semantics for Clight will stop at external calls and yield control to the execution environment. By convention, I use f and f_i to range over function definitions (Figure 3.2), while id_f range over function names.

Semantics. The semantics of Clight depends on continuations κ, described in the figure, and core states c, which come in three varieties: Normal states $\mathsf{State}_{\mathsf{CL}}$ model the "running" states of a Clight program, during evaluation of anything but function calls, and consist of the current function being executed f_i, the function body s, the control continuation κ, and two environments, ρ_v for mapping address-taken stack variables to their locations in memory, and ρ_t for mapping temporary variables to their values.$\mathsf{CallState}\ f\ \vec{v}\ \kappa$ models Clight programs that are about to call function f (either internal or external) with arguments $\vec{v}$ and continuation κ. $\mathsf{ReturnState}\ v\ \kappa$ gives the state that results after returning from function

$$\frac{\begin{array}{c} ge \vdash a \Downarrow_{\rho_v,\rho_t,m} v_f \qquad ge \vdash \vec{a} \Downarrow_{\rho_v,\rho_t,m} \vec{v} \qquad ge\ v_f = \mathsf{Some}\ f \\ \mathsf{typeOf}\ f = \mathsf{Tfunction}\ \vec{\tau}\ \tau \end{array}}{\begin{array}{c} ge \vdash (\mathsf{State_{CL}}\ f_0\ (\mathsf{Scall}\ id_{opt}\ a\ \vec{a})\ \kappa\ \rho_v\ \rho_t), m \longmapsto \\ (\mathsf{CallState}\ f\ \vec{v}\ (\mathsf{Kcall}\ id_{opt}\ f_0\ \rho_v\ \rho_t\ \kappa)), m \end{array}} \quad (\textsc{Scall})$$

$$\frac{\begin{array}{c} \mathsf{noRepeat}\ (\mathsf{params}\ f_i\ \cup\ \mathsf{locals}\ f_i) \\ \mathsf{allocVars}\ \rho_\emptyset\ m\ (\mathsf{params}\ f_i\ \cup\ \mathsf{locals}\ f_i) = \mathsf{Some}\ (\rho_v, m_1) \\ \mathsf{bindParams}\ \rho_v\ m_1\ (\mathsf{params}\ f_i)\ \vec{v}\ = \mathsf{Some}\ m' \end{array}}{\begin{array}{l} ge \vdash (\mathsf{CallState}\ (\mathsf{Internal}\ f_i)\ \vec{v}\ \kappa), m \longmapsto \\ \quad (\mathsf{State_{CL}}\ f_i\ (\mathsf{body}\ f_i)\ \kappa\ \rho_v\ (\mathsf{initTempEnv}\ (\mathsf{temps}\ f_i))), m' \end{array}} \quad (\textsc{CallInternal})$$

Figure 3.4: Call rules from the operational semantics of Clight

calls (either internal or external). v is the value returned by the callee; κ is the continuation to be executed after the call returns.

Figure 3.4 shows our reformulation of the function call rules of the Clight 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.

The Scall rule steps a run state $\mathsf{State_{CL}}$ calling function a with arguments $\vec{a}$ ($\mathsf{Scall}\ id_{opt}\ a\ \vec{a}$) to a CallState. The result of the call is stored in id_{opt}. The current function context f_0 is pushed into the return continuation Kcall. f is the function being called, and may be either internal or external.

The CallInternal rule steps into a function body. Function parameters and locals are stored in memory: allocVars allocates a new memory region for each parameter/local, producing variable–location mapping ρ_v. bindParams writes the function arguments $\vec{v}$ into the parameter locations in memory. There is no corresponding rule for external function calls (they are at_external).

I define the at_external function of interaction semantics as a straightforward match on a core state c, returning Some $(f, \vec{v})$ when c is a CallState, f is external, and the arguments $\vec{v}$ to f are well-defined (not CompCert's v_{undef} value), and None otherwise:

$$
\begin{array}{l}
\mathsf{cl_at_external}\ c : \mathsf{option}\ (\mathcal{F} \times \mathsf{list}\ \mathcal{V}) \triangleq \\
\quad \textbf{case}\ c\ \textbf{of} \\
\qquad \mid \mathsf{State_{CL}}\ _\ _\ _\ _\ _ \rightarrow \mathsf{None} \\
\qquad \mid \mathsf{Callstate}\ (\mathsf{Internal}\ f_i)\ \vec{v}\ \kappa \rightarrow \mathsf{None} \\
\qquad \mid \mathsf{Callstate}\ (\mathsf{External}\ \mathsf{f}\ \vec{\tau}\ \tau)\ \vec{v}\ \kappa \rightarrow \\
\qquad\quad \textbf{if}\ \mathsf{defined}\ \vec{v}\ \textbf{then}\ \mathsf{Some}\ (f, \vec{v})\ \textbf{else}\ \mathsf{None} \\
\qquad \mid \mathsf{ReturnState}\ _\ _ \rightarrow \mathsf{None}
\end{array}
$$

Clight after_external injects the return value of an external function into a Clight core state as follows:

$$
\begin{array}{l}
\mathsf{cl_after_external}\ v_{opt}\ c : \mathsf{option}\ C \triangleq \\
\quad \textbf{case}\ c\ \textbf{of} \\
\qquad \mid \mathsf{CallState}\ f\ \vec{v}\ \kappa \rightarrow \\
\qquad\quad \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \mid \mathsf{Internal}\ _ \rightarrow \mathsf{None} \\
\qquad\qquad \mid \mathsf{External}\ id_f\ \vec{\tau}\ \tau \rightarrow \\
\qquad\qquad\quad \textbf{case}\ v_{opt}\ \textbf{of} \\
\qquad\qquad\qquad \mid \mathsf{None} \rightarrow \mathsf{Some}\ (\mathsf{ReturnState}\ v_{undef}\ \kappa) \\
\qquad\qquad\qquad \mid \mathsf{Some}\ v \rightarrow \mathsf{Some}\ (\mathsf{ReturnState}\ v\ \kappa) \\
\qquad \mid _ \rightarrow \mathsf{None}
\end{array}
$$

First, we check whether c is a CallState, with continuation κ. If it is, and the function being called was external, we produce a ReturnState with return value v (whenever v_{opt} was Some v) and v_{undef} (whenever v_{opt} was None). In the v_{opt} = None case, as long as the external function that was called has **`void`** return type, the value v_{undef} will never be used by the caller. In all other cases, we just return None.

The definition of initial_core $ge\ v\ \vec{v}$ is also straightforward, 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 $\vec{v}$ are defined and match f_i's type signature, then introduce state

$$\mathsf{CallState}\ (\mathsf{Internal}\ f_i)\ \vec{v}\ \mathsf{Kstop}$$

which immediately steps to the body of function f_i with the initial local variable environment ρ_v that maps the function's formal parameters to its arguments $\vec{v}$. 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.

Readers familiar with CompCert (versions 1.5 through 2.4) will observe the proximity of our definition to Leroy *et al.*'s presentation: our adaptation removes the memory components from the state constructors $\mathsf{State}_{\mathsf{CL}}$, CallState, and ReturnState and adds the definitions of after_external and so on. The operational semantics arises by refactoring the existing definition to match these state representation changes and by removing the rule for external function calls: such calls are handled directly by interaction semantics at_external and after_external.

3.2.2 CompCert x86 Assembly

Adapting x86 assembly (Figure 3.5) 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 Chapter 4, 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 ("C declaration", parameters in pushed in reverse order on the stack), 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 function $B.g$.

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 b_f with arguments $\vec{v}$, we produce state Asm_CallStateIn b_f $\vec{v}$, which immediately steps to a running State. The initial_core function as well as the operational semantics rule that steps an Asm_CallStateIn to a running $\mathsf{State}_{\mathsf{ASM}}$ are given in Figure 3.6. As a side effect of this step we allocate a "dummy" stack frame in memory in which we store the incoming arguments $\vec{v}$, in right-to-left `cdecl` order as expected by CompCert and `gcc`. (Asm_CallStateOut performs the symmetric step of marshalling arguments out of memory.)

In initial_core, we first check (line 3) that v is a function pointer. If it is, we look up the function body f_i associated with the pointer (line 5), if any, and then check that the arguments to the function match f's type signature (line 9), the arguments are defined (line 10), and that the arguments are representable in memory (also line 10). The last check is subtle: it is possible that the arguments $\vec{v}$ overflow the address space, in which case the values written into the initial stack frame do not directly match $\vec{v}$. The 2 in this

Registers

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

Instructions

$$
\begin{array}{rcll}
p & ::= & \mathsf{MOV_{RR}}\ r_i\ r_i \mid \mathsf{MOV_{RI}}\ r_i\ i \mid \cdots & \text{moves} \\
 & \mid & \mathsf{JMP_L}\ l \mid \mathsf{JMP_S}\ id \mid \mathsf{JMP_c}\ cond\ l \mid \cdots & \text{jumps} \\
 & \mid & \mathsf{CALL_S}\ id \mid \mathsf{CALL_R}\ r_i \mid \mathsf{RET} \mid \cdots & \text{calls/return} \\
 & \mid & \cdots \quad \text{moves with conversion, integer arithmetic, } \textit{etc.} &
\end{array}
$$

Load Frames

$$
lf \ ::= \ \mathsf{mkLoadFrame}\ b_f\ \tau_0
$$

Core States

$$
\begin{array}{rcll}
d & ::= & \mathsf{State_{ASM}}\ rs\ lf & \text{normal states} \\
 & \mid & \mathsf{Asm_CallStateIn}\ b_f\ \vec{v} & \text{marshall args. in} \\
 & \mid & \mathsf{Asm_CallStateOut}\ (b_f, \vec{\tau_0}, \tau_0)\ \vec{v}\ rs & \text{marshall args. out}
\end{array}
$$

Figure 3.5: Syntax and semantics of CompCert x86 assembly (excerpts). Core states appear only in the operational semantics. Int-floatness types τ_0 are int, float, long, or single. Load frames ($\mathsf{mkLoadFrame}\ b_f\ \tau_0$) store a pointer b_f to a (copied) stack frame containing incoming arguments, as well as the return type τ_0 of the function that was initialized.

```
1  asm_initial_core ge v v⃗ : option C_asm ≜
2    case v of
3      | Vptr b_f i →
4          if i==0
5          then case find_funct_ptr ge b_f of
6                  | None → None
7                  | Some (Internal f) →
8                      let tys ≜ sig_args (funsig f) in
9                      if vals_have_types v⃗ tys
10                        && defined v⃗ && 4∗(2∗length v⃗) < max_unsigned
11                     then Some (Asm_CallStateIn b v⃗ tys)
12         else None
13     | _ → None
```

$$\frac{\text{argsLen } \vec{v}\ \vec{\tau} = \text{Some } z \qquad \text{alloc } m\ 0\ (4 * z) = (m_2, b_{stk}) \qquad \text{storeArgs } m_2\ b_{stk}\ \vec{v}\ \vec{\tau} = \text{Some } m' \qquad rs_0 = \text{empty } \textbf{with } \{\text{PC} := \text{Vptr } b_f\ 0\}\{\text{RA} := 0\}\{\text{ESP} := \text{Vptr } b_{stk}\ 0\}}{ge \vdash (\text{Asm_CallStateIn } b_f\ \vec{v}\ \vec{\tau}\ \tau), m \longmapsto (\text{State}_{\text{ASM}}\ rs_0\ (\text{mkLoadFrame } b_{stk}\ \tau)), m'} \ (\text{AsmInit})$$

Figure 3.6: x86 initialization. Top is x86 initial_core. Bottom is the operational rule that steps an Asm_CallStateIn to a running $\text{State}_{\text{ASM}}$.

line conservatively approximates value encoding: doubles and `long long` integers are encoded in CompCert x86 as two 32-bit words. The 4 specializes bytes-per-word in (32-bit) x86.

The AsmInit operational rule stores the $\vec{v}$ into a freshly allocated dummy stack frame. First, we calculate the size of the stack frame (argsLen). The types $\vec{\tau}$ are passed as a second argument to facilitate value encoding. Then we allocate a block of size $4 * z$ (because 4-byte words) and store the arguments (storeArgs) into the allocated block b_{stk}. rs_0 is the initial register state for the module. It sets PC to function pointer Vptr b_f 0, return address register RA to 0, and stack pointer register ESP to Vptr b_{stk} 0, a pointer to the allocated dummy stack frame. When we step from Asm_CallStateIn to the running state $\text{State}_{\text{ASM}}$, we record the block address b_{stk} of the dummy stack frame and the return type τ of the initial function as a *load frame* (mkLoadFrame). We use the load frame (a state component not present in original CompCert's x86) to express simulation invariants on the initial stack frame in the proof of the translation to x86.

Discussion. For x86 modules that share the same calling convention, the modeling step I describe above does not occur at runtime (nor does the compiler output any "marshalling" or copying code). The advantage of 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, is increased flexibility to model the interactions of modules in a wide variety of languages. By a variety of languages, I mean not only Clight and x86 but also x86 modules following different calling conventions, such as, *e.g.*, `cdecl` and Microsoft `fastcall` (for which additional code would have to be inserted at linktime).

On the other hand, it is not immediately obvious that the use of a "dummy" stack frame accurately models the interactions of linked modules running on a real machine, even when the modules share the same calling convention. Take, for example, the following C function:

```
int f(int* p, int x) {
  x = 0;
  *p = 1;
  return x;
}
```

The function `f` takes two arguments, an integer pointer `p` and an integer `x`. First, it assigns `x` the value `0`. Then it writes the value `1` to memory at location `p`, and returns `x`. If we compile and link `f` with the code

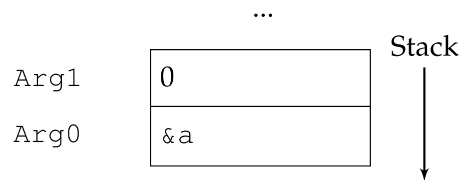
```
extern int f(int* p, int x);
int main(void) {
  int a;
  return f(&a, 0);
}
```

in a second translation unit, the resulting x86 program returns `0`, as expected. In fact, `f` should return `0` regardless of the values of `x` and `p`. For example, it is sound to rewrite this function, by simple constant propagation, to:

```
int f(int* p, int x) {
  *p = 1;
  return 0;
}
```

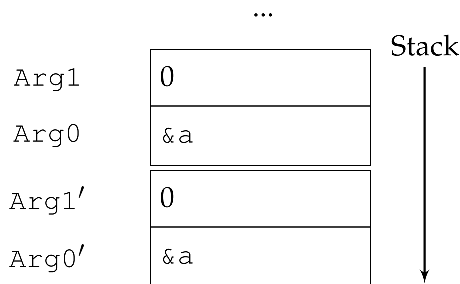
The question is: does our x86 semantics adequately model this behavior?

Graphically, at the point of the call to `f` in `main`, the stack looks like this:



The arguments have been pushed in right-to-left order. In the body of `f`, the write to `p` corresponds to a write to address `&a` (as loaded from address `Arg0`); the `x` returned is equal to the value at address `Arg1` (equal `0`).

The x86 semantic model produces a slightly different runtime state. In order to handle the call to `f`, it initializes a new x86 core which immediately allocates a fresh memory region in which to store copies of the function arguments. The resulting state is the following:



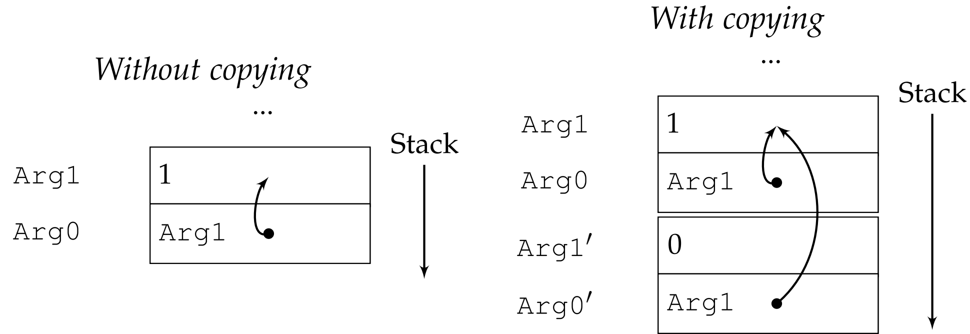
`Arg1`′ and `Arg0`′ are the addresses of the copied arguments. In this case, the copying does not change the behavior of the program (`x` and `p` have the same values as before). In general, when the arguments on the stack are not written to by the callee, the copying semantics simulates the no-copying semantics, meaning copying is a sound abstraction.

But it is also possible that `p` aliases the stack location at which the `x` argument to `f` is passed, leading `f` to inadvertently mutate its first parameter. For example, the following x86 assembly code, due to Tahina Ramananandro:

```
main:
  pushl 0
  pushl %esp
  call  f
  popl  %eax
  popl  %eax
  ret
```

causes `f` to mutate `x`, by first pushing the argument corresponding to parameter `x` (equal `0`) onto the stack, then pushing `%esp`—the address at which `0` was just stored—at the parameter position corresponding to `p`.[1]

Graphically, the situation is:



The stack layout without copying is on the left. The layout with copying is on the right. The pointer at address `Arg0` still points to the `Arg1` memory location, even after copying. The incoming parameter `x` resolves to the memory cell at location `Arg1′` with copying, and to `Arg1` without, resulting in two different return values depending on whether copying occurs.

In this case, the no-copying semantics produces the wrong result (as we mention above, `f` should return `0` regardless of its parameters). But where does the fault lie? Is it the compiler? `gcc -O0`, for example, uses the no-copying semantics on the left, resulting in return value `1` when linked with the assembly code implementation of `main` above. But to further confuse matters, `gcc -O2` turns on constant propagation, resulting in the "correct" return value `0`. Constant propagation should at least be sound for `f`. At the same time, the compiler should not be *forced* to copy in order to produce correct code, since copying is expensive.

The better answer is that the assembly code implementation of `main` is at fault. We just should not pass pointers to stack-allocated parameters when calling external functions from assembly contexts. Since `gcc` is the *de facto* standard C compiler, and it implicitly requires well-behaved contexts that do not alias parameter slots, then so should we.

One rationale here is that the outgoing parameters to an external call, while technically allocated in the caller's stack frame, are fresh locations in a sense "owned" by the callee. The caller should not be allowed to generate and pass pointers to these locations. More pragmatically, the compiler should not be *required* to copy in order to produce correct code, and to val-

---

[1] `pushl %esp` pushes the value of `%esp` as it existed before the stack pointer is decremented. [Int, Volume 2, Chapter 4 (Instruction Set Reference, N-Z), PUSH]

idate common optimizations such as constant propagation, in which case the program context—not the no-copying compiler (gcc)—is at fault.

When implementing modules in C, the issue does not arise quite as directly: The outgoing arguments of a C external function call are not laid out concretely, *i.e.*, on the stack, at the level of the C-programming-language abstraction. A C program cannot manipulate pointers to these (nonexistent) locations. When compiling from C to assembly, it is possible to show (though I have not proved this theorem in Coq) that the generated assembly code follows the no-pointers-to-parameters policy: Because the parameter stack slots introduced by register allocation are fresh memory locations (they are not allocated at all until the register allocation phase), there are no existing pointers in memory to these locations. Nor does the compiler introduce new pointers to these locations.

3.2.3 Gallina Semantics

The interaction semantics abstraction is suitable for expressing more than just traditional programming language semantics (*e.g.*, the one for Clight given in Section 3.2.1, or x86 in Section 3.2.2, or any of the other CompCert intermediate languages). Interaction semantics are general enough to model *arbitrary* relations over values and memories. This section demonstrates by constructing an interaction semantics of relations in Coq's specification language, Gallina (which is in turn suitable for expressing all of mathematics).

States in the Gallina semantics have type

$$\mathsf{gallinaState} \triangleq \mathsf{option}\ (\mathsf{gallinaRel} \times \mathsf{list}\ \mathcal{V})$$

where by gallinaRel I mean the type of relations

$$\mathsf{gallinaRel} \triangleq \forall(\vec{v} : \mathsf{list}\ \mathcal{V})(m_{pre} : M)(m_{post} : M).\ \mathsf{Prop}$$

that map a list of argument values $\vec{v}$, and pre- and post-memories m_{pre} and m_{post} to Coq propositions (type Prop). The option in the definition of gallinaState is used to indicate whether a Gallina semantics has been "executed" yet. If the option is None, then the Gallina semantics is halted.

$$\begin{array}{l} \mathsf{halted}\ (c : \mathsf{gallinaState}) : \mathsf{option}\ \mathcal{V} \triangleq \\ \quad \textbf{case}\ c\ \textbf{of} \\ \quad\quad \mid \mathsf{None} \rightarrow \mathsf{Some}\ 0 \\ \quad\quad \mid \mathsf{Some}\ _ \rightarrow \mathsf{None} \end{array}$$

Otherwise, we take a step, provided the following conditions hold.

$$\mathsf{corestep}\ (ge : G)\ (c : \mathsf{gallinaState})\ (m : M)\ (c' : \mathsf{gallinaState})\ (m' : M)\ \triangleq$$
$$(\exists R : \mathsf{gallinaRel}.\ \exists \vec{v}.\ c = \mathsf{Some}\ (R, \vec{v})\ \wedge\ R\ \vec{v}\ m\ m') \wedge c' = \mathsf{None}$$

In other words, a Gallina semantics steps from configuration c, m to new configuration c', m' provided that c is $\mathsf{Some}\ (R, \vec{v})$, for some relation R and initial arguments $\vec{v}$, the relation R is satisfied by $\vec{v}$, m, and m' ($R\ \vec{v}\ m\ m'$), and c' is halted ($c' = \mathsf{None}$). The key point is that the semantics is stuck whenever the relation R *does not* hold.

The definitions of the other required functions (at_external, after_external, initial_core) are straightforward. We simply say that the semantics is never blocked.

$$\mathsf{at_external}\ (c : \mathsf{gallinaState})\ :\ \mathsf{option}\ (\mathsf{ext_fun} \times \mathsf{list}\ \mathcal{V})\ \triangleq\ \mathsf{None}$$
$$\mathsf{afterExternal}\ (v_{opt} : \mathsf{option}\ \mathcal{V})\ (c : \mathsf{gallinaState})\ :\ \mathsf{option\ gallinaState}\ \triangleq\ \mathsf{None}$$

To construct an initial Gallina core, we parameterize by a relation R : gallinaRel and instantiate the initial core with this relation.

$$\mathsf{initial_core}\ (R : \mathsf{gallinaRel})\ (ge : G)\ (v : \mathcal{V})\ (\vec{v} : \mathsf{list}\ \mathcal{V})\ :\ \mathsf{option\ gallinaState}$$
$$\triangleq\ \mathsf{Some}\ (R, \vec{v})$$

There are many other ways in which such a semantics can be expressed. The relation R can enforce that $m = m'$, in which case we have a semantics of (unary) assertions on memory states. It is also possible to parameterize R not only by the arguments to initial_core, but also by the value v (typically of form $\mathsf{Vptr}\ b\ z$) that identifies the function (*e.g.*, at block address b) that this core was spawned to handle.

3.2.4 Trace Semantics

Gallina semantics (Section 3.2.3) demonstrated an interaction model of arbitrary relations on function arguments and memories. But the relations R which I employed in that section were history independent, in the sense that they did not predicate over the history, or trace, of external function call events produced by program executions.

In this section, I demonstrate an interaction semantics that *does* record interaction traces. This *trace semantics*, $\mathcal{T}$, differs from the semantics shown previously for Clight, x86, and Gallina in that it is an operator, or functor, over interaction semantics. As input, $\mathcal{T}$ takes an interaction semantics sem : Semantics $G\ C\ M$ and an axiomatization, *spec*, of the external functions that

may be called by sem; as output, it produces a new interaction semantics in which core states have been augmented to record the interaction trace.

Core states of the resulting semantics are products of

- C states,
- a program trace trs, and
- an external state (modeling the configuration of the environment).

In the definition of $\mathcal{T}$, we thread external states through the semantics. The external states themselves are only updated over external function calls.

At external call points, we use $spec$, which contains pre- and postconditions for each external function, to specify the pre- and postmemories of the external function being called, as well as the effect of that function on the external state (for example, an external call to `fopen` might change the state of a model of the filesystem to specify that a particular function is now open). The details are as follows.

Traces (of finite program executions) are defined as lists of events

$$trs \in \mathsf{trace} \triangleq \mathsf{list\ event}$$

where by event I mean a record of the input–output behavior of a call to an external function:

$$\begin{array}{l} \textbf{Record}\ \mathsf{event} : \mathsf{Type} \triangleq \\ \quad \{\ \mathsf{ef} : \mathsf{ident}; \\ \quad\ \ \mathsf{args} : \mathsf{list}\ \mathcal{V}; \\ \quad\ \ \mathsf{retv} : \mathsf{option}\ \mathcal{V}; \\ \quad\ \ \mathsf{pre} : M; \\ \quad\ \ \mathsf{post} : M\ \} \end{array}$$

The field ef is the identifier of the external function that was called. Values args are the arguments to ef; retv is the optional return value. pre and post are the memory states at the external function call and return points respectively.

We define the core states of trace semantics as the type

$$C_{\mathsf{tr}} \triangleq C \times \mathsf{list\ event} \times \Omega$$

where $\Omega : \mathsf{Type}$ is the additional parameter to $\mathcal{T}$ that gives the type of external states.

The step relation has two cases. The first is just a congruence rule, in which we step the core state c and memory m in trace semantics configuration (c, trs, ω) using the step relation of the underlying semantics:

$$\frac{ge \vdash c, m \longmapsto c', m'}{ge \vdash (c, trs, \omega), m \Longmapsto (c', trs, \omega), m'} \qquad (\textsc{TraceStep})$$

The external state and trace are left unchanged (internal steps are not observable). The second case handles external function calls.

$$\frac{\begin{array}{c}\mathsf{at_external}\ c = \mathsf{Some}\ (ef, \overrightarrow{v}) \qquad spec\ ef = (P, Q) \\ P(x, \omega, \overrightarrow{v}, m) \qquad Q(x, \omega', v_{opt}, m') \qquad \mathsf{after_external}\ v_{opt}\ c = \mathsf{Some}\ c'\end{array}}{ge \vdash (c, trs, \omega), m \Longmapsto (c', \mathsf{mkEvent}\ ef\ \overrightarrow{v}\ v_{opt}\ m\ m' :: trs, \omega'), m'} \text{(TRACEEXTERNAL)}$$

Core state c is at_external calling external function ef with arguments $\overrightarrow{v}$. The pre- and postconditions of ef as provided by $spec$ are P and Q respectively. P is satisfied by $\overrightarrow{v}$, the precall memory m, and external state ω; m', the return value v_{opt}, and new external state ω' satisfy the postcondition Q. Injecting the return value v_{opt} into c using after_external results in the new core state c'. From initial trace semantics state (c, trs, ω), we step to final state $(c', \mathsf{mkEvent}\ ef\ \overrightarrow{v}\ v_{opt}\ m\ m' :: trs, \omega')$ in which the new event $\mathsf{mkEvent}\ ef\ \overrightarrow{v}\ v_{opt}\ m\ m'$, signaling a call to external function ef, has been consed onto the head of the current trace.

The x parameter to both P and Q is a peculiarity of how we define function specifications (Chapter 7): for each function identifier ef, $spec$ provides a pre- and postcondition that are parameterized not only by the function arguments, return value, initial and final memories, *etc.*, but also by a value x of the (dependent) type spec_type ef, where

$$\mathsf{spec_type}\ :\ \mathsf{ident}\ \rightarrow\ \mathsf{Type}$$

is a function from external function identifiers to Type. Using this spec_type convention makes it possible to communicate information, in a function-specific way, between the precondition P and the postcondition Q. (For example, requiring $x = \overrightarrow{v}$ in P communicates the function arguments to the postcondition.) Binary postconditions, which parameterize Q by the initial state m and arguments $\overrightarrow{v}$ in addition to m' and v_{opt}, serve a similar purpose.

Trace semantics as presented above assumes that external functions do not *themselves* produce observable events, besides the single mkEvent consed onto the trace above to mark the external call. To lift this restriction, external function specifications can be augmented to include the function trace produced for given inputs, via a relation

$$\mathsf{traceOf}\ :\ \mathsf{ident}\ \rightarrow\ \mathsf{list}\ \mathcal{V}\ \rightarrow\ M\ \rightarrow\ \mathsf{trace}\ \rightarrow\ \mathsf{Prop}$$

that associates an external function name, the function arguments, and precall memory to a set of event traces. Assuming $trs' = \mathsf{traceOf}\ ef\ \overrightarrow{v}\ m$,

the trace that results in rule TraceExternal is then

$$\mathsf{mkEvent}\ ef\ \overrightarrow{v}\ v_{opt}\ m\ m' :: trs' +\!\!+\ trs$$

in which trs' has been interposed between the mkEvent and the tail trs.

Another limitation is that $\mathcal{T}$ does not deal gracefully with nontermination. The semantics faithfully models programs that may make an arbitrary *finite* number of external function calls, but not those that make infinitely many external function calls (reactive divergent programs). Nor does the trace model deal adequately with external functions that may diverge.

To deal with the first problem, it is sufficient to model traces coinductively, *i.e.* as streams instead of lists. To deal with the second problem, it's necessary to alter the specification of external functions to permit other behaviors (*e.g.*, divergence) in addition to termination in a poststate satisfying Q. Extended behaviors of external function must then be propagated through the trace semantics, *e.g.*, by adding to the core state type C_{tr} additional constructors for the additional behaviors and by updating the step relation $\longmapsto$ to do the propagation. These variations have not been done in Coq (yet) but would not be particularly difficult to implement.

3.3 Reach-Closed and Valid Semantics

In addition to the specialized interaction semantics I presented in the previous section, the results of later chapters (in particular, Chapter 6) will rely on two further specializations of the basic interface. The first specialization is to what I call *reach-closed semantics*. At a high level, a reach-closed semantics is one that writes only to memory locations *leaked* to it, *e.g.*, by following the reach-closure in memory of pointers returned to the module by external functions. A reach-closed semantics may also write to locations it allocated itself.

The second specialization I describe here but do not use until Chapter 6 is to *valid semantics*. A valid semantics is one that does not store invalid pointers into memory (in the sense of the val_valid predicate of Chapter 2). In the following I present the details.

3.3.1 Reach-Closed Semantics

Reach-closed semantics are defined by an invariant $\mathcal{R}$ on states c, memories m, and block sets B that satisfies the laws given in Figure 3.7.[2]

[2] File `compcomp/linking/rc_semantics.v`.

Reach-Closed Invariant

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

Reach-Closed Initial Core

$$\begin{array}{l} \mathsf{initial_core}\ ge\ v\ \vec{v} = \mathsf{Some}\ c \\ \implies \forall m.\ \mathcal{R}\ c\ m\ (\mathsf{REACH}\ m\ (\mathsf{blocksOf}\ \vec{v})) \end{array}$$

Reach-Closed Step

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

$$\begin{array}{l} \mathcal{R}\ c\ m\ B \land ge \vdash c, m \overset{E}{\longmapsto} c', m' \\ \implies (1)\ E \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \land \\ \qquad (2)\ \mathcal{R}\ c'\ m'\ (\mathsf{REACH}\ m'\ (\mathsf{freshblks}\ m\ m' \\ \qquad\qquad\qquad \cup\ \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B))) \end{array}$$

Reach-Closed At External

$$\mathcal{R}\ c\ m\ B \land \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \vec{v}) \implies \mathsf{defined}\ \vec{v}$$

Reach-Closed After External

$$\begin{array}{l} \mathbf{let}\ B' \triangleq \mathbf{case}\ v_{opt}\ \mathbf{of}\ \mathsf{None} \to B \\ \qquad\qquad\qquad \mid \mathsf{Some}\ v \to \mathsf{blocksOf}\ (v :: \mathsf{nil}) \cup B \\ \mathbf{in}\ \mathcal{R}\ c\ m\ B \\ \quad \land\ \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \vec{v}) \\ \quad \land\ \mathsf{after_external}\ v_{opt}\, c = \mathsf{Some}\ c' \\ \quad \implies \forall m'.\ \mathcal{R}\ c'\ m'\ B' \end{array}$$

Reach-Closed At External

$$\mathcal{R}\ c\ m\ B \land \mathsf{halted}\ c = \mathsf{Some}\ v_{ret} \implies \mathsf{defined}\ (v_{ret} :: nil)$$

Figure 3.7: Reach-Closed Semantics

Definition 2 (Reach-Closed Semantics (reach_closed)). *An interaction semantics is* reach-closed *iff there is an $\mathcal{R}$, specialized to the states and step relation of that semantics, that satisfies the laws of Figure 3.7.*

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 the step relation $\longmapsto$ and the functions after_external, initial_core, at_external, and halted.

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 of memory regions that records the memory 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).

The roots of a block set B and global environment ge are the union of the global blocks and B. The operative conditions of reach-closed semantics are those that characterize the reach-closed step relation (clauses 1 and 2). Clause (1) instruments the step relation of the underlying semantics with a restriction on the *effects* E produced by the step. The judgment

$$ge \vdash c, m \overset{E}{\longmapsto} c', m'$$

means: configuration c, m steps to c', m', writing to or freeing exactly the locations E.[3] Locations not contained in this set are guaranteed not to be modified. $E \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\, ge\, B)$ states that this set of modified locations E is a subset of the reach-closure (in m) of 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 (this set is 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 by the closure of the initial arguments, in the initial memory. at_external and halted (not shown) assert that the arguments to external calls and return values, respectively, are well-defined.

Reach closure is not an unrealistic proof obligation. One can show, for example, that all Clight programs satisfy the restrictions imposed in Fig-

[3]See `compcomp/core/effect_semantics.v` for the formal definition.

ure 3.7. In the following theorem, $\mathsf{CL}_{\mathsf{Sem}}$ is the Clight interaction semantics presented in Chapter 3.

Theorem 1 (Clight is Reach-Closed)**.** $\mathsf{CL}_{\mathsf{Sem}}$ *is* reach_closed*.*

Proof. The proof[4] of this (perhaps counterintuitive) theorem takes advantage of 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 [ISO11, 6.3.2.3].) Also perhaps counterintuitively, (safe) C pointer arithmetic does not violate the theorem. The REACH relation that appears Figure 3.7, and which was first defined in Chapter 2, is closed under intrablock pointer arithmetic.

The toplevel invariant $\mathcal{R}\ c\ m\ B$ holds when either c is the initial core and $B = \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ \vec{v})$, or c, m, and B satisfy the invariant cl_core_inv, defined by induction on the structure of c as follows.

$$
\begin{array}{l}
\mathcal{R}\ c\ m\ B : \mathsf{Prop} \triangleq \\
\quad (\exists\ v\ \vec{v}.\ B = \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ \vec{v}) \wedge \mathsf{initial_core}\ ge\ v\ \vec{v} = \mathsf{Some}\ c) \\
\quad \vee\ \mathsf{cl_core_inv}\ c\ m\ B
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cl_core_inv}\ c\ m\ B \triangleq \\
\quad \textbf{case}\ c\ \textbf{of} \\
\qquad \mid \mathsf{State}_{\mathsf{CL}}\ f\ s\ k\ e\ te \rightarrow \\
\qquad\qquad \mathsf{cl_state_inv}\ c\ m\ e\ te \\
\qquad\qquad \wedge\ \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \subseteq \mathsf{roots}\ ge\ B \\
\qquad\qquad \wedge\ \mathsf{cl_cont_inv}\ c\ k\ m \\
\qquad \mid \mathsf{Callstate}\ f\ \vec{v}\ k \rightarrow \\
\qquad\qquad \mathsf{blocksOf}\ \vec{v} \subseteq \mathsf{roots}\ ge\ B \\
\qquad\qquad \wedge\ \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \subseteq \mathsf{roots}\ ge\ B \\
\qquad\qquad \wedge\ \mathsf{cl_cont_inv}\ c\ k\ m \\
\qquad \mid \mathsf{Returnstate}\ v\ k \rightarrow \\
\qquad\qquad \mathsf{blocksOf}\ (v :: nil) \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \\
\qquad\qquad \wedge\ \mathsf{cl_cont_inv}\ c\ k\ m
\end{array}
$$

The key relation above is $\mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ B) \subseteq \mathsf{roots}\ ge\ B$, which asserts that roots $ge\ B$ (a set of blocks, as defined in Figure 3.7) is closed under reachability in memory m.

In the $\mathsf{State}_{\mathsf{CL}}$ case, the subsidiary relation cl_state_inv:

[4] File `compcomp/linking/safe_clight_rc.v`.

$$
\begin{array}{l}
\mathsf{cl_state_inv}\ c\ (m : \mathsf{mem})\ (B : \mathsf{set\ block})\ (e : \mathsf{var_env})\ (te : \mathsf{temp_env}) \triangleq \\
\quad \forall\ x\ b\ (ty : \mathsf{Ctypes.type}).\ e(x) = \mathsf{Some}\ (b, ty) \implies b \in \mathsf{roots}\ ge\ B \\
\wedge \forall\ x\ v.\ te(x) = \mathsf{Some}\ v \implies \mathsf{blocksOf}\ (v :: \mathsf{nil}) \subseteq \mathsf{roots}\ ge\ B
\end{array}
$$

asserts that

1. The memory region associated with every identifier x in the addressable local variable environment e is reachable; and
2. Every temporary identifier x in the temporaries environment te maps to a value whose blocks are contained in the current roots.

The cl_cont_inv invariant applies cl_state_inv recursively to the call stack:

$$
\begin{array}{l}
\mathsf{cl_cont_inv}\ c\ (k : \mathsf{cont})\ m \triangleq \\
\quad \textbf{case}\ k\ \textbf{of} \\
\qquad \mid \mathsf{Kstop} \to \mathsf{True} \\
\qquad \mid \mathsf{Kseq}\ s\ k' \to \mathsf{cl_cont_inv}\ c\ k'\ m \\
\qquad \mid \mathsf{Kloop}\ s_1\ s_2\ k' \to \mathsf{cl_cont_inv}\ c\ k'\ m \\
\qquad \mid \mathsf{Kswitch}\ k' \to \mathsf{cl_cont_inv}\ c\ k'\ m \\
\qquad \mid \mathsf{Kcall}\ id_{opt}\ f\ e\ te\ k' \to \mathsf{cl_state_inv}\ c\ m\ e\ te \wedge \mathsf{cl_cont_inv}\ c\ k'\ m
\end{array}
$$

It is tedious (but not difficult) to verify that $\mathcal{R}$ as defined above satisfies the laws of Figure 3.7. □

3.3.2 Valid Semantics

A semantics is *valid*[5] according to the following definition.

Definition 3 (Valid Semantics)**.** *A semantics is* valid *when there exists an invariant $\mathcal{I}$, specialized to the core states of the semantics, that satisfies the laws given in Figure 3.8.*

Informally, a valid semantics is one that never stores *invalid* pointers into memory. An invalid pointer is one that references a memory region that has not yet been allocated (freed memory regions are never invalid).

In the Compositional CompCert proofs, we establish that a semantics is valid by exhibiting an invariant $\mathcal{I}$ over core states of the semantics c and memories m with the properties given in Figure 3.8. As in Chapters 2 and 5, mem_valid m states that the memory m contains no invalid pointers. The other definitions, such as vals_valid m $\vec{v}$ (the values $\vec{v}$ are all valid with respect to m), are similar. In the clauses that mention the global environment ge, ge is assumed to contain only valid pointers as well.

[5] File `compcomp/core/nucular_semantics.v`.

Initially Valid

$$\begin{aligned}\forall v\ \vec{v}\ c.\ &\mathsf{initial_core}\ ge\ v\ \vec{v} = \mathsf{Some}\ c\\ &\wedge \mathsf{vals_valid}\ m\ \vec{v}\\ &\wedge \mathsf{mem_valid}\ m \implies \mathcal{I}(c, m)\end{aligned}$$

Corestep Valid

$$\begin{aligned}&\forall c\ m\ c'\ m'.\ \mathcal{I}(c, m)\\ &\quad\wedge\ ge \vdash c, m \longmapsto c', m' \implies \mathcal{I}(c', m')\end{aligned}$$

At External Valid

$$\begin{aligned}&\forall c\ m\ id_f\ \vec{v}.\ \mathcal{I}(c, m)\\ &\qquad\wedge\quad \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \vec{v})\\ &\quad\implies\ \mathsf{vals_valid}\ m\ \vec{v}\ \wedge\ \mathsf{mem_valid}\ m\end{aligned}$$

After External Valid

$$\begin{aligned}&\forall c\ m\ v\ c'\ m'.\ \mathcal{I}(c, m)\\ &\quad\wedge\ \mathsf{after_external}\ v\ c = \mathsf{Some}\ c'\\ &\quad\wedge\ \mathsf{val_valid}\ m'\ v\\ &\quad\wedge\ \mathsf{forward}\ m\ m'\\ &\quad\wedge\ \mathsf{mem_valid}\ m' \implies \mathcal{I}(c', m')\end{aligned}$$

Halted Valid

$$\begin{aligned}&\forall c\ m\ v.\ \mathsf{halted}\ c = \mathsf{Some}\ v\\ &\quad\implies\ \mathsf{val_valid}\ m\ v\ \wedge\ \mathsf{mem_valid}\ m\end{aligned}$$

Figure 3.8: Valid semantics maintain an internal invariant $\mathcal{I} : C \to \mathsf{mem} \to \mathsf{Prop}$ satisfying the five properties listed above.

The *Initially Valid* clause states that initializing c with valid arguments $\vec{v}$ in valid memory m results in a state satisfying $\mathcal{I}(c, m)$. One can think of this as the introduction rule for the invariant. *Corestep Valid* asserts that core steps of the underlying semantics preserve $\mathcal{I}$. *At External Valid* states that when c is at external, $\mathcal{I}(c, m)$ implies that the external function arguments $\vec{v}$ and memory m are valid. *After External Valid* asserts that it is possible to reestablish $\mathcal{I}$ after an external function call returns, under the assumptions given in the clause. *Halted Valid* states that $\mathcal{I}(c, m)$ implies that both the return value v and memory m are valid whenever c is halted.

It is possible to show that CompCert x86 assembly is a valid semantics.

Theorem 2 (CompCert x86 is Valid)**.** $\mathsf{Asm}_{\mathsf{Sem}}$ *is valid.*

Proof. $\mathsf{Asm}_{\mathsf{Sem}}$ is the x86 assembly semantics given in Chapter 3. To prove[6] that $\mathsf{Asm}_{\mathsf{Sem}}$ is valid, we must exhibit an $\mathcal{I}$ satisfying the laws in Figure 3.8. Let $\mathcal{I}$ equal:

$$
\begin{array}{l}
\mathcal{I}\ c\ m : \mathsf{Prop} \triangleq \\
\quad \mathsf{mem_valid}\ m\ \wedge \\
\quad \textbf{case}\ c\ \textbf{of} \\
\qquad \mid \mathsf{State}_{\mathsf{ASM}}\ rs\ lf \rightarrow \mathsf{regset_valid}\ m\ rs \wedge \mathsf{loadframe_valid}\ m\ lf \\
\qquad \mid \mathsf{Asm_CallstateIn}\ b_f\ \vec{v}\ \vec{\tau}\ \tau \rightarrow \\
\qquad\quad \mathsf{block_valid}\ m\ b_f \wedge \mathsf{vals_valid}\ m\ \vec{v} \\
\qquad \mid \mathsf{Asm_CallstateOut}\ _\ \vec{v}\ rs\ lf \rightarrow \\
\qquad\quad \mathsf{regset_valid}\ m\ rs \wedge \mathsf{loadframe_valid}\ m\ lf \wedge \mathsf{vals_valid}\ m\ \vec{v}
\end{array}
$$

with regset_valid and loadframe_valid defined as follows:

$$
\mathsf{regset_valid}\ m\ (rs : \mathsf{regset}) \triangleq \forall\ r.\ \mathsf{val_valid}\ rs(r)\ m
$$

$$
\begin{array}{l}
\mathsf{loadframe_valid}\ m\ (lf : \mathsf{load_frame}) \triangleq \\
\quad \textbf{case}\ lf\ \textbf{of}\ \mathsf{mkLoadFrame}\ b_{stk}\ \tau \rightarrow \mathsf{block_valid}\ m\ b_{stk}
\end{array}
$$

□

[6]File `compcomp/backend/Asm_nucular.v`.

Chapter

4 Language-Independent Linking

In Chapter 3, I introduced *interaction semantics* in order to interpret the behavior of isolated modules. Trace semantics $\mathcal{T}$ introduced the notion of an operator over interaction semantics. In this Chapter, I define a second operator

$$\mathcal{L}(\llbracket S_0 \rrbracket, \llbracket S_1 \rrbracket, \ldots, \llbracket S_{N-1} \rrbracket)$$

over interaction semantics that models the linked behavior of a *set* of interacting modules, as given by a multimodule program $P = S_0, S_1, \cdots, S_{N-1}$. Each S_i here ranges over (the syntax of) a program in a language such as Clight or x86 assembly (though linking semantics as I present it in this chapter is formally language-independent; one can start directly from semantics). $\llbracket S_i \rrbracket$ is the semantics of such a module, as defined by the Modsem record below.

4.1 Linking Semantics

As input, $\mathcal{L}$ takes N interaction semantics, each with (perhaps) a different global environment and core state type (for example, the modules may be programmed in perhaps different languages). The global environments of the modules must have equal domain (map the same set of addresses).[1]

[1]Throughout this thesis, $\mathcal{L}$-semantics are assumed to satisfy this property. See Section 7.3.1, Theorem 9 for further discussion, in particular of why this assumption is compatible with programs that declare different (but consistent) sets of global variables.

The output of $\mathcal{L}$ is a new interaction semantics

$$[\![P]\!] = \mathcal{L}([\![S_0]\!], [\![S_1]\!], \ldots, [\![S_{N-1}]\!])$$

that models the linked program execution 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.

```
Record Modsem : Type ≜
  mkModsem {
    F, V, C : Type;
    ge : Genv F V;
    sem : Semantics (Genv F C) C mem
  }
```

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 (and global identifiers to the addresses at which they are defined). 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).

The final component is sem, an interaction semantics. It defines the interface functions initial_core, at_external, after_external, and halted, as well as a step relation $\mathsf{ge} \vdash c, m \longmapsto c', m'$. 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.

```
Record Core (N : pos) (modules : I_N → Modsem) ≜
  mkCore {
    idx : I_N;
    core : C (modules idx)
  }
```

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 core field of the record (of dependent type C $(modules$ idx$)$ of core states of module idx) gives the current runtime state of this particular core.

The runtime state of a linked program is then:

```
Record LinkedState (N : pos) (modules : I_N → Modsem) ≜
  mkLinkedState {
    plt : ident → option I_N;
    stack : Stack (Core N modules)
  }
```

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 4.2 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.[2]

---

[2] This ge_c need not have the same type as the linked-program ge, or that of the global environments of other modules. Since each module may be implemented in a different language, each $ge_{\{c,d,\ldots\}}$ will in general map function addresses to function bodies of different types. We do require that the individual ges have equal domain (map the same set of global addresses). Section 7.3.1 of Chapter 7, in which

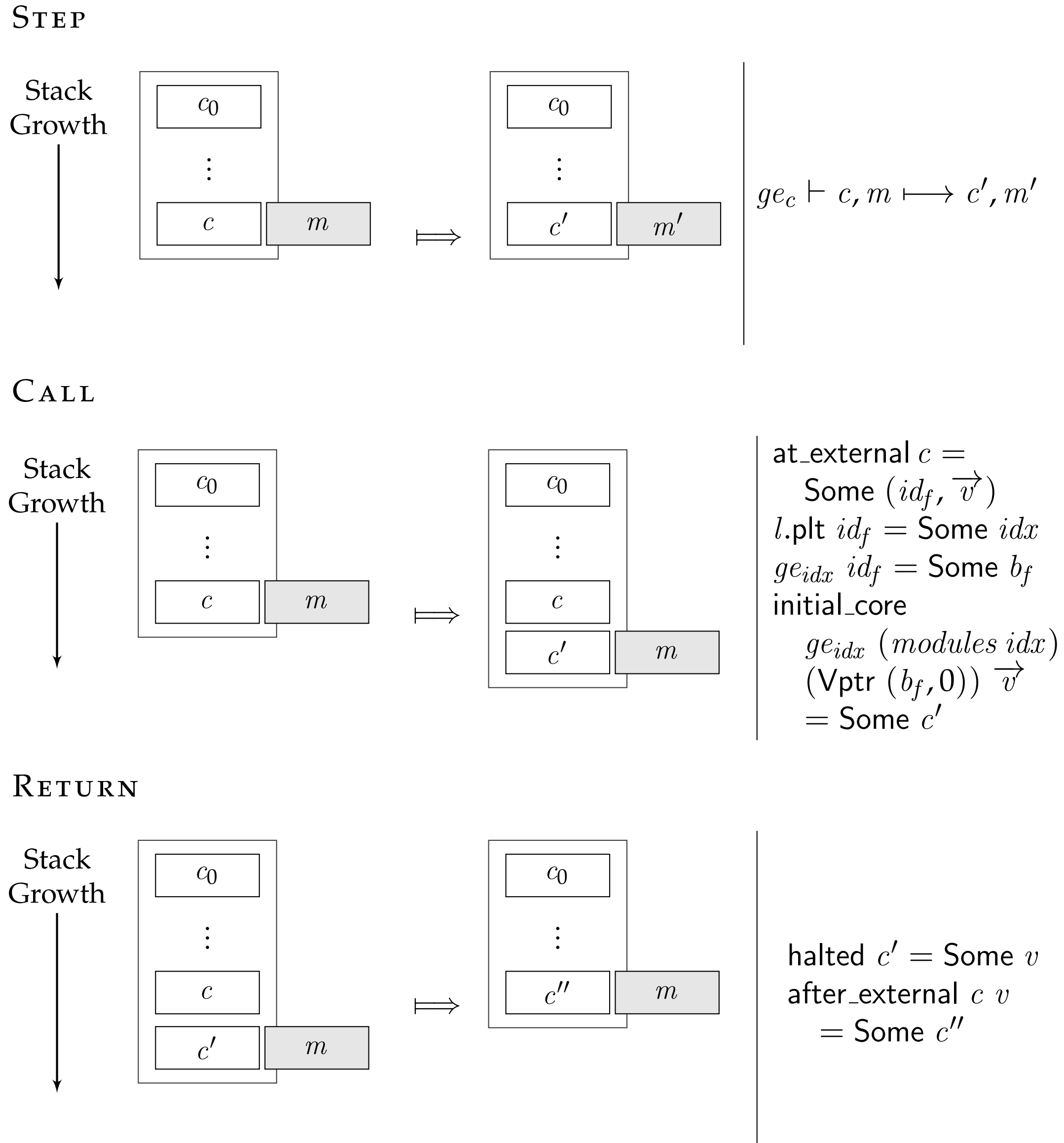


Figure 4.1: Cases of the linking corestep relation. Gray dashed boxes are "stacks-of-cores." Side conditions for each of the three rules are on the right.

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

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

$$\frac{\begin{array}{c} c = \text{peek } l.\text{stack} \qquad \text{at_external } c = \text{Some } (id_f, \overrightarrow{v}) \\ l.\text{plt } id_f = \text{Some } idx \qquad ge_{idx}\; id_f = \text{Some } b_f \\ \text{initial_core } ge_{idx} \; (modules \; idx) \; (\text{Vptr } (b_f, 0)) \; \overrightarrow{v} = \text{Some } c' \end{array}}{ge \vDash l, m \Longmapsto l \textbf{ with } \{\text{stack} := \text{push } c' \; l.\text{stack}\}, m} \; (\textsc{Call})$$

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

Figure 4.2: Corestep relation of Linking Semantics $\mathcal{L}$

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** $\{\text{stack} := \text{push } c' \; (\text{pop } l.\text{stack})\}$ 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 $c = \text{Some } (id_f, \overrightarrow{v})$) making a cross-module function call. In this case, we use the initial_core function of the module semantics that defines function id_f ($l.\text{plt } id_f = \text{Some } idx$) to initialize a new core state (in the global environment ge_{idx} associated with modules idx) to handle the function call:

$$\text{initial_core } ge_{idx} \; (modules \; idx) \; (\text{Vptr } (b_f, 0)) \; \overrightarrow{v} = \text{Some } c'$$

The core c' is then pushed onto the stack (l **with** $\{\text{stack} := \text{push } c' \; l.\text{stack}\}$) to become the new running core.

our safety proofs for linking semantics impose a stronger correspondence among the individual ges, tightens this requirement, while also presenting additional justification.

The RETURN rule models external function returns. In this case, the core state c is halted with return value v (l **with** $\{\mathsf{stack} := \mathsf{push}\ c'\ l.\mathsf{stack}\}$). To resume execution, we use the after_external function exposed by the caller's semantics $c' = \mathsf{peek}\ (\mathsf{pop}\ l.\mathsf{stack})$ to inject the return value v ($\mathsf{after_external}\ (\mathsf{Some}\ v)\ c' = \mathsf{Some}\ c''$). State c is then popped from the stack, and c' is updated to c'':

$$l\ \textbf{with}\ \{\mathsf{stack} := \mathsf{push}\ c''\ (\mathsf{pop}\ (\mathsf{pop}\ l.\mathsf{stack}))\}$$

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 stack.

By case analysis on $\Longmapsto$, we get that if all the *modules* semantics are deterministic, then so is the linked semantics $\mathcal{L}$.

Theorem 3 ($\mathcal{L}$-Determinism)**.**

$$(\forall M \in \{M_0, \ldots, M_{N-1}\}.\ \mathsf{deterministic}\ M) \implies \mathsf{deterministic}\ \mathcal{L}(M_0, \ldots, M_{N-1})$$

Proof. In Coq.[3] □

$\mathcal{L}$'s final ingredient is the definition of the interface functions: initial_core, at_external, after_external, and halted. In order to reduce the number of explicit case analyses and to aid comprehension, I present the code in monadic style.

A linking semantics is initialized (initial_core) by spawning a new core to handle the entry point function that was called.

```
1  initial_core (ge : G) (v : V) (v⃗ : list V) ≜
2    do { Vptr (b_f,0) ← v;
3         id_f ← invertSymbol ge b_f;
4         idx ← plt id_f;
5         c ← initial_core (modules idx) (Vptr (b_f,0)) v⃗;
6         return (Some (mkLinkedState plt (singletonStack c))) }
```

First, we look up the identifier id_f associated with function pointer b_f, if any (line 3). Then, we determine the index idx of the module that defines id_f

[3] Lemma `linking_det` in file `compcomp/linking/compcert_linking.v`.

Semantics G (LinkedState N $modules$) mem

```
initial_core (ge : G) (v : V) (v⃗ : list V) ≜
  do { Vptr (b_f,0) ← v;
       id_f ← invertSymbol ge b_f;
       idx ← plt id_f;
       c ← initial_core (modules idx) (Vptr (b_f,0)) v⃗;
       return (Some (mkLinkedState plt (singletonStack c))) }

at_external (l : LinkedState N modules) : option (F × list V) ≜
  let c ≜ peek l.stack in
  do { (id_f, v⃗) ← at_external c;
       case l.plt id_f of
         None → return (Some (id_f, v⃗))
         Some _ → return None }

after_external (v_opt : option V) (l : LinkedState N modules)
  : option (LinkedState N modules) ≜
  let c ≜ peek l.stack in
  do { c′ ← after_external v_opt c;
       return (Some l with {stack ≜ push c′ (pop l.stack)}) }

halted (l : LinkedState N modules) : option V ≜
  let c ≜ peek l.stack in
  do { v ← halted c;
       if size l.stack = 1 then return (Some v)
       else return None }

corestep ≜ ⟹
```

Figure 4.3: Interaction semantics of program linking. G is the type Genv unit unit. Corestep relation $\Longmapsto$ is as defined in Figure 4.2.

(line 4) and initialize module idx with function pointer Vptr $(b_f, 0)$ and arguments $\overrightarrow{v}$ (line 5), producing initial core c. We return the new LinkedState (mkLinkeState plt (singletonStack c)). The plt used in line 4 is a parameter to the definition, which is then stored in the linker state.[4] I also assume the existence of a global environment ge : Genv unit unit with the same address set as the ge_i of each module in *modules*. Such a ge can always be constructed (by mapping global addresses to unit) whenever the ge_i are consistent (the linking semantics is undefined otherwise).

A 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).

```
at_external (l : LinkedState N modules) : option (F × list V) ≜
  let c ≜ peek l.stack in
  do { (id_f, →v) ← at_external c;
       case l.plt id_f of
         None → return (Some (id_f, →v))
         Some _ → return None }
```

Line 3 peeks the top core on the stack. (The "callstack nonempty" invariant maintained by linking semantics ensures that such a core always exists.)

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', line 4).

```
after_external (v_opt : option V) (l : LinkedState N modules)
  : option (LinkedState N modules) ≜
  let c ≜ peek l.stack in
  do { c' ← after_external v_opt c;
       return (Some l with {stack ≜ push c' (pop l.stack)}) }
```

The LinkedState we return in this case is the same as l but with the topmost core c replaced by c'.

Finally, a linking semantics is halted when the stack contains a singleton halted core state (halted c and size l.stack $= 1$), *i.e.*, the topmost core is halted and has no return context.

---

[4]Storing the PLT in LinkedState makes it possible to model operations that *change* the PLT, like dynamic linking.

```
halted (l : LinkedState N modules) : option 𝒱 ≜
  let c ≜ peek l.stack in
  do { v ← halted c;
       if size l.stack = 1 then return (Some v)
       else return None }
```

4.2 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: $\mathcal{L}(C, [\![P]\!])$.

We can then define contextual equivalence of two open multimodule programs P_S and P_T as equitermination (halted in interaction semantics) in all contexts:

Definition 4 (Contextual Equivalence)**.**

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

$P\Downarrow$ is termination of program P. 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 particular predicate.

The above definition plays a bit fast and loose with the initial arguments and memory states in which the two programs P_S and P_T are executed; these details will be made precise when we present reach-closed contextual equivalence in Section 6.2.1.

4.3 Gallina Contexts

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. As an example Gallina context, consider the following Gallina semantics (cf. Section 3.2.3) that enforces the protocol:

> The character argument `c` to external function `putchar` satisfies the predicate `isLowerAlpha`: `'a'<= c && c <= 'z'`.

Recall that Gallina semantics are parameterized by a relation R of type $\forall(\overrightarrow{v} : \mathsf{list}\ \mathcal{V})(m_{pre} : M)(m_{post} : M).\ \mathsf{Prop}$. The corestep relation of a Gallina semantics is defined only when this R is satisfied:

$$\begin{array}{l}\mathsf{corestep}\ (ge : G)\ (c : \mathsf{gallinaState})\ (m : M)\ (c' : \mathsf{gallinaState})\ (m' : M)\ \triangleq \\ \quad (\exists R : \mathsf{gallinaRel}.\ \exists \overrightarrow{v}.\ c = \mathsf{Some}\ (R, \overrightarrow{v})\ \wedge\ R\ \overrightarrow{v}\ m\ m') \wedge c' = \mathsf{None}\end{array}$$

To "check" that `isLowerAlpha` holds at each call to `putchar`, we therefore define R as follows:

$$\begin{array}{l}R(\vec{v}, m_{pre}, m_{post}) \triangleq \\ \quad \exists c.\ \vec{v} = (c :: \mathtt{nil})\ \wedge\ \mathtt{isLowerAlpha}(c)\ \wedge\ m_{post} = m_{pre}\end{array}$$

The relation R is undefined (and the corestep relation of the Gallina context stuck) when either: (1) the arguments $\vec{v}$ to `putchar` are not of shape $(c :: \mathtt{nil})$, or (2) `c` does not satisfy `isLowerAlpha`.

4.4 Stateful Contexts

The `isLowerAlpha` protocol above is stateless, in the sense that it does not predicate over the history of interactions up to a certain point. It is also possible to define stateful Gallina contexts that *do* observe properties of the program trace.

For example, imagine we would like to enforce the protocol:

> In every execution of the program, a particular external function `f` is always called before a second external function `g`.

Perhaps `f` is an initialization routine, or provides access to a particular resource (*e.g.*, a file), while `g` accesses this resource.

Why should this specification be preserved by the compiler? Recall that, while the compiler is allowed to reorder calls to *internal* functions—as long as such reorderings are justified semantically—it is never allowed to reorder calls to external functions, since such calls are observable in interaction semantics. The order in which calls to external functions are made must be preserved.

We can construct a context that observes the order in which `f` and `g` are called as follows. First, extend the Gallina semantics of Section 3.2.3 to predicate over the global environment and function pointer of the called external function, in addition to the arguments and pre- and postmemories. That is, R is now a relation of type:

$$R : \forall(ge : G)(v_f : \mathcal{V})(\overrightarrow{v} : \mathsf{list}\ \mathcal{V})(m_{pre} : M)(m_{post} : M), \mathsf{Prop}$$

with $ge : G$ the global environment and v_f the additional value parameter. G is the type Genv unit unit (in which function bodies and variable type annotations are of type unit). The global environment will be used to look up the memory location associated with a ghost global variable, id_{hist}, storing context state (explained below; initialized at program startup to 0).

In order to case analyze in R on which external function is called at each interaction point, we update the initial_core function of the Gallina context to store the $v_f : \mathcal{V}$, in addition to R and $\vec{v}$:

$$\begin{aligned}&\text{initial_core } (R : \text{gallinaRel})(ge : \text{G})(v_f : \mathcal{V})(\vec{v} : \text{list } \mathcal{V}) : \text{option gallinaState}\\ &\quad \triangleq \text{Some } (R,\ v_f,\ \vec{v})\end{aligned}$$

with the type gallinaState appropriately extended to:

$$\text{gallinaState} \triangleq \text{option } (\text{gallinaRel} \times \mathcal{V} \times \text{list } \mathcal{V})$$

The corestep relation must be updated as well, to pass the v_f to R.

Now we define R as follows:

```
1   R(ge, v_f, v⃗, m_pre, m_post) ≜
2     do { l_hist ← ge(id_hist);
3          Vint n ← m_pre(l_hist);
4          Vptr (b_f,0) ← v_f;
5          id_f ← invertSymbol ge b_f;
6          if n == 0 then
7            if id_f == f then return m_post = m_pre[l_hist ↦ Vint 1]
8            else if id_f == g then return False
9                 else return m_post = m_pre
10         else return m_post = m_pre }
```

The code is presented in monadic style. Operations that fail do so by returning False. For example, the code on Line 2 desugars to:

$$\textbf{case } ge(id_{hist}) \textbf{ of } \text{None} \rightarrow \text{False} \mid \text{Some } l_{hist} \rightarrow \ldots$$

In lines 2 through 5, we look up the location l_{hist} associated with identifier id_{hist}, read the value Vint n at that address, case analyze the value v_f, returning a pointer Vptr $(b_f,0)$, and do a reverse lookup in the ge for the identifier id_f associated with block b_f. If any of these operations fails, R evaluates to False.

Line 6 branches on the value of n. When $n = 0$ (the initial state at program startup), we do an inner case analysis on the identifier id_f. In the expected case, in which $id_f = \text{f}$, we change state by asserting that the postmemory m_{post} equals the prememory with location l_{hist} updated to the

value Vint 1 ($m_{pre}[l_{hist} \mapsto \text{Vint } 1]$). If id_f equals g when $n = 0$, the program has violated the policy, in which case we return False. When id_f is neither f nor g, or n is no longer 0, we assert that the memory is unchanged.

Discussion. Whether this relation adequately models the specification presented informally above depends on a number of factors. The most important is the use of memory to record the integer n. Because the *entire* memory is communicated between modules at each intermodule function call, linking semantics does not directly prevent a function in one module from overwriting the location l_{hist} used to store the context state. It is possible, however, to prove that such overwrites do not occur, *e.g.*, by proving a global invariant on the value in memory at l_{hist}, or by showing that the program is safe when executed in an initial state that does not contain valid memory at location l_{hist} (the location can be initialized with read-only or even empty permission, for example, causing writes to get stuck). Because the context is "implemented" as a Gallina relation, it can update the value at l_{hist} regardless of the permissions, by bypassing the memory model interface. The other modules in the program must be implemented in languages (*e.g.*, Clight or x86) that respect the CompCert permission model.

There is a second way in which to model the f-before-g specification that bypasses the memory issues, at the cost of increased complexity in linking semantics. This is to directly record module-local state (nonaddressed file-scope static variables in C), in the form of a finite map

$$\mathsf{stateType} : I_N \rightarrow \mathsf{Type}$$

mapping module indices to the type of auxiliary state used by each module, and a second map

$$\mathsf{moduleStates} : \forall idx : I_N.\ \mathsf{stateType}\ idx$$

recording the current state associated with each module. As opposed to core states, which are initialized at each module entry, module-local states would persist across multiple dynamic invocations of each module. For example, the CALL rule of Figure 4.2 would be updated to:

$$\frac{\begin{array}{c}(c, \omega) = \mathsf{peek}\ l.\mathsf{stack} \qquad \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \overrightarrow{v}) \\ l.\mathsf{plt}\ id_f = \mathsf{Some}\ idx \qquad ge\ id_f = \mathsf{Some}\ b_f \\ \mathsf{initial_core}\ (modules\ idx)\ (\mathsf{Vptr}\ (b_f, 0))\ \overrightarrow{v} = \mathsf{Some}\ c' \\ \mathsf{moduleStates}\ idx = \omega'\end{array}}{ge\ \vDash\ l, m \ \Longrightarrow\ l\ \textbf{with}\ \{\mathsf{stack} := \mathsf{push}\ (c', \omega')\ l.\mathsf{stack}\}, m}\ (\text{CALL}')$$

to pair the module-local state ω' (as stored in moduleStates the previous time module idx returned to its caller) with the new core state c' initialized to handle function id_f. The map $modules : I_N \to \mathsf{Modsem}$ would also have to be updated, to contain interaction semantics that operate on pairs of core states and module-local states. Returning to a caller would involve an update to the moduleStates map, at the caller's module index.

Chapter

5 Compiler Correctness

Interaction and linking semantics (Chapters 3 and 4) provide the machinery with which to state compiler correctness (as cross-language contextual equivalence; *cf*. Section 4.2 for the basic definition). We do not yet have a way to *establish* such equivalences, however.

This chapter lays the groundwork. First, I present *whole-program simulations*, which recapitulate standard forward simulation proofs for closed programs, but adapted to the interaction semantics of Chapter 3. In Chapter 6 I will establish whole-program simulation for open programs by linking with a closing context, depending on the results here.

The second half of this chapter introduces *structured simulations*, a new equivalence method for open programs. Structured simulations are the compiler correctness relations we establish for each phase in Compositional CompCert (Chapter 8). As the results of the next chapter demonstrate, if each pair of modules S_i and T_i in a multimodule program is related by structured simulation, then the overall linked source and target programs $\mathcal{L}(S)$ and $\mathcal{L}(T)$ are contextually equivalent (Theorem 7 of Chapter 6).

In contrast with standard forward simulations and the logical simulation relations of previous work [BSDA14], the two distinguishing characteristics of structured simulations are their rely–guarantee and ownership disciplines.

Rely–Guarantee: Structured simulations impose a rely–guarantee discipline 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.

Ownership: Structured simulations enrich CompCert's standard 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 of **Ownership** above 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 of Ley-Wild and Nanevski's subjective concurrent separation logic [LWN13], though here it is applied to the *two-program* invariants used to prove compiler correctness.

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 $\vec{v}$, $A.f$'s compilation invariant must "give up exclusive control" of all the memory regions reachable from $\vec{v}$ (*i.e.*, following pointer chains rooted in $\vec{v}$). This condition represents the guarantee that, while later compilation stages of $A.f$ can still reorganize parts of the state reachable from $\vec{v}$ (*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.

Interestingly, this leakage protocol bears much in common with the *system-level semantics* of Ghica and Tzevelekos [GT12]. 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 I develop in Section 4.2. While Ghica and Tzevelekos were interested in modeling open C-like programs and their environments, not compiler correctness in this setting, I view the coincidence of our leakage conditions with their system-level semantics as evidence of the naturalness of our leakage protocol (Section 5.3.2).

5.1 Whole-Program Simulations

Simulation relations (or just *simulations*), and the related notion of bisimulations, were first used to prove program equivalence by Milner in the early 1970s (*cf.* [Mil71]). The idea is to define a relation R on the states of two infinite systems S and T—*e.g.*, two potentially nonterminating programs—such that $R(s,t)$ implies:

steps of the first system $s \mapsto s'$ are matched by steps of the second $t \mapsto t'$;

the relation $R(s',t')$ can be reestablished after each such pair of steps.

The first system S is generally called the *source* system in this thesis, while the second system T is the *target*, by analogy with the source and target languages of a compiler.

There are many variations on the basic idea. A *bisimulation* is a simulation R such that R^{-1} is also a simulation. A weak simulation (or bisimulation) is one in which the number of steps taken by the two systems is not one-to-one: for example, $R(s,t)$ and $s \mapsto s'$ may imply only that $t \mapsto^+ t'$ such that $R(s',t')$, *i.e.*, t takes one or more steps to t' in order to reestablish the relation.

A further useful generalization is *stuttering simulation*, in which multiple steps $s \mapsto^+ s'$ in the first system correspond to just a single step in the target system. Stuttering is typically modeled by defining a well-founded order $<$ on states s, s' (*i.e.*, such that there are no infinite descending chains $s > s' > s'' > \ldots$) for which $s' < s$ holds at each stuttering step. The well-founded order precludes infinite source sequences $s \mapsto s' \mapsto s'' \mapsto \ldots$ that do not cause the target to make at least some progress. This is useful for proving, *e.g.*, preservation of termination behavior from S to T. In Compositional CompCert, and in the rest of this thesis, I will generally employ simulations of the stuttering kind, since they present a nice balance between expressivity

(many kinds of compiler transformations can be proved correct in this way) and simplicity.

Figure 5.1 presents the clauses of what I call *whole-program simulations*,[1] adapted to the interaction semantics interface of Chapter 3. The simulations are "whole program" because they do not (yet) relate program modules that make external function calls (there are no cases for at_external and after_external). These will be dealt with in Section 5.3.2.

Whole-program simulations are nevertheless useful. For example, as we will see in Chapter 6, they can be used to relate the behaviors of source and target programs linked with a closing context, leading to a proof method for contextual equivalence. They are also simpler than the structured simulations I will present next, in Section 5.3.2, in the sense that it is easier to prove corollaries of whole-program simulation such as termination and safety preservation (Section 5.2).

There are three main clauses in Figure 5.1. The first, *Initial Core*, relates programs at initialization. The second, *Core Step*, relates programs over core steps. The third, *Halted Core*, relates programs at program exit. The main datatypes are:

- the source/target interaction semantics S and T;
- the source/target global environments ge_S and ge_T;
- function pointers, arguments, and return values v, $\vec{v_1}$, $\vec{v_2}$, v_1, and v_2;
- core states c, c', d, and d' of S and T respectively;
- CompCert memory injections f and f'; and
- a matching relation $\langle c, m\rangle \sim_f \langle d, tm\rangle$ on source and target core states and memories, indexed by the memory injection f (the R relation of the beginning of Section 5.1).

The *Initial Core* clause says: if initialization of source semantics S succeeds when passed function pointer v, arguments $\vec{v_1}$, in global environment ge_S, to produce a new core state c of the S semantics, then initializing T to execute the same function v, with related arguments $\vec{v_2}$ in related global environment ge_T, results in a state d such that $\langle c, m\rangle \sim_f \langle d, tm\rangle$. The arguments $\vec{v_1}$, $\vec{v_2}$ and the global environments ge_S, ge_T are related by the following auxiliary relations that parameterize every whole-program simulation structure:

globals_inv ge_S ge_T, which relates the global environments ge_S and ge_T. This is typically defined as dom ge_S = dom ge_T; and

init_inv f ge_S $\vec{v_1}$ m ge_T $\vec{v_2}$ tm, which specifies conditions that hold of the initial arguments and memories to a pair of programs.

[1] File `compcomp/core/closed_simulations.v`.

Initial Core

$$
\begin{array}{l}
(1)\ \mathsf{globals_inv}\ ge_S\ ge_T\ \wedge \\
(2)\ \mathsf{initial_core}\ S\ ge_S\ v\ \vec{v_1}\ = \mathsf{Some}\ c\ \wedge \\
(3)\ \mathsf{init_inv}\ f\ ge_S\ \vec{v_1}\ m\ ge_T\ \vec{v_2}\ tm \\
\implies \exists d.\ (4)\ \mathsf{initial_core}\ T\ ge_T\ v\ \vec{v_2}\ = \mathsf{Some}\ d\ \wedge \\
\qquad\qquad (5)\ \langle c, m\rangle \sim_f \langle d, tm\rangle
\end{array}
$$

Core Step

$$
\begin{array}{l}
(1)\ \mathsf{globals_inv}\ ge_S\ ge_T\ \wedge \\
(2)\ \langle c, m\rangle \sim_f \langle d, tm\rangle\ \wedge \\
(3)\ ge_S \vdash c, m \longmapsto c', m' \\
\implies \exists d'\ tm'\ f'. \\
\qquad (4)\ \langle c', m'\rangle \sim_{f'} \langle d', tm'\rangle \\
\qquad\quad \wedge\ (5)\ ge_T \vdash \langle d, tm\rangle \longmapsto^{+} \langle d', tm'\rangle\ \vee \\
\qquad\qquad (6)\ (ge_T \vdash \langle d, tm\rangle \longmapsto^{*} \langle d', tm'\rangle\ \wedge\ c' < c)
\end{array}
$$

Halted Core

$$
\begin{array}{l}
(1)\ \mathsf{globals_inv}\ ge_S\ ge_T\ \wedge \\
(2)\ \langle c, m\rangle \sim_f \langle d, tm\rangle\ \wedge \\
(3)\ \mathsf{halted}\ S\ c = \mathsf{Some}\ v_1 \\
\implies \exists v_2.\ (4)\ \mathsf{halted}\ T\ d = \mathsf{Some}\ v_2\ \wedge \\
\qquad\qquad (5)\ \mathsf{halt_inv}\ f\ ge_S\ v_1\ m\ ge_T\ v_2\ tm
\end{array}
$$

Figure 5.1: Whole-program simulations $S \leq T$

In Compositional CompCert, we specialize the init_inv relation to:

$$
\begin{array}{l}
\mathsf{init_inv}\ f\ ge_S\ \vec{v_1}\ m\ ge_T\ \vec{v_2}\ tm\ \triangleq \\
\quad \mathsf{inject}\ f\ m\ tm\ \wedge\ \mathsf{vals_inject}\ f\ \vec{v_1}\ \vec{v_2}\ \wedge\ \mathsf{preserves_globals}\ ge_S\ f\ \wedge \\
\quad \mathsf{mem_valid}\ tm\ \wedge\ \mathsf{globals_valid}\ ge_T\ tm\ \wedge\ \mathsf{vals_valid}\ \vec{v_2}\ tm
\end{array}
$$

The inject conditions state that the memories m, tm and initial arguments $\vec{v_1}$ and $\vec{v_2}$ are related by the CompCert injection f, as defined in Chapter 2. The preserves_globals clause states that f at least maps the blocks in dom ge_S, and is the identity mapping in this range (*i.e.*, global blocks are not removed by f, or translated to new blocks). The last three conditions (starting with mem_valid tm) state that

the target memory tm does not contain pointers to invalid blocks (recall that, as in Chapter 2, an invalid block is one that has not yet been allocated),

the target global blocks are all valid (globals_valid ge_T tm), and

the initial target arguments do not contain pointers to invalid blocks either (vals_valid $\vec{v_2}$ tm).

These conditions are easily satisfied when a target program is initialized in the memory containing, *e.g.*, just the globals for the program, with arguments that are either constants (such integers or floats) or pointers to global variables or functions. Indeed, there is no other static data to point to at program startup (I am not yet considering the core initializations that occur, *e.g.*, in linking semantics, at intermodule external calls).

The *Core Step* clause is a bit more involved: Assume

1. globals_inv holds of ge_S and ge_T,
2. $\langle c, m \rangle$ and $\langle d, tm \rangle$ are matching source and target configurations indexed by injection f, and
3. $\langle c, m \rangle$ steps to $\langle c', m' \rangle$.

Then there exist d', tm', and f' such that

4. $\langle d', tm' \rangle$ matches $\langle c', m' \rangle$, and either
5. $\langle d, tm \rangle$ takes one or more steps to $\langle d', tm' \rangle$, or
6. $\langle d, tm \rangle$ takes zero or more steps to $\langle d', tm' \rangle$ and c' descends the stuttering order ($c' < c$). (Alternatively, one could say that $d = d' \ \wedge \ tm = tm'$ in this case.)

The final clause, *Halted Core*, defines what it means for the halted state to be preserved: Assume

1. globals_inv holds of ge_S and ge_T,
2. $\langle c, m \rangle$ and $\langle d, tm \rangle$ are matching configurations, and
3. c is halted with return value v_1.

Then there exists a v_2 such that d is also halted, with return value v_2, and v_1 and v_2 (along with ge_S, ge_T, m, and tm) satisfy halt_inv, a predicate parameter chosen by the user who proved the simulation. In Compositional CompCert, we specialize this relation to:

$$\begin{array}{l}\textsf{halt_inv}\ f\ ge_S\ v_1\ m\ ge_T\ v_2\ tm \triangleq \\ \quad \textsf{inject}\ f\ m\ tm\ \wedge\ \textsf{val_inject}\ f\ v_1\ v_2\ \wedge\ \textsf{preserves_globals}\ ge_S\ f\end{array}$$

The memories m and tm, as well as the return values v_1 and v_2, are injected by f. In addition, f is a superset of the identity relation on dom ge_S (preserves_globals ge_S f).

5.2 Corollaries

There is nothing overly novel in the results presented in the previous section, beyond the adaptation of standard forward simulations to the interaction semantics interface of Chapter 3. Why focus on simulations for whole programs at all then?

The results of Chapter 6 will establish soundness for open programs P by linking with closing contexts C (those that do not themselves call external functions; they may call back into P). Section 6.2 constructs—from the open-program simulations on source P_S and target P_T to be presented in Section 5.3.2—a whole-program simulation between $\mathcal{L}(C, [\![P_S]\!])$ and $\mathcal{L}(C, [\![P_T]\!])$ (source and target open programs linked with closing context C). Preservation of behaviors in context C then follows (Theorem 7) from corollaries of whole-program simulation I present below.

5.2.1 Termination

We say that a program configuration c, m *terminates* in global environment ge if it steps, in zero or more steps, to a configuration c', m' for which c' is halted.

Definition 5 (terminates ge $\langle c, m\rangle$)**.**

$$\exists c'\ m'.\ ge \vdash \langle c, m\rangle \longmapsto^* \langle c', m'\rangle\ \wedge\ \exists v.\ \mathsf{halted}\ c' = \mathsf{Some}\ v$$

It is not hard to show that whole-program simulation $S \leq T$ implies preservation of termination from source program S to target T.[2] Recall that $S \leq T$ defines a matching relation $\langle c, m\rangle \sim_f \langle d, tm\rangle$, subject to the laws in Figure 5.1.

Corollary 1 (Termination Preservation)**.** *Assume $S \leq T$. For source configurations $\langle c, m\rangle$ and target configurations $\langle d, tm\rangle$, if $\langle c, m\rangle \sim_f \langle d, tm\rangle$ and* terminates ge_S $\langle c, m\rangle$*, then* terminates ge_T $\langle d, tm\rangle$*.*

[2]The definitions, theorems, and proofs in this subsection on termination can be found in file `compcomp/core/closed_simulations_lemmas.v`.

Proof. Assumption $\mathsf{terminates}\ ge_S\ \langle c, m\rangle$ unfolds to:

$$\exists c'\ m'.\ ge_S \vdash \langle c, m\rangle \longmapsto^* \langle c', m'\rangle\ \wedge\ \exists v.\ \mathsf{halted}\ c' = \mathsf{Some}\ v.$$

The corollary follows by induction on the multistep relation $ge \vdash \langle c, m\rangle \longmapsto^* \langle c', m'\rangle$, using the *Core Step* and *Halted Core* cases of the simulation $S \leq\ T$. □

The other direction of this corollary, reflection of termination behavior from T to S, only holds under additional assumptions. In particular, the target language L_T must be deterministic and source configuration $\langle c, m\rangle$ must be safe.

Corollary 2 (Termination Reflection). *For source configurations $\langle c, m\rangle$ and target configurations $\langle d, tm\rangle$, if $\langle c, m\rangle \sim_f \langle d, tm\rangle$, $\mathsf{terminates}\ ge_T\ \langle d, tm\rangle$, configuration $\langle c, m\rangle$ is safe in ge_S, and language L_T is deterministic, then $\mathsf{terminates}\ ge_S\ \langle c, m\rangle$.*

Proof. Assumption $\mathsf{terminates}\ ge_T\ \langle d, tm\rangle$ unfolds to:

$$\exists d'\ tm'\ n.\ ge_T \vdash \langle d, tm\rangle \longmapsto^n \langle d', tm'\rangle\ \wedge\ \exists v.\ \mathsf{halted}\ d' = \mathsf{Some}\ v.$$

The corollary follows by well-founded induction on n, using the usual less-than relation $<$ on the naturals, and relying on Lemmas 1 and 2 below. □

Lemma 1 (Split Multistep). *If*

- *L_T is deterministic,*
- *$ge_T \vdash \langle d, tm\rangle \longmapsto^n \langle d', tm'\rangle$,*
- *$ge_T \vdash \langle d, tm\rangle \longmapsto^m \langle d'', tm''\rangle$, and*
- *$n \leq m$*

then there exists q such that

- *$m = n + q$, and*
- *$ge_T \vdash \langle d', tm'\rangle \longmapsto^q \langle d'', tm''\rangle$*

Lemma 2 (Match Cases). *A state c is halted if there exists return value v such that $\mathsf{halted}\ c = \mathsf{Some}\ v$. If $\langle c, m\rangle \sim_f \langle d, tm\rangle$, then either*

- *$\mathsf{halted}\ c\ \wedge\ \mathsf{halted}\ d$; or*
- *$\exists f'\ c'\ m'.\ ge_S \vdash \langle c, m\rangle \longmapsto^+ \langle c', m'\rangle$ and either*
  - *$\langle c', m'\rangle \sim_{f'} \langle d, tm\rangle \wedge \mathsf{halted}\ c'\ \wedge\ \mathsf{halted}\ d$; or*
  - *$\exists d'\ tm'.\ ge_T \vdash \langle d, tm\rangle \longmapsto^+ \langle d', tm'\rangle\ \wedge\ \langle c', m'\rangle \sim_{f'} \langle d', tm'\rangle$.*

Lemma 1 asserts that, for deterministic languages, if we step in n steps for some n from $\langle d, tm\rangle$ to $\langle d', tm'\rangle$:

$$\langle d, tm \rangle \longmapsto \langle d', tm' \rangle$$

and we also step, in $m \geq n$ steps, from $\langle d, tm \rangle$ to $\langle d'', tm'' \rangle$:

$$\langle d, tm \rangle \longmapsto \langle d'', tm'' \rangle$$

then the second multistep relation, from $\langle d, tm \rangle$ to $\langle d'', tm'' \rangle$, can be decomposed into two multisteps intersecting at $\langle d', tm' \rangle$:

$$\langle d, tm \rangle \longmapsto \langle d', tm' \rangle \longmapsto \langle d'', tm'' \rangle$$

Lemma 2 is a useful elimination principle for $\langle c, m \rangle \sim_f \langle d, tm \rangle$ that facilitates reasoning by cases.

Putting everything together, we get that $S \leq T$ implies equitermination of matching states, under the additional assumptions required to prove Corollary 2.

Corollary 3 (Equitermination)**.** *For source configurations $\langle c, m \rangle$ and target configurations $\langle d, tm \rangle$, if $\langle c, m \rangle \sim_f \langle d, tm \rangle$, configuration $\langle c, m \rangle$ is safe in ge_S, and language L_T is deterministic, then* terminates ge_S $\langle c, m \rangle \iff$ terminates ge_T $\langle d, tm \rangle$.

Proof. By Corollaries 1 and 2. □

5.2.2 Safety

Simulations $S \leq T$ also imply safety preservation from source to target. When I say a configuration $\langle c, m \rangle$ of interaction semantics S is *safe*, in global environment ge, I mean, as usual, that the configuration will never get stuck (it may safely halt or infinite loop). In the context of whole-program simulations, the notion of safety we care about is that of closed programs (those that do not call external functions). I generalize safety to open programs in Chapter 6.

We say a configuration $\langle c, m \rangle$ is safe in global environment ge for n steps if it satisfies the following recursive predicate, expressed in Coq notation:

```
safeN n ge c m : Prop ≜
  case n of
    | 0 → True
    | n' + 1 →
      case halted c of
        | None → ∃c' m'. ge ⊢ c, m ⟼ c', m'  ∧  safeN n' ge c' m'
        | Some v → True
```

When n is 0, the predicate evaluates to True (we have given up interrogating the system). When n is greater than 0, there are two cases. If c is not halted (None), we assert that there exist a core state c' and memory m' such that (a) the systems steps from $\langle c, m \rangle$ to $\langle c', m' \rangle$ (we make progress) and (b) the new state $\langle c', m' \rangle$ is still safe, for at least $n-1$ steps. When c is halted (Some v), safeN reduces to True.

Definition 6 (Safety). *A configuration $\langle c, m \rangle$ is* safe *in global environment ge if it is* safeN *for all n.*

$$\mathsf{safe}\ ge\ c\ m\ \triangleq\ \forall n.\ \mathsf{safeN}\ n\ ge\ c\ m$$

If one views safeN as a finite approximation of safety (*i.e.*, for some number n steps), then safe is the intersection of all such approximations. This style of definition is quite useful when doing proofs by induction (on n), especially with respect to step-indexed semantics. Another, equivalent, definition is the more standard: a configuration $\langle c, m \rangle$ is safe if any configuration $\langle c', m' \rangle$ it can multistep to $ge \vdash \langle c, m \rangle \longmapsto^* \langle c', m' \rangle$ is either halted or can take at least one step.

Now we can state what it means for safety to be preserved by $S \leq T$:

Corollary 4 (Safety Preservation[3]). *If*

- $\langle c, m \rangle \sim_f \langle d, tm \rangle$,
- safe $ge_S\ c\ m$, *and*
- L_S *and* L_T *are deterministic*

then safe $ge_T\ d\ tm$.

Proof. safe $ge_T\ d\ tm$ unfolds to: $\forall n.$ safe $ge_T\ n\ d\ tm$. The corollary follows by induction on n, relying on Lemmas 2, 3, 4, and 5. □

Lemma 3 (Safe Downward). $\forall ge\ n\ n'\ c\ m.\ n' \leq n\ \wedge\ \mathsf{safeN}\ ge\ n\ c\ m\ \Longrightarrow\ \mathsf{safeN}\ ge\ n'\ c\ m.$

Lemma 4 (Safe Forward). $\forall ge\ n\ n'\ c\ m.\ \mathsf{deterministic}\ L_S\ \wedge\ ge \vdash \langle c, m \rangle \longmapsto^n \langle c', m' \rangle\ \wedge\ \mathsf{safeN}\ ge\ (n + n')\ c\ m\ \Longrightarrow\ \mathsf{safeN}\ ge\ n'\ c'\ m'.$

Lemma 5 (Safe Backward). $\forall ge\ n\ n'\ c\ m.\ ge \vdash \langle c, m \rangle \longmapsto^n \langle c', m' \rangle\ \wedge\ \mathsf{safeN}\ ge\ (n' - n)\ c'\ m'\ \Longrightarrow\ \mathsf{safeN}\ ge\ n'\ c\ m.$

Lemma 3 proves safeN is closed under approximation. Lemma 4 states that, for deterministic languages L_S, multistepping from a safe state results

[3]The Coq proof is in file `compcomp/core/closed_simulations_lemmas.v`.

in a safe state. Lemma 5 is the backward analog of Lemma 4: multistepping from $\langle c, m\rangle$ to a safe state $\langle c', m'\rangle$ implies that $\langle c, m\rangle$ is also safeN.

To prove termination preservation, we needed only that L_T was deterministic. Why do we need determinism of L_S here, in order to prove safety preservation? The answer has to do with how the definition of safety is formulated in Definition 6 and the auxiliary safeN. There, in safeN, we state only that there exist a c' and m' such that $ge \vdash c, m \longmapsto c', m'$ and safeN n' ge c' m'. This is sufficient for deterministic languages (since there can be only one such pair $\langle c', m'\rangle$) but it is not quite strong enough to capture safety in nondeterministic languages, for which we would like to know instead that (a) there exist such a c' and m', but also that (b) for *all such* c' and m' (*i.e.*, for which $ge \vdash c, m \longmapsto c', m'$), safeN n' ge c' m', along the lines of:

$$
\begin{array}{l}
\mathsf{safeN}'\ n\ ge\ c\ m : \mathsf{Prop} \triangleq \\
\quad \textbf{case}\ n\ \textbf{of} \\
\quad\quad \mid 0 \to \mathsf{True} \\
\quad\quad \mid n'+1 \to \\
\quad\quad\quad \textbf{case}\ \mathsf{halted}\ c\ \textbf{of} \\
\quad\quad\quad\quad \mid \mathsf{None} \to \exists c'\ m'.\ ge \vdash c, m \longmapsto c', m' \wedge \\
\quad\quad\quad\quad\quad\quad \forall c'\ m'.\ ge \vdash c, m \longmapsto c', m' \implies \mathsf{safeN}'\ n'\ ge\ c'\ m' \\
\quad\quad\quad\quad \mid \mathsf{Some}\ v \to \mathsf{True}
\end{array}
$$

$$\mathsf{safe}'\ ge\ c\ m \triangleq \forall n.\ \mathsf{safeN}'\ n\ ge\ c\ m$$

Under this second formulation of safety, we have that Corollary 4 holds even if S is nondeterministic. In addition, we can prove that—assuming S *is* deterministic—the first formulation of safety implies the second.

Lemma 6. *Assume L_S is deterministic. For all ge, c, m,* safe ge c m $\implies$ safe$'$ ge c m.

We use the first formulation, as given in Definition 6, because it matches the definition of safety used in the Verifiable C program logic [ADH+14]. This definition is sufficient in Compositional CompCert because all of the languages of the compiler, from Clight to CompCert x86 assembly, are deterministic. On the other hand, it could be useful in the future to generalize the definition of safeN used in the Verifiable C logic for nondeterminism.

5.2.3 Behavior Refinement

There is a third corollary of $S \leq T$, tying together both termination and safety preservation: Define the *behavior* of a configuration by the following inductive:

Inductive behavior : Type $\triangleq$ Termination | Divergence | Going_wrong.

A (closed) program (*i.e.*, one that does not call any external functions) either terminates, diverges, or goes wrong (gets stuck).

Behavior refinement says that if target configuration $\langle d, tm\rangle$ exhibits some behavior tb, then matching source configurations $\langle c, m\rangle$ will exhibit behaviors b that are *refined by* tb ($b \geq_{beh} tb$).

Corollary 5 (Behavior Refinement)**.** *If*

- $\langle c, m\rangle \sim_f \langle d, tm\rangle$,
- $\langle d, tm\rangle$ *has behavior* tb *in environment* ge_T, *and*
- L_T *is deterministic*

then there exists behavior b *such that*

- $\langle c, m\rangle$ *exhibits behavior* b *in environment* ge_S, *and*
- $b \geq_{beh} tb$.

Proof. Proved in Coq.[4] □

Refinement of behaviors $b \geq_{beh} tb$ is defined as in the following table:

Source behavior...		is refined by target behavior...
Termination	$\geq_{beh}$	Termination
Divergence	$\geq_{beh}$	Divergence
Going_wrong	$\geq_{beh}$	Termination, Divergence, Going_wrong

If the source configuration terminates or diverges, then so must the target configuration. If the source program goes wrong (gets stuck), then the target may either terminate, diverge, or itself go wrong.

The relation that ascribes behaviors to programs is given by:

In env. ge, $\langle c, m\rangle$ has behavior...	iff...
Termination	terminates ge $\langle c, m\rangle$
Divergence	forever_steps_or_halted ge $\langle c, m\rangle$ $\wedge$ $\neg$terminates ge $\langle c, m\rangle$
Going_wrong	$\neg$safe ge $\langle c, m\rangle$

To handle nondeterministic languages, replace safe above with the alternate definition safe′. For deterministic languages, forever_steps_or_halted is equivalent to safe.

If we know, in addition, that the source configuration $\langle c, m\rangle$ is safe, then we get an even stronger result, namely:

[4] File `compcomp/core/closed_simulations_lemmas.v`.

Corollary 6 (Behavioral Equivalence)**.** *If*

- $\langle c, m\rangle \sim_f \langle d, tm\rangle$,
- L_T *is deterministic, and*
- $\langle c, m\rangle$ ***is safe in*** ge_S

then for all behaviors b*,* $\langle c, m\rangle$ *has behavior* b *in environment* ge_S *iff* $\langle d, tm\rangle$ *has behavior* b *in environment* ge_T*.*

Proof. ($\Longrightarrow$) By Corollaries 3 and 4. ($\Longleftarrow$) By Corollary 5. □

5.3 Open Program Simulations

In this section, I extend the whole-program simulations $S \leq T$ of Section 5.1 to *open* programs (*i.e.*, those that may call external functions defined in other translation units), in the form of *structured simulations* $S \preceq T$. Structured simulations are an extension of the related *logical simulation relations (LSRs)*, which were first described in [BSDA14]. I first give background on LSRs, as motivation, then present structured simulations.

5.3.1 Logical Simulation Relations

Logical simulation relations (LSRs) established compiler correctness by showing that compilation *preserved the protocol structure* of the interaction semantics of Chapter 3. They used CompCert's original match relations $\sim_f$, with memory injections f, to relate source and target states.

What does "preserving the protocol structure of interaction semantics" mean? For internal execution steps, that LSRs followed CompCert's standard forward simulation proofs: internal steps of the source semantics were matched by (one or more) internal steps of the target semantics, up to stuttering of the source. For external calls, LSRs departed from CompCert by asserting that related modules:

- called the same function with related arguments; and
- were *receptive*, at the point at which external function calls returned, to any related values and memories the environment might provide. By receptive, I mean the equivalence of related modules could be re-established at the point of external function call return assuming related return values and memories.

This last condition was subject to a few constraints on how memory could evolve over the external calls, the two most crucial of which were:

1. in the source execution, external calls did not modify any memory region the compiler wished to remove;[5] and that
2. in the target execution, external calls did not modify target memory locations that did not correspond to readable locations 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, structured simulations make three adjustments to the LSR framework.

To index the match relation $\sim$, they use *structured injections* μ instead of CompCert's original injections f. The additional structure in μ maintains the block-level ownership information necessary to tell a callee's blocks (or other blocks associated with the environment) apart from blocks associated with the caller.

Structured simulations 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.

Structured simulations impose a restriction axiom on $\sim$ that ensures that 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.

5.3.2 Structured Simulations

Recall from Chapter 2 that, in CompCert, memory is allocated in regions, or *blocks*. Within each block, memory bytes are addressed using integer offsets

---

[5]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 I say this memory region is removed by the compiler.

(pointer arithmetic is allowed only within blocks). CompCert's memory injections

$$f : \mathsf{block} \to \mathsf{option}\ \ (\mathsf{block} \times \mathcal{Z})$$

relate source and target memories. For example, the memory injection that maps b to Some (b', z) associates source address $(b, 8)$ with target address $(b', 8 + z)$.

Structured injections μ (Figure 5.2) strengthen CompCert's memory injection relations with additional ownership structure.[6] They have four components: Two *ownership* functions $\mathsf{own}_S, \mathsf{own}_T : \mathsf{block} \to \mathsf{Ownership}$, 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: f_{us} and $\mathsf{f}_{\mathsf{them}}$. f_{us} records the source–target mapping of blocks that were allocated by the current module; $\mathsf{f}_{\mathsf{them}}$ maps external blocks (those allocated by other modules).

The Ownership modes are:

Mode...	applies to...
Priv	blocks (memory regions) allocated by the module being compiled but which haven't been leaked to the environment
Pub	allocated blocks that *have* been leaked at a previous interaction point
Frgn	foreign blocks leaked into μ at external calls
Invis	blocks that have been allocated (by another module) but not leaked in
None	blocks that may not yet have been allocated.

A block is (locally) owned by μ 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 μ are those in the set $\mathsf{vis}_S \triangleq \mathsf{owned}_S \cup \mathsf{shared}_S$ (and likewise for vis_T). I 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.

[6] File `compcomp/core/structured_injections.v`.

Structured Injections

$$\mathsf{Ownership} \triangleq \mathsf{Priv} \mid \mathsf{Pub} \mid \mathsf{Frgn} \mid \mathsf{Invis} \mid \mathsf{None}$$

$$\mu \in \mathsf{StructuredInjection} : \mathsf{Type} \triangleq \left\{ \begin{array}{rcl} \mathsf{own}_S & : & \mathsf{block} \to \mathsf{Ownership} \\ \mathsf{own}_T & : & \mathsf{block} \to \mathsf{Ownership} \\ \mathsf{f_{us}} & : & \mathsf{block} \to \mathsf{option}\ \ (\mathsf{block} \times \mathcal{Z}) \\ \mathsf{f_{them}} & : & \mathsf{block} \to \mathsf{option}\ \ (\mathsf{block} \times \mathcal{Z}) \end{array} \right.$$

$$\begin{array}{rcl} \mathsf{public}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Pub}\}\}, i \in \{S, T\} \\ \mathsf{private}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Priv}\}\}, i \in \{S, T\} \\ \mathsf{foreign}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Frgn}\}\}, i \in \{S, T\} \\ \mathsf{invis}_i & \triangleq & \{b \mid \mathsf{own}_i(b) \in \{\mathsf{Invis}\}\}, i \in \{S, T\} \\ \mathsf{owned}_i & \triangleq & \mathsf{public}_i \cup \mathsf{private}_i, i \in \{S, T\} \\ \mathsf{shared}_i & \triangleq & \mathsf{public}_i \cup \mathsf{foreign}_i, i \in \{S, T\} \\ \mathsf{extern}_i & \triangleq & \mathsf{foreign}_i \cup \mathsf{invis}_i, i \in \{S, T\} \\ \mathsf{vis}_i & \triangleq & \mathsf{owned}_i \cup \mathsf{foreign}_i, i \in \{S, T\} \end{array}$$

Structured Injection Axioms

$$\begin{array}{l} \mathsf{owned}_i \ \cap\ \mathsf{extern}_i = \emptyset, i \in \{S, T\} \\ \forall b_1\ b_2\ z.\ \mathsf{f_{us}}\, b_1 = \mathsf{Some}\ (b_2, z) \implies b_1 \in \mathsf{owned}_S \ \wedge\ b_2 \in \mathsf{owned}_T \\ \forall b_1\ b_2\ z.\ \mathsf{f_{them}}\, b_1 = \mathsf{Some}\ (b_2, z) \implies b_1 \in \mathsf{extern}_S \ \wedge\ b_2 \in \mathsf{extern}_T \\ \forall b_1.\ b_1 \in \mathsf{public}_S \implies \exists b_2\ z.\ \mathsf{f_{us}}\ b_1 = \mathsf{Some}\ (b_2, z) \ \wedge\ b_2 \in \mathsf{public}_T \\ \forall b_1.\ b_1 \in \mathsf{foreign}_S \implies \exists b_2\ z.\ \mathsf{f_{them}}\ b_1 = \mathsf{Some}\ (b_2, z) \ \wedge\ b_2 \in \mathsf{foreign}_T \\ \quad \mathsf{public}_T \ \subseteq\ \mathsf{owned}_T \\ \quad \mathsf{foreign}_T \ \subseteq\ \mathsf{extern}_T \end{array}$$

Injection Restriction

$$\begin{array}{rcl} f \downharpoonleft_X & \triangleq & \lambda b.\ \textbf{if}\ b \in X\ \textbf{then}\ f\ b\ \textbf{else}\ \mathsf{None} \\ \mu \downharpoonleft_X & \triangleq & \mu\ \textbf{with}\ \{\mathsf{f_{us}} := \mathsf{f_{us}} \downharpoonleft_X\}\{\mathsf{f_{them}} := \mathsf{f_{them}} \downharpoonleft_X\} \end{array}$$

Figure 5.2: Structured Injections and the axioms they satisfy. In Coq, the structured injection axioms are imposed via a dependent record type.

Complementing the data in Figure 5.2 are laws[7] that ensure proper interaction of ownership, leakage, and compilation. These laws, given in the lower half of Figure 5.2, enforce that f_{us} and $\mathsf{f}_{\mathsf{them}}$ (1) operate exclusively on blocks of appropriate ownership (*i.e.* f_{us} only maps owned blocks, to owned blocks, and similarly for $\mathsf{f}_{\mathsf{them}}$ and external blocks); and (2) are total on their portion of shared blocks: f_{us} must map all Pub_S blocks, and must map them to Pub_T blocks, and similarly for $\mathsf{f}_{\mathsf{them}}$ 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, the structured injections are adjusted 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). As an additional invariant, structured simulations maintain that the source visible set vis_S is always closed under pointer dereferencing and pointer arithmetic.

Structured Simulation Details. The structured simulation clauses for initial_core, at_external, and halted are given in Figure 5.3.[8]

The *Initial Core* clause states the conditions under which core initialization tracks from source to target. For any source memory m, target memory tm, and CompCert-style memory injection f, and for block sets dom_S and dom_T, if c is the core initialized by initial_core to handle function pointer v with arguments $\vec{v_S}$, then there exists a target core d that results from initializing the target semantics at v with the related arguments $\vec{v_t}$. The other hypotheses of this clause, such as those marked by $(*)$, further constrain f, dom_S, and dom_T. For example, $\mathsf{REACH}\ tm\ (\mathsf{globalsOf}\ ge_T \cup \mathsf{blocksOf}\ \vec{v_t}) \subseteq \mathsf{dom}_T$ states that the set of blocks reachable from target globals ge_T and $\vec{v_t}$ is contained in dom_T. The hypotheses marked (†) are required to satisfy a technical invariant of structured simulations, that blocks mentioned by the current structured injection were allocated at some point in the past (they may have been freed in the meantime).

The function μ_{init} constructs a structured injection from components:

[7]File `compcomp/core/structured_injections.v`.

[8]File `compcomp/core/simulations.v`.

Structured Simulations $S \preceq T$

Initial Core

$$
\begin{array}{l}
\forall f\ m\ tm\ v\ \vec{v_s}\ \vec{v_t}\ \mathsf{dom}_S\ \mathsf{dom}_T. \\
\quad \mathsf{initial_core}\ S\ ge_S\ v\ \vec{v_s} = \mathsf{Some}\ c\ \wedge \\
\quad \mathsf{inject}\ f\ m\ tm\ \wedge \\
\quad \mathsf{vals_inject}\ f\ \vec{v_s}\ \vec{v_t}\ \wedge \\
\quad (*)\ \mathsf{preserves_globals}\ ge_S\ f\ \wedge \\
\quad (*)\ (\forall b_1 b_2 z.\ f\ b_1 = \mathsf{Some}\ (b_2, z) \implies b_1 \in \mathsf{dom}_S \wedge b_2 \in \mathsf{dom}_T) \wedge \\
\quad (*)\ \mathsf{REACH}\ tm\ (\mathsf{globalsOf}\ ge_T\ \cup\ \mathsf{blocksOf}\ \vec{v_t})\ \subseteq\ \mathsf{dom}_T\ \wedge \\
\quad (\dagger)\ \mathsf{dom}_S \subseteq \mathsf{validBlocks}\ m\ \wedge \\
\quad (\dagger)\ \mathsf{dom}_T \subseteq \mathsf{validBlocks}\ tm \\
\quad \implies \exists \mu\ d.\ \mathsf{initial_core}\ T\ ge_T\ v\ \vec{v_t} = \mathsf{Some}\ d \\
\qquad\quad \wedge\ \langle c, m \rangle \sim_\mu \langle d, tm \rangle \\
\qquad\quad \wedge\ \mu = \mu_{\mathsf{init}}(\mathsf{dom}_S, \mathsf{dom}_T, \mathsf{REACH}\ m\ (\mathsf{globalsOf}\ ge_S\ \cup\ \mathsf{blocksOf}\ \vec{v_s}), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{REACH}\ tm\ (\mathsf{globalsOf}\ ge_T\ \cup\ \mathsf{blocksOf}\ \vec{v_t}), f)
\end{array}
$$

At External

$$
\begin{array}{l}
\langle c, m \rangle \sim_\mu \langle d, tm \rangle\ \wedge\ \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \vec{v_s}) \\
\quad \implies \exists \vec{v_t}.\ \mathsf{inject}\ \mu\ m\ tm \\
\qquad\quad \wedge\ \mathsf{vals_inject}\ \mu\lfloor_{\mathsf{vis}_S\ \mu}\ \vec{v_s}\ \vec{v_t} \\
\qquad\quad \wedge\ \mathsf{at_external}\ d = \mathsf{Some}\ (id_f, \vec{v_t}) \\
\qquad\quad \wedge\ \langle c, m \rangle \sim_{\mathsf{leak_out}(\mu, \vec{v_s}, \vec{v_t}, m, tm)} \langle d, tm \rangle \\
\qquad\quad \wedge\ \mathsf{vals_inject}\ \mu\lfloor_{\mathsf{shared}_S\ \mu}\ m\ tm
\end{array}
$$

Halted Core

$$
\begin{array}{l}
\langle c, m \rangle \sim_\mu \langle d, tm \rangle \wedge\ \mathsf{halted}\ c = \mathsf{Some}\ v_s \\
\quad \implies \exists v_t.\ \mathsf{inject}\ \mu\ m\ tm\ \wedge\ \mathsf{val_inject}\ \mu\lfloor_{\mathsf{vis}_S\ \mu}\ v_s\ v_t \\
\qquad\quad \wedge\ \mathsf{halted}\ d = \mathsf{Some}\ v_t
\end{array}
$$

Figure 5.3: Structured Simulations: Initial Core, At External, and Halted Core clauses

$$
\begin{array}{l}
\mu_{\mathsf{init}}\ \mathsf{dom}_S\ \mathsf{dom}_T\ \mathsf{frgn}_S\ \mathsf{frgn}_T\ f\ \ :\ \mathsf{StructuredInjection} \triangleq \\
\quad \mathsf{mkStructuredInjection}\ \{ \\
\quad\quad \mathsf{own}_i \triangleq \lambda b. \\
\quad\quad\quad \textbf{if}\ b \in \mathsf{frgn}_i\ \textbf{then}\ \mathsf{Frgn} \\
\quad\quad\quad \textbf{else if}\ b \in \mathsf{dom}_i\ \textbf{then}\ \mathsf{Invis}\ \textbf{else}\ \mathsf{None}; \\
\quad\quad \mathsf{f_{us}} \triangleq \lambda b.\ \mathsf{None}; \\
\quad\quad \mathsf{f_{them}} \triangleq f \\
\quad \}
\end{array}
$$

The $\mathsf{owned}_{\{S,T\}}$ functions are constructed from the sets dom_S,dom_T,frgn_S, and frgn_T. The "us" injection $\mathsf{f_{us}}$ initially maps no blocks, because a freshly initialized core has not yet allocated any blocks. The $\mathsf{f_{them}}$ injection is set equal to the argument injection f. The hypotheses marked $(*)$ in the figure ensure that the structured injection we build using μ_{init} satisfies the axioms of Figure 5.2.

The *At External* clause asserts that at_external can be tracked from source to target: source states calling an external function only match target states calling the same external function, with related arguments. The extra match clause in the conclusion of the rule,

$$\langle c, m\rangle \sim_{\mathsf{leak_out}(\mu, \vec{v_s}, \vec{v_t}, m, tm)} \langle d, tm\rangle$$

enforces that, at external function call points, the match relation $\sim$ is closed under the "leak out" operation defined later in this chapter. In other words, $\sim$ is not invalidated if we mark as public, at external call points, all those blocks reachable from the arguments $\vec{v_s}$ and $\vec{v_t}$. I explain this property in more detail in the next section, when I introduce the rule for external function call returns.

The final clause, *Halted Core*, asserts preservation of termination behavior. It says that halted source states only match target states that are also halted. In addition, we get that at termination, the source and target memories m and tm are related by μ, and that the return values v_s and v_t are related by the restriction of μ to visible source blocks $\mathsf{vis}_S\,\mu$.

Internal and External Steps. Figure 5.4 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*

Structured Simulations $S \preceq T$ (cont'd)

Internal Steps

$$\langle c, m\rangle \sim_\mu \langle d, tm\rangle \wedge ge_S \vdash c, m \xrightarrow{E_S} c', m' \implies$$
$$\exists d'\ tm'\ \mu'.$$
$$(1)\ \mu \sqsubseteq_{\mathsf{us}} \mu' \wedge$$
$$(2)\ \mathsf{separated}\ \mu\ \mu'\ m\ tm \wedge$$
$$(3)\ \mathsf{locally_allocated}\ \mu\ \mu'\ m\ tm\ m'\ tm' \wedge$$
$$(4)\ \langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle \wedge$$
$$(5)\ \exists E_T.\ ge_T \vdash d, tm \xrightarrow{E_T}{}^{+} d', tm' \wedge$$
$$(6)\ E_S \subseteq \mathsf{vis}_S\, \mu \implies$$
$$(6a)\ E_T \subseteq \mathsf{vis}_T\ \mu \wedge$$
$$(6b)\ \forall b_t\ z_t.\ (b_t, z_t) \in E_T \wedge \mathsf{owned}_T\, \mu\, b_t = \mathsf{false} \implies$$
$$\exists b_s\, z.\, \mathsf{f}_{\mathsf{them}}(b_s) = \mathsf{Some}\ (b_t, z) \wedge (b_s, z_t - z) \in E_S$$

External Steps

(at-external)

$$\left(\begin{array}{l} \langle c, m\rangle \sim_\mu \langle d, tm\rangle \wedge \\ \mathsf{vals_inject}\ \mu\ \vec{v_s}\ \vec{v_t} \wedge \mathsf{inject}\ \mu\ m\ tm \wedge \\ \mathsf{at_external}\ c = \mathsf{Some}\ (id_f, \vec{v_s}) \wedge \\ \mathsf{at_external}\ d = \mathsf{Some}\ (id_f, \vec{v_t}) \wedge \\ \nu \triangleq \textbf{leak_out}\ \mu\ \vec{v_s}\ \vec{v_t}\ m\ tm \end{array}\right) \implies$$

(environment)

$$\forall \nu'\ v_s\ v_t\ m'\ tm'.$$
$$\nu \sqsubseteq_{\mathsf{them}} \nu'$$
$$\wedge\ \mathsf{separated}\ \nu\ \nu'\ m\ tm \wedge \mathsf{injection_valid}\ \nu'\ m'\ tm'$$
$$\wedge\ \mathsf{val_inject}\ \nu'\ v_s\ v_t \wedge \mathsf{inject}\ \nu'\ m'\ tm'$$
$$\wedge\ \mathsf{forward}\ m\ m' \wedge \mathsf{forward}\ tm\ tm'$$
$$\wedge\ \textbf{unchanged_on}\ \{(b, z) \mid \mathsf{own}_S\, \nu\, b = \mathsf{Priv}\}\ m\ m'$$
$$\wedge\ \textbf{unchanged_on}\ (\textbf{local_out_of_reach}\ \nu\ m)\ tm\ tm'$$
$$\wedge\ \mu' \triangleq \textbf{leak_in}\ \nu'\ v_s\ v_t\ m'\ tm'$$

(after-external)

$$\implies \exists c'\ d'.\ \mathsf{after_external}\ v_s\ c = \mathsf{Some}\ c'$$
$$\wedge\ \mathsf{after_external}\ v_t\ d = \mathsf{Some}\ d'$$
$$\wedge\ \langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle$$

Figure 5.4: Structured Simulations: Internal and External Step cases

$$
\begin{array}{l}
\mathsf{separated}\ \mu\ \mu'\ m\ tm\ \triangleq \\
\quad (\forall b_1\ b_2\ z.\ \mu\ b_1 = \mathsf{None} \implies \mu'\ b_1 = \mathsf{Some}\ (b_2, z) \implies \\
\qquad\quad b_1 \notin \mathsf{dom}_S\,\mu\ \wedge\ b_2 \notin \mathsf{dom}_T\,\mu) \\
\wedge\ (\forall b_1.\ b_1 \notin \mathsf{dom}_S\,\mu\ \wedge\ b_1 \in \mathsf{dom}_S\,\mu' \implies \neg\mathsf{valid}\ m\ b_1) \\
\wedge\ (\forall b_2.\ b_2 \notin \mathsf{dom}_T\,\mu\ \wedge\ b_2 \in \mathsf{dom}_T\,\mu' \implies \neg\mathsf{valid}\ tm\ b_2)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{locally_allocated}\ \mu\ \mu'\ m\ tm\ m'\ tm'\ \triangleq \\
\begin{array}{rrcl}
 & \mathsf{dom}_S\,\mu' & = & \mathsf{dom}_S\,\mu\ \cup\ \mathsf{freshlocs}\,m\,m' \\
\wedge & \mathsf{dom}_T\,\mu' & = & \mathsf{dom}_T\,\mu\ \cup\ \mathsf{freshlocs}\,tm\,tm' \\
\wedge & \mathsf{owned}_S\,\mu' & = & \mathsf{owned}_S\,\mu\ \cup\ \mathsf{freshlocs}\,m\,m' \\
\wedge & \mathsf{owned}_T\,\mu' & = & \mathsf{owned}_T\,\mu\ \cup\ \mathsf{freshlocs}\,tm\,tm' \\
\wedge & \mathsf{extern}_S\,\mu' & = & \mathsf{extern}_S\,\mu \\
\wedge & \mathsf{extern}_T\,\mu' & = & \mathsf{extern}_T\,\mu
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{local_out_of_reach}\ \mu\ m\ \triangleq \\
\quad \{(b, z)\ |\ b \in \mathsf{owned}_T\,\mu\ \wedge \\
\qquad \forall\ b_0\ \delta.\ \mathsf{f_{us}}\ \mu\ b_0 = \mathsf{Some}\ (b, \delta) \implies \\
\qquad \mathsf{max_perm}\ m\ b_0\ (z - \delta)\ \sqsubset_{\mathsf{perm}}\ \mathsf{Nonempty}\ \vee\ b_0 \notin \mathsf{public}_S\,\mu\}
\end{array}
$$

Figure 5.5: Structured simulations: additional definitions

Steps). The structure of the internal diagram 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 $ge_S \vdash c, m \overset{E_S}{\longmapsto} c', m'$ with effect E_S. Then there exists a matching d', tm', and Kripke-extended structured injection μ' such that $ge_T \vdash d, tm \overset{E_T}{\longmapsto}{}^{+} d', tm'$ and $\langle c', m'\rangle \sim_{\mu'} \langle d', tm'\rangle$. Clause (1) (Kripke extension, $\mu \sqsubseteq_{\mathsf{us}} \mu'$) says that μ' may map more owned blocks than μ (in order to deal with allocations) but otherwise is equal to μ. Clauses (2) and (3) are side conditions, the definitions of which are given in Figure 5.5 (separated and (locally_allocated). Separated $\mu\ \mu'\ m\ tm$ states, essentially, that new regions mapped by μ' but not by μ do not correspond to regions that already exist in m or tm. Locally_allocated $\mu\ \mu'\ m\ tm\ m\ tm'$ states that any new blocks in μ' (fresh blocks allocated in this step) are recorded as local.

Clause (6) is the **guarantee condition**:

- (6a) asserts that the target effects E_T are contained in $\mathsf{vis}_T\ \mu$, assuming that $E_S \subseteq \mathsf{vis}_S\ \mu$. In other words, the compiler preserves the property of "writing and freeing only to visible regions."
- (6b) guarantees that writes to (and frees of) memory locations in the target that are not owned by μ ($\mathsf{owned}_T\ \mu\ b_t = \mathsf{false}$) can be "tracked back" to corresponding writes and frees in the source ($\exists b_s\ z.\ \mathsf{f_{them}}(b_s) =$

Some (b_t, z) and $(b_s, z_t - z) \in E_S$). 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. I 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 $E_1 \cup E_2 \nsubseteq E$ if, for example, the second step restored a value that was overwritten by the first step. Decomposition is required to prove transitivity of structured simulations, as described in Chapter 6.

The external step diagram occupies the bottom half of Figure 5.4. 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 ν' 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 μ', 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 ν' may map more external blocks than ν—in order to deal with allocations performed by the environment—but otherwise is equal to ν. The other nonbolded conditions are adapted from CompCert, and follow in our Coq proofs directly from symmetric conditions on the match-state relation and the internal step diagram.

The conditions listed in bold together compose the **structured simulation rely**. The predicate $\mathsf{unchanged_on}\ U\ m\ m'$ specifies that memories m and m' are equal (same contents and permissions) at the locations in set U. In the source execution, I 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 ν, which is built from μ by updating leakage information as described below. The target-execution condition

$$\mathsf{unchanged_on}\ (\mathsf{local_out_of_reach}\ \nu\ m)\ tm\ tm'$$

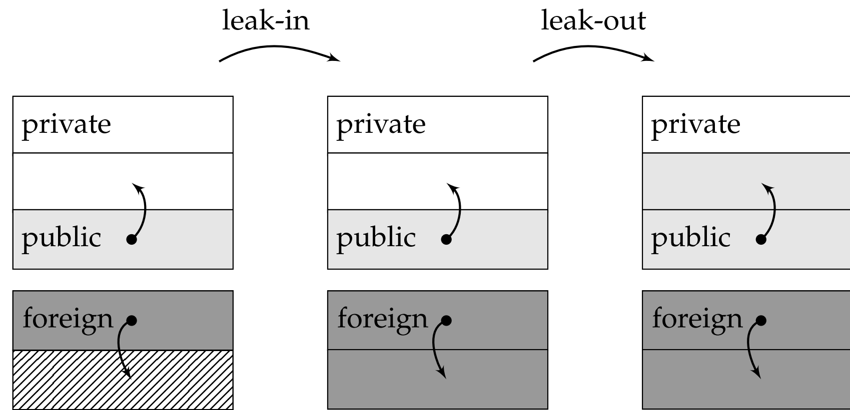


Figure 5.6: Graphical representation of the structured injection leakage operations. The thick black arrows are pointers in memory. The white private and light gray public boxes are owned ("us") blocks. The dark gray boxes are foreign ("them") blocks. The striped box is an Invis memory region that was 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.

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 unchanged_on here, I stipulate the nonmodification conditions of the rely extensionally.

The structured injection ν is built from μ—the injection that originally related at_external states $\langle c, m\rangle \sim_\mu \langle d, tm\rangle$—using the leak_out function depicted graphically in Figure 5.6 and defined in Figure 5.7.[9] The idea is: leak_out "leaks" to the public (other modules) blocks that are reachable by following pointer paths either from the arguments $\vec{v_i}$ to the external call (blocksOf $\vec{v_i}$) or from blocks that were previously shared (shared$_i$ μ). This is a consistency condition: It says that structured simulations may not assume anything about the contents of leaked blocks (the unchanged_on conditions that form the rely satisfied by the environment apply only to private blocks). The functions reach and REACH (as defined in Chapter 2) calculate the transitive closure of the points-to relation on CompCert memories. In the definition of leak_out, I use the auxiliary function export to update the ownership functions of μ to map blocks in the reachable set to Pub.

The leak_in function used to define μ' at the end of the external step diagram plays a role analogous to that of leak_out, except that here, we are

[9] File `compcomp/core/reach.v`.

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

$\mathsf{leak_out}\ \mu\ \vec{v_s}\ \vec{v_t}\ m\ tm\ :\ \mathsf{StructuredInjection}\ \triangleq$
$\quad \textbf{let}\ L_S \triangleq \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ \vec{v_s} \cup \mathsf{shared}_S\ \mu) \cap \mathsf{owned}_S\ \mu$
$\quad\quad L_T \triangleq \mathsf{REACH}\ tm\ (\mathsf{blocksOf}\ \vec{v_t} \cup \mathsf{shared}_T\ \mu) \cap \mathsf{owned}_T\ \mu$
$\quad \textbf{in}\ \mathsf{export}_T\ (\mathsf{export}_S\ \mu\ L_S)\ L_T$

$\mathsf{leak_in}\ \mu\ v_s\ v_t\ m\ tm\ :\ \mathsf{StructuredInjection}\ \triangleq$
$\quad \textbf{let}\ L_S \triangleq \mathsf{REACH}\ m\ (\mathsf{blocksOf}\ [v_s] \cup \mathsf{shared}_S\ \mu) \cap \mathsf{extern}_S\ \mu$
$\quad\quad L_T \triangleq \mathsf{REACH}\ tm\ (\mathsf{blocksOf}\ [v_t] \cup \mathsf{shared}_T\ \mu) \cap \mathsf{extern}_T\ \mu$
$\quad \textbf{in}\ \mathsf{import}_T\ (\mathsf{import}_S\ \mu\ L_S)\ L_T$

Figure 5.7: Block leakage

leaking into μ' new *foreign* blocks reachable from the return value v_i of the external call. Likewise, the import function is similar 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.

Additional Conditions. Structured simulations impose two additional consistency conditions which I have not yet discussed in detail: (1) the simulation relation $\sim_\mu$ is closed under *restriction* of μ to the visible source blocks of μ;[10] and (2) whenever $S \preceq T$, the global environment of target module T is *consistent* with the globals of S: Any symbol mapped by S's global environment is mapped to the same address by T's globals (module T may declare *additional* globals).

Restriction, defined in Figure 5.2 as $\mu\,|_X$ (with X a block set), just limits the domain of μ to X. If $\sim_\mu$ is closed under restriction to the visible blocks, then it does not distinguish memories that differ only at Invis (or None) memory regions. All of the compiler invariants of Compositional CompCert satisfy this property.

[10]Restriction is in `compcomp/core/structured_injections.v`. The closure condition on $\sim_\mu$ is in `compcomp/core/simulations.v`.

The more general version of restriction, and the one actually used in Compositional CompCert, is: μ is closed under restriction to *any* reach-closed superset of $\mathsf{vis}_S\,\mu$, stated as follows:

$$\begin{aligned}\forall X \supseteq \mathsf{vis}_S\,\mu.\ \mathsf{REACH_closed}\ m\ X\ \wedge\ \langle c, m\rangle \sim_{\mu} \langle d, tm\rangle \\ \implies\ \langle c, m\rangle \sim_{\mu|_X} \langle d, tm\rangle\end{aligned}$$

A block set X is $\mathsf{REACH_closed}$ in memory m when it contains its reach closure, as calculated in m:

Definition 7 ($\mathsf{REACH_closed}$)**.**

$$\mathsf{REACH_closed}\ m\ X\ \triangleq\ \mathsf{REACH}\ m\ X \subseteq X$$

Although the key motivation for restriction is the proof of vertical composition for structured simulations (Theorem 4, Chapter 6), the condition also makes intuitive sense: the simulation invariant used to prove correctness of a particular compilation phase should be independent of those blocks that were allocated by other modules but not leaked to the module being compiled. This is one of the ways in which we ensure that compiler phases can, *e.g.*, remove Invis blocks (for example, during a compilation pass that removes a dead function call or memory allocation).

Chapter

6 Separate Compilation

The structured simulations of the previous chapter compose both:

vertically in the sense that multiple structured simulations, for the distinct phases of a compiler, can be composed end-to-end; and

horizontally in the sense that module-local structured simulations for the individual translation units of a program can be composed to build an overall simulation relation for linked source and target programs, as expressed in the linking semantics of Chapter 4.

This chapter presents and explains these two results. I do not give full LaTeXproofs (the mechanized proofs are available in the Coq sources that accompany this thesis). But I do describe the most important invariants in detail.

6.1 Vertical Composition

Realistic compilers are composed of multiple translation phases. These phases are composed transitively, or *vertically*, to yield a full compiler. For example, at the time this thesis was written, the most recent release of the CompCert compiler (version 2.4) included 18 verified compilation phases, each of which was proved correct indepedently of all the others. CompCert 2.1, upon which Compositional CompCert is based, included 16 verified phases, also proved correct independently.

There are a number of reasons why it makes sense to structure a compiler, whether verified or not, as the transitive composition of a number of

small phases. Decomposing the compiler in this way means each translation phase does less, simplifying correctness invariants. The various phases of the compiler can also be used independently. For example, the author of a compiler for another source language besides C, such as Java or Haskell, could target just the backend of CompCert.

This section presents the proof that structured simulations, as presented in Chapter 5, compose transitively. The result is not unexpected—standard forward simulations are trivially transitive, for example. The proof for structured simulations is complicated primarily by the external-call clause (lower half of Figure 5.4), which requires the construction of an interpolating after-external memory m_2' during the transitivity proof, in the intermediate execution between source and target. As mentioned in Chapter 5, the proof of transitivity of the internal-step diagram is tightly dependent on our treatment of effect annotations.

Theorem 4 (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.*

Proof. In the Coq code that accompanies this thesis (lemma `eff_sim_trans` in file `compcomp/core/simulations_trans.v`). □

The most interesting case of the proof is that for the after_external clause. In order to establish the $\langle c_1', m_1' \rangle \sim_{\mu'} \langle c_3', m_3' \rangle$ relation 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, we must provide a suitable intermediate state $\langle c_2', m_2' \rangle$, and in particular the memory m_2'. Figure 6.1 depicts this situation graphically.

6.1.1 Interpolation

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 μ_1' of μ_1) and m_2' can be injected to m_3' via μ_2', such that $\mu' = \mu_2' \circ \mu_1'$ ($\mu_2 \circ \mu_1$ is injection composition). This is assuming μ_1 injects m_1 to m_2, μ_2 injects m_2 to m_3, and μ' injects m_1' to m_3'.

Prior to CompCert 2.0, memory injections did not compose, *i.e.* inject $(\mu_2 \circ \mu_1)$ m_1 m_3 did not follow from inject μ_1 m_1 m_2 and inject μ_2 m_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

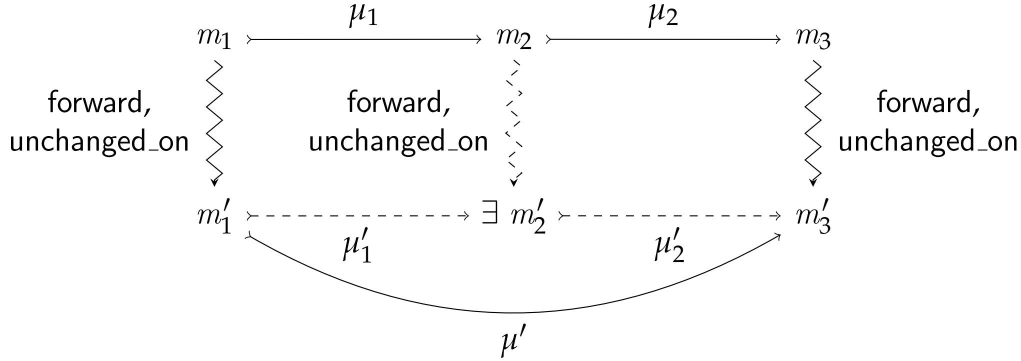


Figure 6.1: Interpolation lemma for composing injection phases $L_1 \preceq L_2$ and $L_2 \preceq L_3$. Solid lines represent assumptions; dashed lines represent constraints that the constructed m'_2 has to satisfy. Composition of simulation proofs in this diagram is left-to-right (contrary to my use of the term *vertical* for phase-by-phase composition of simulations).

injections to facilitate composition, based on a suggestion of Tahina Ramananandro. The interpolation lemma provides the counterpart to this composition, by guaranteeing that the post-call injection inject μ' m'_1 m'_3 can be split into some m'_2, μ'_1, and μ'_2 with inject μ'_1 m'_1 m'_2 and inject μ'_2 m'_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 unchanged_on 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 μ'_1 and μ'_2 with the required properties can indeed be constructed.

6.2 Horizontal Composition

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 con-

ditions corresponding to the structured simulation guarantees of Chapter 5, and to *valid* target semantics (a technical property related to the CompCert memory model, explained below).

Theorem 5 (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****,*
- 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,*
- *each* T_i *is* ***valid****, and*
- *the global environments of the* S_i *(resp.* T_i*) have equal domain, 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*.*

Proof. In Coq.[1] The simulation invariant is described in Section 6.2.2. □

The $\leq$ in the theorem denotes forward simulation on whole programs, as in Chapter 5. As Corollary 7 will show, establishing $\leq$ is sufficient to prove contextual equivalence of open multimodule programs (by linking with a closing context). Restricting to modules with equal global domain may seem counterintuitive; linking can, at least in principle, enlarge the set of global addresses that are visible to any one module in isolation. The "globals have equal domain" assumption defers this reasoning to the program logic (Chapter 7), in which it is necessary either to prove safety monotonicity under global environment extension, or to preprocess modules to propagate global declarations (the current VST strategy).

A *valid* semantics, as in Section 3.3, 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). This condition is true for all contexts we care about (for example, it holds of all programs in CompCert's Clight [Lemma 9] and x86 languages [Theorem 2], which do not permit storing invalid addresses into memory).

But why is it necessary? The answer is technical. In order to establish, during the proof of Theorem 5, the

$$\mathsf{REACH}\ tm\ (\mathsf{globalsOf}\ ge_T\ \cup\ \mathsf{blocksOf}\ \overrightarrow{v_t})\ \subseteq\ \mathsf{dom}_T$$

---

[1] The theorem statement is in `compcomp/linking/linking_spec.v`. Theorem `link` in file `compcomp/linking/linking_proof.v` gives the proof.

condition of the *Initial Core* clause of structured simulations (Figure 5.3), it is necessary to know that the target memory tm satisfies mem_valid at each intermodule interaction point. Otherwise, we could not show that the reach-closure of the set globalsOf ge_T $\cup$ blocksOf $\overrightarrow{vt}$ is contained in dom_T (we instantiate dom_T, in the proof, to equal the set of valid blocks in tm). One could imagine a different proof strategy, in which dom_T is instantiated directly to REACH tm (globalsOf ge_T $\cup$ blocksOf $\overrightarrow{vt}$), for example. But then we fall afoul of a validity condition required elsewhere in structured simulations, that $\mathsf{dom}_T \subseteq$ validBlocks tm. Ultimately, these properties are tied deeply to the specifics of CompCert's memory model, such as the CompCert allocation model (Chapter 2), and to the specifics of structured simulations.

The restriction to reach-closed semantics (Section 3.3) is best motivated with an example. Consider the following C program, a variation of the second of the C example programs from Chapter 1:

```
//Module A
void g(int*);
int f(void) {
  int a; int b = 3;
  g(&a);
  return b;
}
```

Function `A.f` calls an external function `B.g`, passing `&a` as argument.

Now imagine we link with the following context:

```
//Module B
void g(int* p) {
  *(int*)((uintptr_t)p + 4) = 4;
}
```

in which `B.g` writes the value `4` to address `&b = p+1` by first casting `p` to an integer, adding `4` (the size in bytes of integers on a 32-bit machine), then casting back to an integer pointer and performing the write. In the context of this (implementation-defined) `g`, standard compiler optimizations such as constant propagation of `b` in `A.f` are unsound, as the discussion in Chapter 1 showed.

The point of a compositional compiler, however, is to enable local modular compilation, which should depend only on translation-unit-local analyses. Correctness of optimizations like constant propagation, dead-code elimination, and inlining should not depend on the *particulars* of the larger

program context in which a module is executed (*e.g.*, the implementation of `B.g`), only that the larger context respects the C-level abstractions assumed by the compiler.

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 5. We do this in general, for arbitrary interaction semantics, by observing that the write to `&b = (int*)((uintptr_t)p + 4)` is ill-formed not because it goes wrong (the write is safe under certain interpretations of the behavior of integer–pointer casts), but because it's a write to a location that the context `B.g` shouldn't have "known about" in the first place.

Put another way, address `&b` was not reachable via pointer arithmetic[2] either from `g`'s initial arguments (pointer arithmetic across local variable regions is undefined), from global variables, or from the return values of external calls `g` may have made. 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 5.4, but stated as a single-program property, independent of any particular structured injection μ. We formalize the notion of a semantics that respects this characterization of visible locations as the *reach-closed semantics* of Chapter 3, Section 3.3.

From the perspective of compiler correctness proofs, the restriction to reach-closed contexts is what *enables* program transformations: It would be unsound, for example, to constant-propagate `b` out of memory if the larger program context depended on it, as in the example program above.

6.2.1 Reach-Closed Contextual Equivalence

As a corollary of Theorem 5, we get a form of contextual equivalence when the source modules are reach-closed and the target modules are valid, stated in terms of a variation of Definition 4 in which contexts satisfy a few additional properties. Informally, if each module in multimodule program P_S is compiled to the corresponding module in target P_T, then P_S and P_T have the same behavior (termination, divergence) when linked with a *well-defined* program context C. $\mathcal{L}(C, P_S)$ may also go wrong, in which case we say nothing about the behavior of $\mathcal{L}(C, P_T)$.

---

[2]When the program context is implemented in a language like x86 assembly, it might seem strange to say "not reachable via pointer arithmetic" 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.

More formally, well-defined contexts are those deterministic C that self-simulate, and which are both reach-closed and valid.

Definition 8 (Well-Defined Contexts)**.**

$$\mathsf{well_defined}\ C \ \triangleq\ \mathsf{deterministic}\ C\ \wedge\ C \preceq C\ \wedge\ \mathsf{reach_closed}\ C\ \wedge\ \mathsf{valid}\ C$$

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 5, 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 Chapter 5 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).

Reach-closed contextual equivalence is then just equitermination, assuming the source linked program is safe, in all well-defined contexts:

Definition 9 (Reach-Closed Contextual Equivalence)**.**

$$\begin{array}{l} P_S \sim_{\mathsf{rc}} P_T \ \triangleq\ \forall C\ j\ m\ tm\ ge\ v\ \vec{v_S}\ \vec{v_T}. \\ \quad \mathsf{well_defined}\ C\ \wedge\ \mathsf{init_inv}\ j\ ge\ \vec{v_S}\ m\ ge\ \vec{v_T}\ tm\ \wedge\ \mathsf{safe}\ ge\ C\ P_S\ v\ \vec{v_S}\ m \\ \quad\quad \Longrightarrow\ (\mathsf{terminates}\ ge\ C\ P_S\ v\ \vec{v_S}\ m\ \Longleftrightarrow\ \mathsf{terminates}\ ge\ C\ P_T\ v\ \vec{v_T}\ tm) \end{array}$$

Invariant $\mathsf{init_inv}$ is defined as in Section 5.1. Essentially: injection j relates m to tm and $\vec{v_S}$ to $\vec{v_T}$, and is the identity on $\mathsf{dom}(ge)$. Also, $\vec{v_T}$ and tm must not contain invalid pointers (to memory regions that have not yet been allocated).

The global environment ge : $\mathsf{Genv\ unit\ unit}$ is used solely to ensure that the global environments of the linked modules have equal domain; hence we use the same ge in both source and target.[3]

Predicates safe and $\mathsf{terminates}$ are overloaded to operate on whole programs, instead of configurations, as follows. Say a program P is *initializable* at entry point v with arguments $\vec{v}$ if initialization succeeds for v at $\vec{v}$.

Definition 10 ($\mathsf{initializable}\ ge\ C\ P\ v\ \vec{v}$)**.** $\exists c.\ \mathsf{initial_core}_{\mathcal{L}(C,[\![P]\!])}\ ge\ v\ \vec{v} = \mathsf{Some}\ c$

[3]Recall that the global environments used to look up function bodies in $\mathcal{L}$ are language-specific, and therefore the per-module ones.

A program P has behavior b in context C and memory m, at entry point v with arguments $\vec{v}$, if either (i) P is initializable, in linked semantics $\mathcal{L}(C, [\![P]\!])$, to a configuration $\langle c, m \rangle$ that has behavior b, or (ii) $b = \mathsf{Going_wrong}$ and P is not initializable (P initially went wrong in context C). Program P terminates if it has behavior Termination.

If each of the pairs S_i, T_i in a multimodule program is related by structured simulations $[\![S_i]\!] \preceq [\![T_i]\!]$, then the linked source and target programs are reach-closed contextual equivalent.

Corollary 7 (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 $S_0, S_1, \cdots, S_{N-1}$ with equal global domains, and valid deterministic target modules $T_0, T_1, \cdots, T_{N-1}$. If for each i, $[\![S_i]\!] \preceq [\![T_i]\!]$, then $P_S \sim_{\mathsf{rc}} P_T$.

Proof. The Coq proof is file `compcomp/linking/context_equiv.v`. □

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 well-defined (*cf.* Definition 8). 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).

We also have a form of contextual refinement.[4]

Definition 11 (Reach-Closed Contextual Refinement)**.**

$$\begin{array}{l} P_T \sqsubseteq_{\mathsf{rc}} P_S \triangleq \forall C\ j\ m\ tm\ ge\ v\ \vec{v_S}\ \vec{v_T}. \\ \quad \mathsf{well_defined}\ C \ \wedge\ \mathsf{init_inv}\ j\ ge\ \vec{v_S}\ m\ ge\ \vec{v_T}\ tm \\ \quad \Longrightarrow\ (C, P_T, ge, v, \vec{v_T}, tm)\ \leq_{beh}\ (C, P_S, ge, v, \vec{v_S}, m) \end{array}$$

In the definition, the $\leq_{beh}$ relation of Section 5.2.3 is overloaded to operate on programs, in addition to behaviors. The relation

$$(C, P_T, ge, v, \vec{v_T}, tm)\ \leq_{beh}\ (C, P_S, ge, v, \vec{v_S}, m)$$

means: for all behaviors b_T of program P_T in context C, initialized at v with arguments $\vec{v_T}$ in memory tm, there exists a behavior b_S of P_S in context C (initialized at $v, \ldots$) such that b_T is a refinement of b_S ($b_T \leq_{beh}$

[4]Strictly speaking, not derivable from the equivalence shown above (Definition 9; we will refine divergence as well as termination behavior).

b_S). Refinement of behaviors is defined as in Section 5.2.3. Up to classical reasoning, every program has at least one behavior:

Theorem 6 (Behavior Exists)**.** *For all programs P, global environments ge, entry points v, initial arguments $\vec{v}$, initial memories m, and contexts C, there exists a behavior b such that $(C, P, ge, v, \vec{v}, m)$ has behavior b.*

Proof. In Coq.[5] □

We then get that simulation, as in Corollary 7, implies reach-closed contextual refinement.

Corollary 8 (Simulation Implies Contextual Refinement)**.** *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 $S_0, S_1, \cdots, S_{N-1}$ with equal globals domains, and valid deterministic target modules $T_0, T_1, \cdots, T_{N-1}$. If for each i, $[\![S_i]\!] \preceq [\![T_i]\!]$, then $P_T \sqsubseteq_{\mathsf{rc}} P_S$.

Proof. The Coq proof is file `compcomp/linking/context_equiv.v`. By Theorem 5, we have $\mathcal{L}(C, [\![P_S]\!]) \leq \mathcal{L}(C, [\![P_T]\!])$. By assumption and Theorem 3, we have deterministic $\mathcal{L}(C, [\![P_T]\!])$. The theorem follows by Corollary 5 of Chapter 5 (behavior refinement from whole-program simulation). □

In the definitions $\sim_{\mathsf{rc}}$ and $\sqsubseteq_{\mathsf{rc}}$ above, it's important that the definition of well-defined contexts is not too narrow. Otherwise, we risk ruling out reasonable programs. At the very least, every C-program context should be well-defined in the sense of Definition 8. Otherwise, the equivalence $\sim_{\mathsf{rc}}$ would be quite weak (it would not be robust to linked C-language contexts).

As justification, I have proved the following.

Theorem 7 (Clight Programs are Well-Defined Contexts)**.** *Take Clight program P. The interpretation of $[\![P]\!]$ as module semantics is well-defined according to Definition 8.*

This theorem lower-bounds the qualification over contexts in $\sim_{\mathsf{rc}}$: C is at least instantiable by any relation that corresponds to a well-defined Clight program.

Since C may express arbitrary relations in Coq's Gallina, up to the conditions Definition 8, there are contexts C that correspond to no well-defined Clight program, yet are still well-defined according to Definition 8.

---

[5]File `compcomp/linking/context_equiv.v`.

For example, take C equal the (undecidable) relation that reads as input the description in memory of a Turing machine (and its input) and returns Vint 1 if the Turing machine terminates (on that input), and Vint 0 otherwise. This C is deterministic (no deterministic Turing machine both terminates and infinite loops), self-simulates (the Turing-machine description is an array of integers in the heap, pointed to by one of C's arguments), and is both reach-closed and valid (C could be implemented to write only integers to a sequence of memory regions it allocates itself).

The proof [6] of Theorem 7 relies on a number of auxiliary lemmas, corresponding one-for-one with the conditions of Definition 8.

Lemma 7. *Clight programs are deterministic.*

Proof. File `compcomp/cfrontend/Clight_lemmas.v`. Due to a misspecification in original CompCert of certain compiler intrinsics (architecture-dependent instructions for certain 64-bit operations) the proof of this theorem currently assumes the I64 helpers case. □

Lemma 8. *For all Clight programs P, $[\![P]\!] \preceq [\![P]\!]$.*

Proof. File `compcomp/cfrontend/Clight_self_simulates.v`. □

Lemma 9 (Clight Programs are Valid)**.** *Every Clight program is valid, in the sense of Definition 3.*

Proof. File `compcomp/linking/clight_nucular.v`. □

Theorem 1 proved that all Clight programs are reach-closed.

6.2.2 Linking Invariants

The main difficulty in proving Theorem 5 and by extension, Corollaries 7 and 8, is in devising a simulation invariant[7] to relate the stacks-of-cores runtime states of the linked programs P_S and P_T.

The situation is presented schematically in Figure 6.2. 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 μ to denote the structured simulation that relates the callees c and d, and

[6] File `compcomp/linking/context.v`.

[7] File `compcomp/linking/linking_inv.v`.

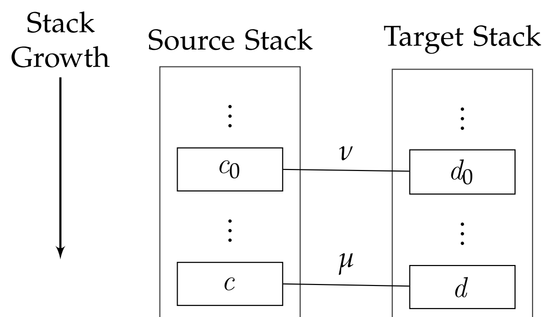


Figure 6.2: Schematic representation of the stacks-of-cores linking invariant. The white boxes are core states. Source core c and target core d are callees at the bottom of the LinkedState callstack, related by structured injection μ (memory is elided). Cores c_0 and d_0 are caller cores related by ν.

ν to denote the injection that relates callers c_0 and d_0. In the figure, we elide the memories (for callers, the memory at the call point is existentially quantified). A caller core may be a callee with respect to another caller higher on the callstack.

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

$$\mathsf{foreign}_S\,\mu\ \cap\ \mathsf{owned}_S\,\nu \subseteq \mathsf{public}_S\,\nu \tag{6.1}$$

From the fact that source modules are reach-closed

$$E_S \subseteq \mathsf{REACH}\ m\ (\mathsf{roots}\ ge\ r) \tag{6.2}$$

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 ν, which are disjoint from the blocks marked as public by ν, remain unmodified.

A difficulty here is how to relate the root sets of source modules to the visible sets vis_S used in the simulation relations. We do this by maintaining the following two invariants:

$$\mathsf{roots}\ ge\ r \subseteq \mathsf{vis}_S\,\mu \tag{6.3}$$

$$\mathsf{REACH}\ m\ (\mathsf{vis}_S\ \mu) \subseteq \mathsf{vis}_S\ \mu \tag{6.4}$$

Invariant (6.3) says that the root set of the source semantics is a subset of the visible source blocks in μ. This invariant holds initially, when r is first created, and is maintained at external function calls and returns. Condi-

tion (6.4), which we maintain as an invariant of all structured simulations, says that the visible set is closed under reachability. These two conditions, plus (6.2) and monotonicity of the REACH relation, imply that E_S is a subset of $\mathsf{vis}_S\ \mu$. This fact, together with condition (6.1) above, is sufficient to prove the unchanged_on relies of Figure 5.4 at the point at which the running core returns to its calling context.

Linking Invariants: Details. More formally, we define the toplevel simulation invariant $\sim_\mu$ (called match_state below) that relates linked states x_1 (in the source) and x_2 (in the target) as follows:

```
1  match_state μ (x1 : LinkedState N modulesS) m
2                (x2 : LinkedState N modulesT) tm ≜
3    let s1 ≜ x1.stack in
4    let s2 ≜ x2.stack in
5    let pf1 ≜ callStack_nonempty s1 in
6    let pf2 ≜ callStack_nonempty s2 in
7    let c ≜ Stack.head pf1 s1 in
8    let d ≜ Stack.head pf2 s2 in
9    (∃ (pf : c.idx = d.idx) p⃗. head_inv c d pf μ p⃗ m tm
10    ∧ tail_inv μ⃗ (pop s1) (pop s2) m tm)
11    ∧ x1.plt = x2.plt
12    ∧ ∀i : IN. valid_genv (modulesT i).ge tm
```

Recall here that the core states of linking semantics are defined (*cf.* Chapter 4) as records of: a procedure linkage table (plt) and a (heterogeneous) stack of core states, corresponding to dynamic invocations of the modules in the program.

```
Record LinkedState (N : pos) (modules : IN → Modsem) ≜
  { plt : ident → option IN;
    stack : Stack (Core N modules) }
```

The $modules$ map pairs integers in the range 0 to $N-1$ with the Modsem module semantics associated with each module in the program. The invariant is also parameterized by a number of structured simulation relations $\sim$, one for each compiled module in the program. We index $\sim^{idx}$ to denote the simulation relation for module number idx.

The first few lines of the match_state invariant just introduce new names for the various parts of the linked states x_1 and x_2. The let-bound variables c and d are defined, via Stack.head, as the cores on top of the source and

target stacks respectively, as in Figure 6.2. The names s_1 and s_2 are aliases for the stack component of linked states x_1 and x_2. pf_1 and pf_2 are proofs that stacks s_1 and s_2 are nonempty (an invariant of linking semantics, as defined in Chapter 4). These proofs are passed as arguments to Stack.head, to ensure that Stack.head is total. Line 11 states that source and target linked states contain equal PLTs. Line 12 asserts that each target module global environment is valid with respect to tm (the environment does not map globals to addresses that were never allocated in tm).

The key section of the invariant runs from line 9 to line 10. Line 9 is the invariant head_inv on the topmost cores on the stack—c and d. Line 10 is the invariant tail_inv on the remaining (suspended) cores—pop s_1 and pop s_2. Existentially quantified are: a proof pf that that the module index of core c equals the index of core d, and a list of *frame packages* $\vec{p}$ that relate each pair of source–target suspended (caller) cores on the source and target stacks. Frame packages are records

$p \in$ frame_pkg $\triangleq$ mk_frame_pkg
{ frame_μ : StructuredInjection;
  frame_m : mem;
  frame_tm : mem;
  frame_val : valid frame_μ frame_m frame_tm }

that contain:

- frame_μ: a structured injection relating two (source–target) suspended cores;
- frame_m: the source memory at which the cores are related;
- frame_tm: the target memory at which the cores are related; and
- frame_val: a proof that frame_μ is valid for frame_m and frame_tm.

Think of frame_m_0 and frame_tm_0 as the source and target memories, respectively, in which this particular pair of source–target (indirect) caller cores (say c_0 and d_0 as in Figure 6.2) were suspended, waiting on an external function call. frame_μ is likewise the structured simulation that related the source–target caller cores c_0 and d_0 at the point of suspension. We describe head_inv and tail_inv in turn.

Running Cores. The invariant head_inv that holds of the topmost cores on the stack—c and d—is defined as follows:

```
1  head_inv c d pf μ p⃗ m tm ≜
2    let idx ≜ c.idx in
3     ⟨c, m⟩ ~_μ^idx ⟨d, tm⟩
4     ∧ (∀p ∈ p⃗. callee_caller_inv m μ p)
5     ∧ (∃B. roots ge B ⊆ vis_S μ ∧ R_idx c m B)
6     ∧ dom_T μ = valid_blocks tm
7     ∧ I_idx d tm
```

The parameters of the definition are: c and d, the source and target running cores respectively; pf, a proof that the modules indices of c and d are equal; the structured injection μ that relates c and d; the frame packages $\vec{p}$ that relate the remaining suspended cores; and the source and target memories m and tm.

Line 3 asserts that configuration c, m is related to d, tm by $\sim_{\mu}^{idx}$, the simulation relation associated with module idx, indexed by structured injection μ. In line 5, we state that there exists a block set B such that (1) the roots of B and global environment ge are a subset of the visible source blocks of μ (this is Condition 6.3 above); and (2) B satisfies the invariant $\mathcal{R}_{idx}$ maintained by the reach-closed semantics of source module idx. Line 6 is a technical condition on the target blocks of μ (asserting that $\mathsf{dom}_T\ \mu$ is always the set of valid blocks in tm). Line 7 maintains the validity invariant $\mathcal{I}_{idx}$ associated with the semantics of target module idx.

Predicate callee_caller_inv on line 4 delineates the relation between the structured injection μ and the frame packages $p \in \vec{p}$ that relate the suspended cores on the stack, at memory m. It is defined as follows:

```
1  callee_caller_inv m μ p ≜
2    let μ₀ ≜ p.frame_μ in
3    let m₀ ≜ p.frame_m in
4    let tm₀ ≜ p.frame_tm in
5       μ₀ ⊑ μ
6     ∧ separated μ₀ μ m₀ tm₀
7     ∧ owned_S μ₀ ∩ owned_S μ = ∅
8     ∧ owned_T μ₀ ∩ owned_T μ = ∅
9     ∧ REACH m (vis_S μ) ∩ (owned_S μ₀) ⊆ public_S μ₀
```

The assertion $\mu_0 \sqsubseteq \mu$ on line 5 ensures that μ extends μ_0 with respect to the Kripke-order $\sqsubseteq$. This order is similar to $\sqsubseteq_{\mathsf{us}}$ and $\sqsubseteq_{\mathsf{them}}$: μ may map more blocks than μ_0, in order to deal with allocations, but is otherwise equal to μ_0 (wherever μ_0 is defined). The primary difference is that $\sqsubseteq$ does not

distinguish between owned and extern blocks in μ_0 and μ, instead treating μ_0 and μ as if they were "unstructured" injections, in standard CompCert style (*cf.* Chapter 2). Why is the unstructured $\sqsubseteq$ appropriate here? μ_0 and μ relate the running states of different source–target module pairs, for example, of compiled modules idx and idx'. The blocks labeled owned, or allocated, by module idx will be labeled extern (not owned) by module idx', and vice versa.

Lines 6 to 9 give the other key invariants: μ must be separated from μ_0, with respect to the existentially quantified memories m_0 and tm_0 associated with μ_0 (see Figure 5.5 for the definition of separated); the source/target owned blocks of μ_0 and μ must be *disjoint* (Lines 7 and 8); finally, the set of source-language blocks declared owned by the caller injection μ_0 but also reachable in m from the visible set of the callee injection μ must also be declared public by μ_0.[8] In other words, blocks leaked into μ's visible set at previous interaction points must be declared public by the (direct or indirect) caller that owns the blocks in question.

The intuition here is: A well-formed source–target state pair of the linked program is one in which the simulation relation μ_0 (relating cores c_0 and d_0) makes no claim, at external calls, on the values contained in blocks that have been leaked to the cores in callee position with respect to c_0, d_0. (Recall that structured simulation proofs may assume that the memory is unchanged over external calls only at private blocks, as in the *External Steps* case of Figure 5.4.) The reason: leaked regions may be modified over the external call, *e.g.*, by the running cores c and d.

Suspended Cores. The invariant that relates suspended (caller) cores to one another is defined:

```
1  tail_inv p⃗ s1 s2 m tm ≜
2        all_caller_callees (λp p0. caller_callee_inv m (frame_μ p) p0) p⃗
3     ∧  frame_all p⃗ m tm s1 s2
```

where all_caller_callees is given by the following pair of recursive functions:

[8] We impose an analogous invariant on target blocks (not shown); see the code repository that accompanies this thesis for details.

$$
\begin{array}{l}
\textsf{all}\ T\ P\ (l : \textsf{list}\ T) \triangleq \\
\quad \textbf{case}\ l\ \textbf{of} \\
\qquad \mid \textsf{nil} \rightarrow \textsf{True} \\
\qquad \mid a :: l' \rightarrow P\ a \wedge \textsf{all}\ P\ l'
\end{array}
$$

$$
\begin{array}{l}
\textsf{all_caller_callees}\ (T : \textsf{Type})\ (P : T \rightarrow T \rightarrow \textsf{Prop})\ (l : \textsf{list}\ T) \triangleq \\
\quad \textbf{case}\ l\ \textbf{of} \\
\qquad \mid \textsf{nil} \rightarrow \textsf{True} \\
\qquad \mid a :: l' \rightarrow \textsf{all}\ (P\ a)\ l' \wedge \textsf{all_caller_callees}\ T\ P\ l'
\end{array}
$$

Line 2 of the tail_inv listing above asserts that, for each $p \in \vec{p}$, the structured injection given by package p (frame_μ p) is related by caller_callee_inv to *each* caller package p_0 in the tail of $\vec{p}$ at the point at which p appears (*i.e.*, in caller position with respect to p). This invariant is required in order to re-establish head_inv when the running source–target cores return to their callers.

Line 3 of tail_inv asserts an invariant on each source–target pair c_0,d_0 in the source–target stacks s_1 and s_2, as defined by the recursive predicate frame_all:

$$
\begin{array}{l}
\textsf{frame_all}\ \vec{p}\ m\ tm\ s_1\ s_2 \triangleq \\
\quad \textbf{case}\ \vec{p},\ s_1,\ s_2\ \textbf{of} \\
\qquad \mid \textsf{mk_frame_pkg}\ \mu_0\ m_0\ tm_0\ _ :: \vec{p}',\ c_0 :: s_1',\ d_0 :: s_2' \rightarrow \\
\qquad\qquad \exists\ (pf : c_0.\textsf{idx} = d_0.\textsf{idx}). \\
\qquad\qquad \exists\ e_1\ \vec{v}_1. \\
\qquad\qquad \exists\ e_2\ \vec{v}_2. \\
\qquad\qquad\quad \textsf{frame_inv}\ c_0\ d_0\ pf\ \mu_0\ m_0\ m\ e_1\ \vec{v}_1\ tm_0\ tm\ e_2\ \vec{v}_2 \\
\qquad\qquad \wedge\ \textsf{frame_all}\ \vec{p}'\ m\ tm\ s_1'\ s_2' \\
\qquad \mid \textsf{nil},\ \textsf{nil},\ \textsf{nil} \rightarrow \textsf{True} \\
\qquad \mid _,_,_ \rightarrow \textsf{False}
\end{array}
$$

In addition to asserting that $\vec{p}$, s_1, and s_2 are all the same length, frame_all applies a subsidiary invariant, frame_inv, to each pair of cores c_0, d_0 in s_1 and s_2. This "per-frame" invariant is defined as follows:

```
1  frame_inv μ0 m0 m e1 v⃗1 tm0 tm e2 v⃗2 ≜
2    let ν0 ≜ leak_out μ0 v⃗1 v⃗2 in
3    let idx0 ≜ c0.idx in
4
5    (* per−frame invariants, on c0, d0, μ0, m0, and tm0 *)
6    inject (as_inj μ0) m0 tm0
7    ∧ valid μ0 m0 tm0
8    ∧ ⟨c0, m0⟩ ~^{idx0}_{μ0} ⟨d0, tm0⟩
9    ∧ at_external c0 = Some (e1,v⃗1)
10   ∧ at_external d0 = Some (e2,v⃗2)
11   ∧  inject (as_inj μ0 ⇂vis μ0) v⃗1 v⃗2
12
13   (* source visibility *)
14    ∧ (∃B. roots ge B ⊆ vis_S μ0 ∧ R_{idx0} c0 m0 B)
15
16   (* target validity *)
17   ∧  dom_T  μ0 = validBlocks tm0
18   ∧  I_{idx0} d0 tm0
19
20   (* invariants relating m0, tm0 to m, tm *)
21   ∧  forward m0 m
22   ∧ forward tm0 tm
23   ∧ unchanged_on {(b, z) | own_S ν0 b = Priv} m0 m
24   ∧  unchanged_on (local_out_of_reach ν0 m0) tm0 tm
```

Lines 2 and 3 establish local definitions:

- ν_0 is the injection that results by "leaking out" into μ_0 all blocks exposed by c_0 and d_0 to their callers (recall that each pair of cores c_0, d_0 is suspended at_external on an external function call);
- idx_0 is the index of the module from which c_0 was spawned (which happens to be equal to the index of d_0, by the existentially quantified pf on line 4 of frame_all).

The other invariants form four natural groups:

Lines 6 to 11 specify "per-frame" invariants on the states c_0 and d_0, the memories m_0 and tm_0, and the structured injection μ_0 that relates them. Configurations c_0, m_0 and d_0, tm_0 should be related by $\sim^{idx_0}_{\mu_0}$, the simulation invariant of module idx_0. In addition, μ_0 injects m_0 to tm_0 and $\vec{v}_1$ to $\vec{v}_2$, for $\vec{v}_1$ the arguments of c_0's call to external function e_1 (at_external $c_0 = \text{Some } (e_1, \vec{v}_1)$) and $\vec{v}_2$ the arguments of d_0's call to external function e_2 (at_external $d_0 = \text{Some } (e_2, \vec{v}_2)$).

Line 14 maintains that invariant that relates the visible source blocks of μ_0 to the reach-closed invariant R_{idx_0} (of source module idx_0 associated with c_0).

Lines 17 and 18 parallel lines 6 and 7 of the definition of head_inv.

Lines 21 to 24 relate memories m_0 and tm_0 to the "active" memories m and tm. For example, m and tm should be forward from m_0 and tm_0 respectively. But also, m_0 and m must be equal in blocks marked private by ν_0, and likewise for tm_0 and tm at "out of reach" locations. These last two conditions directly match the unchanged_on conditions of the *External Steps* diagram of structured simulations (Figure 5.4).

Chapter

7 Modular Verification

Verifiable C [ADH+14, Chapter 24] is a separation logic for CompCert's Clight language that supports higher-order features such as stored function pointer specifications. This chapter connects the Verifiable C logic to the compiler correctness results I presented in Chapter 6. In particular, I show how modular separation logic proofs in Verifiable C can be connected to the linking semantics of Chapter 4 and in this way, composed with compiler correctness. At a high level, the result is: independent program-logic proofs of the Hoare triples

$$\Gamma_1 \vdash_{func} f_1, f_2 : \Gamma$$

and

$$\Gamma_2 \vdash_{func} f_3, f_4, f_5 : \Gamma$$

in which $f_1, f_2, \ldots$ are function bodies, Γ proved function specifications, and Γ_1, Γ_2 assumed, imply partial correctness of the linked target program

$$\mathcal{L}(C, [\![\mathsf{CompCert}(f_1, f_2)]\!]_{\mathsf{Asm}_{\mathsf{Sem}}}, [\![\mathsf{CompCert}(f_3, f_4, f_5)]\!]_{\mathsf{Asm}_{\mathsf{Sem}}})$$

that results from independently compiling f_1, f_2 and f_3, f_4, f_5. The linked C here is a program context compatible with the Hoare-style specifications of the external functions called by (but implemented by none of) the compiled modules, as encapsulated in the Hoare-logic function specifications $\Gamma_1 \cup \Gamma_2$. The remainder of this chapter explains the details.

7.1 Modular C Program Logic

In some ways, Verifiable C is just a conventional Hoare logic. Judgments have the following familiar form:

$$\Delta \vdash \{P\}\ c\ \{R\}$$

in which Δ is a type context, P is the precondition of C statement c, and R is the postcondition.

At the same time, the Verifiable C logic is also quite complex, owing to the complexity of the C programming language. For example, Δ does not just give the types of variables that may appear in free in c (*i.e.*, temporaries). It also types

- function parameters,
- addressed local variables, and
- global variables

and assigns pre- and postconditions to the functions defined/called by the program. The postcondition R is really a series of postconditions. Since C basic blocks may exit in multiple ways (by **`continue`**, **`break`**, **`return`**, and by falling through a switch), R records multiple postconditions, one for each possible return case.

7.1.1 Inference Rules

Chapter 24 of [ADH+14] describes the inference rules of the Verifiable C logic in detail. For the most part, the rules are conventional (if complicated by the vagaries of C). For example, here is the rule for load from memory:

$$\frac{\text{readable } \pi}{\Delta \vdash \{\triangleright(e \overset{\pi}{\mapsto} v\ *\ P)\}\ x := [e]\ \{\exists v_{old}.\ x = v \wedge (e \overset{\pi}{\mapsto} v\ *\ P)[v_{old}/x]\}}\ (\textsc{SemaxLoad})$$

The command $x := [e]$ assigns to x the value in memory at location e, equal v as specified by precondition $e \overset{\pi}{\mapsto} v$. $e \overset{\pi}{\mapsto} v$ is an instance of the *maps-to* predicate of separation logic, asserting a (singleton) heap containing value v at the location to which expression e evaluates. π is a *share*, the program-logic counterpart to the CompCert permissions of Chapter 2. Chapter 41 of [ADH+14] describes how shares are constructed in Verifiable C; Chapter 42, of which I am a co-author, describes how shares are erased to CompCert permissions. In the rule above, we just need that π is a *readable* share (giving at least read permission; π might permit writes as well).

The precondition $\triangleright(e \overset{\pi}{\mapsto} v * P)$ is prefixed with the *later* operator $\triangleright$. This $\triangleright$ highlights another aspect of the Verifiable C logic: in order to reason about complicated patterns of (mutual) recursion over, *e.g.*, C function pointers, the interpretation of the logic is *step-indexed* [AM01, AMRV07, AAV02, HDA10]. In the model of the logic, predicates (and states) are paired with natural numbers k, the number of steps for which the predicate will continue to make claims on the system. $\triangleright P$ says that P holds not at the current k (*e.g.*, of the state in which the command is executed), but only at $k-1$. In the load rule, we need only that $\triangleright(e \overset{\pi}{\mapsto} v * P)$, as opposed to the stronger $e \overset{\pi}{\mapsto} v * P$, because the assignment $x := [e]$ itself takes a step.

Assertion $R = \exists v_{old}.\ x = v \wedge (e \overset{\pi}{\mapsto} v * P)[v_{old}/x]$ is the strongest postcondition of the load command. It states that, after the assignment,

- variable x equals v, and
- there exists an old value of x, call it v_{old}, such that the precondition $(e \overset{\pi}{\mapsto} v * P)$ holds with v_{old} substituted for x.

This particular load rule is the most general. In the logic, other specialized forms are derived, for loading from arrays, structures, *etc.*, and for the special case in which x does not appear free in the precondition.

7.1.2 Proving Whole Modules

In addition to the basic Hoare triple, the Verifiable C logic provides a judgment form, semax_func, for composing function body proofs in order to prove whole modules. This judgment has form:

$$V, \Gamma \vdash_{func} \vec{f} : \Gamma'$$

in which

varspecs V is an environment mapping (global) variables to their types.

funspecs Γ, Γ' are lists of function name, function specification (funspec) pairs. Γ are the function specifications that one may assume (but only *later*) when proving functions $\vec{f}$. Γ' are the specifications that are proved. A single funspec is defined by the inductive:

```
Inductive funspec : Type ≜
  | mk_funspec : funsig →
    ∀ A : Type.
    ∀ (P : A → environ → mpred)
      (Q : A → environ → mpred). funspec
```

The first constructor argument (of type funsig) is the function's (C-type) signature. environ is a triple of the global environment, the function temporaries environment (unaddressed locals), and the function variable environment (addressed locals). A is a (universally quantified) type that is used to relate values (*e.g.*, of program variables) between pre- and postconditions P and Q. mpred is the type of predicates on (program logic) memory states. Section 7.2 will describe the relation of program logic memories (called rmaps) to CompCert's memories.

fundefs $\vec{f}$ is a list of function name, function definition (fundef) pairs. Each fundef is either an Internal function definition (a C function proved correct in this module) or an External function declaration:

$$\textbf{Inductive}\ \mathsf{fundef} : \mathsf{Type} \triangleq$$
$$|\ \mathsf{Internal} : \mathsf{function} \to \mathsf{fundef}$$
$$|\ \mathsf{External} : \mathsf{ident} \to \mathsf{typelist} \to \mathsf{type} \to \mathsf{fundef}$$

Internal functions are defined as in Section 3.2.1. Here is the corresponding Coq definition, which is for the most part self-explanatory:

$$f \in \textbf{Record}\ \mathsf{function} : \mathsf{Type} \triangleq$$
$$\{\ \mathsf{fn_return} : \mathsf{type};$$
$$\mathsf{fn_params} : \mathsf{list}\ (\mathsf{ident} * \mathsf{type});$$
$$\mathsf{fn_vars} : \mathsf{list}\ (\mathsf{ident} * \mathsf{type});$$
$$\mathsf{fn_temps} : \mathsf{list}\ (\mathsf{ident} * \mathsf{type});$$
$$\mathsf{fn_body} : \mathsf{statement}\ \}$$

type is the type of C types. fn_body is the actual body of the function (a C statement). The distinction between fn_vars and fn_temps is: the vars are addressed (and therefore stack-allocated), while temps are nonaddressed (and therefore allocated in registers, or spilled into the stack by the compiler).

External functions are just function type declarations, as one would see, *e.g.*, in a C header file.[1]

The inference rules of the semax_func judgment are:

$$\frac{}{V, \Gamma \vdash_{func} \mathsf{nil} : \mathsf{nil}} \qquad \text{(SemaxFuncNil)}$$

[1]This statement is a slight simplification. CompCert's external function type also models special-purpose functions such as compiler intrinsics.

$$\frac{\begin{array}{c} id \in \mathsf{map\ fst}\ \Gamma \qquad id \notin \mathsf{map\ fst}\ fs \\ \mathsf{var_sizes_ok}\ (\mathsf{fn_vars}\ f) \qquad \mathsf{precondition_closed}\ f\ P \\ V, \Gamma \vdash_{body} f : (id, \mathsf{mk_funspec}\ (\mathsf{fn_funsig}\ f)\ A\ P\ Q) \qquad V, \Gamma \vdash_{func} fs : \Gamma' \end{array}}{V, \Gamma \vdash_{func} (id, \mathsf{Internal}\ f) :: fs : (id, \mathsf{mk_funspec}\ (\mathsf{fn_funsig}\ f)\ A\ P\ Q) :: \Gamma'}\ (\textsc{SemaxFuncInternal})$$

$$\frac{\begin{array}{c} id \in \mathsf{map\ fst}\ \Gamma \qquad id \notin \mathsf{map\ fst}\ fs \qquad \mathsf{length}\ ids = \mathsf{length}\ \vec{\tau} \\ (\forall gx\ (x : A)\ (v_{ret} : \mathsf{option}\ \mathcal{V})\ \phi.\ Q\ x\ (\mathsf{make_ext_rval}\ gx\ v_{ret})\ \phi \implies \mathsf{welltyped}\ \tau\ v_{ret}) \\ \mathcal{O} \vdash_{ext} (ids, f_{id}) : (A, P, Q) \qquad V, \Gamma \vdash_{func} fs : \Gamma' \end{array}}{V, \Gamma \vdash_{func} (id, \mathsf{External}\ f_{id}\ \vec{\tau}\ \tau) :: fs : (id, \mathsf{mk_funspec}\ (\mathsf{zip}\ ids\ \vec{\tau}), \tau)\ A\ P\ Q) :: \Gamma'}\ (\textsc{SemaxFuncExternal})$$

SemaxFuncNil serves as the base case of a whole-module proof: the empty list of functions satisfies the empty list of specifications.

SemaxFuncInternal is the rule for verifying an Internal definition—a function defined in the current module. The key hypothesis is the subsidiary judgment:

$$V, \Gamma \vdash_{body} f : (id, \mathsf{mk_funspec}\ (\mathsf{fn_funsig}\ f)\ A\ P\ Q)$$

called semax_body, which states what it means for a function body to satisfy its specification. semax_body is defined in terms of Verifiable C's underlying Hoare judgment:

$$\begin{array}{l} V, \Gamma \vdash_{body} f : (id, \mathsf{mk_funspec}\ sig\ A\ P\ Q) \triangleq \\ \quad \forall x : A.\ \mathsf{func_tycontext}\ f\ V\ \Gamma \vdash \\ \qquad\qquad \{P\ x * \mathsf{stackframe_of}\ f\} \qquad \mathsf{fn_body}\ f \\ \qquad\qquad \{\mathsf{function_body_ret_assert}\ (\mathsf{fn_return}\ f)\ (Q\ x) \\ \qquad\qquad\quad * \mathsf{stackframe_of}\ f\}) \end{array}$$

func_tycontext constructs the appropriate Δ from f, V, and Γ. The predicate stackframe_of gives the shape of the stack (in memory) for function f, and is $*$-conjoined in both the pre- and postconditions. function_body_ret_assert binds the function return value in Q and ensures that the function returns properly (as opposed to, *e.g.*, **`break`**).

SemaxFuncExternal is the rule for "verifying" external functions. Why do we need such a rule? External functions are not associated with definitions (not, at least, in the module currently being verified). It stands to reason that we should be able to assume their specifications, *e.g.*, as axioms, at least for purposes of the current module.

The point is not to verify the functions themselves (that will be done when we verify the remaining program modules), but instead to ensure that the function specifications P, Q assumed in Γ for *this* module match those given by an external oracle. This matching is accomplished by judgment

$$\mathcal{O} \vdash_{ext} (ids, f_{id}) : (A, P, Q)$$

which reads "external oracle $\mathcal{O}$ justifies function specification P, Q" (A is the type of state shared between P and Q). $\mathcal{O}$ is only an implicit argument to the semax_func judgment; I do not write it explicitly in the conclusion of SemaxFuncExternal or the other inference rules.

The oracle is shared among all the modules in the program. In this way, it ensures that each module verification agrees on shared function specifications. At the same time, the external specification oracle is language- and (mostly) program-logic-independent, meaning it can be reused even in the proofs that connect Verifiable C to linking semantics and Compositional CompCert.

What is the shape of the oracle? Its type is:

```
𝒪 ∈ Record external_specification (M E Ω : Type) : Type ≜
  { ext_spec_type : E → Type;
    ext_spec_pre : ∀ ef : E.
      ext_spec_type ef → genviron →
      list typ → list 𝒱 → Ω → M → Prop
    ext_spec_post : ∀ ef : E.
      ext_spec_type ef → genviron →
      option typ → option 𝒱 → Ω → M → Prop;
    ext_spec_exit : option 𝒱 → Ω → M → Prop }.
```

The parameters are: M, the type of memory over which the external specification oracle quantifies; E, the type of external function names/declarations; Ω, the type of external oracle states. In Verifiable C, E is typically specialized to CompCert's external_function. The M parameter is variously CompCert's mem type or the type of *juicy memories*, which I introduce in Section 7.2.

The fields of the record are:

ext_spec_type: A function from external function names to the types of auxiliary state shared between their pre- and post-conditions. For an external function name ef, ext_spec_type ef is the analog of A in a Verifiable C funspec (A, P, Q).

ext_spec_pre: A (dependent) map from external function names to preconditions. The parameter of type ext_spec_type ef is dependent on the particular ef that is passed. In the interpretation of external specifications, this second parameter is shared between the pre- and postcondition. genviron is a map from global identifiers to the blocks at which they are allocated in memory. list typ are the expected (language-independent) types of the function arguments.[2]

ext_spec_post: A map from external function names to postconditions, analogous to ext_spec_pre. The argument of type option $\mathcal{V}$ is the (optional) return value to function ef.

ext_spec_exit: A predicate that must hold at module exit (*i.e.*, at return from module entry points).

With external specification oracles, one can extend the definition of safety given for closed programs (Section 5.2.2) to *open* programs, those that may call external functions. Like the definition of 5.2.2, open safeN is still indexed by a natural number n, which gives the number of steps for which we will interrogate the system. Unlike that previous definition, which was a pure safety condition, open safeN also imposes a postcondition (ext_spec_exit) at module halt—making open safeN a partial correctness property. The other major difference is the addition of the SafeN-External case for external function calls:

$$\frac{}{\mathsf{safeN}\ 0\ \omega\ c\ m} \quad (\textsc{SafeN-Zero})$$

$$\frac{ge \vdash c, m \longmapsto c', m' \qquad \mathsf{safeN}\ n\ \omega\ c'\ m'}{\mathsf{safeN}\ (n+1)\ \omega\ c\ m} \quad (\textsc{SafeN-Step})$$

[2]We do not use C types here because external specifications are intended to be language-independent. typs, as opposed to C types, include: int, float, long, single. The most recent versions of CompCert also include any32 and any64, for typing unknown data of fixed width.

$$\frac{\begin{array}{c}\text{at_external } c = \text{Some } (ef, \vec{v}) \\ \text{ext_spec_pre } \mathcal{O}\ ef\ x\ (\text{genv_symb } ge)\ (\text{sig_args } ef)\ \vec{v}\ \omega\ m \\ (\forall v_{ret}\ m'\ \omega'\ n'.\ n' \leq n\ \wedge\ R(n', m, m') \\ \wedge\ \text{ext_spec_post } \mathcal{O}\ ef\ x\ (\text{genv_symb } ge)\ (\text{sig_res } ef)\ v_{ret}\ \omega'\ m' \\ \Longrightarrow\ \exists c'.\ \text{after_external } v_{ret}\ c = \text{Some } c'\ \wedge\ \text{safeN } n'\ \omega'\ c'\ m')\end{array}}{\text{safeN } (n+1)\ \omega\ c\ m}$$

(SafeN-External)

$$\frac{\text{halted } c = \text{Some } v_{ret} \qquad \text{ext_spec_exit } \mathcal{O}\ (\text{Some } v_{ret})\ \omega\ m}{\text{safeN } n\ \omega\ c\ m}$$

(SafeN-Halted)

The generalized safeN for open programs is a predicate of type $\mathbb{N} \to \Omega \to C \to M \to$ Prop. The $n : \mathbb{N}$ is the number of steps for which configuration $\langle c, m \rangle$ is safe. ω is the external state of the oracle, or outside world.

Most of the rules are self-explanatory, by reference to the definition of closed safeN in Section 5.2.2. New is the rule for external function calls, SafeN-External. It handles the case in which a core state c is at external, calling external function ef with arguments $\vec{v}$. In this situation, we say $\langle c, m \rangle$ is safe for $n+1$ steps when:

- $\vec{v}$, ω, and m satisfy ef's precondition (ext_spec_pre $\mathcal{O}\ ef\ x \ldots$); and
- for all return values v_{ret}, new memory states m', new external states ω', and naturals n' such that
  - $n' \leq n$,
  - n' and m' are related to m by a particular relation R (which I will explain in a moment), and
  - v_{ret}, ω', and m' satisfy ef's postcondition,

  running after_external c to inject the return value v_{ret} results in a new state c' that is safe for n' steps in ω' and m'.

The intuition for SafeN-External is: A configuration $\langle c, m \rangle$, calling external function ef, is safe for $n+1$ steps when it is safe for $n' \leq n$ steps in *any* state the external world may return after executing ef, as long as the returned state satisfies the postcondition agreed upon in $\mathcal{O}$. The state exposed by $\langle c, m \rangle$ to the outside world, at the point of the call (the memory m and the function arguments $\vec{v}$), must satisfy ef's precondition as well.

The relation R used above differs depending on the type M at which safeN is parameterized. When M equals CompCert's mem, R is just $\lambda n'\ m\ m'.$ True (meaning we must prove safety, over external calls, for all $n' \leq n$). I call this definition "dry safety" (dry_safe) as opposed to "juicy safety" (juicy_safe), for reasons that will become apparent in Section 7.2.

When M is the type of juicy memories (upcoming, in Section 7.2), R is specialized to:

$R\ n'\ m\ m' \triangleq$
$\quad n' = \mathsf{level}\ m' \wedge \mathsf{level}\ m' < \mathsf{level}\ m$
$\quad \wedge$ (* ...a relation on the function specifications embedded in m, m'. *)

Safety specialized to this R is called "juicy safety." The natural n' must equal the step index of juicy memory m'; also, the step index of m' must be strictly less than that of m. These two conditions are step-index-related: If n' (the number of steps for which we must prove safety in StepN-External) were greater than level m', then it would be possible for level $m' = 0$ (the "we have given up" state), while $n' > 0$ forces us to continue to prove safety for some nonzero number of remaining steps n'.

7.2 Juicy Memories

The semax_func judgment I outlined in the previous section gives a proof theory for C programs. How do we know this theory is sound? When we prove $V, \Gamma \vdash_{func} \vec{f} : \Gamma$, for funspecs Γ and functions $\vec{f}$, who guarantees that the functions $\vec{f}$ actually satisfy their specifications? Or that, for a given $f \in \vec{f}$, if we initialize f in a state satisfying its precondition, it will either safely run forever[3] or halt in a state satisfying its postcondition?

The answer, as Part VI of [ADH+14] demonstrates, is to construct a semantic model of the Hoare judgment, and then prove soundness with respect to this model. In this section, I briefly describe enough of the underlying machinery to explain how the semantic model of the Verifiable C logic is connected to Compositional CompCert (Section 7.3).

Juicy Memories. When reasoning in a program logic, step indexes are unproblematic: the step indexes can often be hidden via use of the $\triangleright$ operator, and do not often appear explicitly in assertions.

How to connect step-indexed states to CompCert's memories (Chapter 2), which are not step-indexed? One could simply step-index CompCert. But this strategy makes it difficult, at least naively, to prove correctness of compiler phases that may change the number of steps. A better solution is to stratify the models into two layers: operational *states* corresponding to states of the operational semantics, and semantic *worlds* appearing in assertions of the program logic. *Juicy memories* are the specialization of this

[3] *E.g.*, with respect to the definition of safety just given.

strategy to Verifiable C rmaps (resource maps, Verifiable C's step-indexed model of the state) and CompCert memories.

To a first approximation, a juicy memory jm defines what it means for an rmap ϕ to *erase* to a CompCert memory m. By erasure, we mean the removal of the "juice" that is unnecessary for execution (as in Curry-style type erasure of simply typed lambda calculus). The "juice" has several components: *permission shares* controlling access to objects in the program logic; *predicates in the heap* describing invariants of objects in the program logic; and the classification of certain addresses as values, locks, function pointers, *etc.*

Resource maps, or just rmaps for short, are the program-logic counterpart to CompCert memories. Their type is abstract (hidden behind a Coq module interface in file `VST/veric/rmaps.v`), but to a first approximation think of rmaps as maps from CompCert addresses (block–offset pairs) to *resources*, where resources generalize CompCert memvals (abstract bytes).

$$\phi \in \mathsf{rmap} \approx \mathsf{address} \to \mathsf{resource}$$

I say "only to a first approximation" because rmap resources will contain assertions (such as function specifications and lock invariants) that may quantify over the rmap itself. Thus rmaps are not defined directly as above, but instead using step indexing. Chapter 39 of [ADH+14] gives more detail. A paper by Hobor, Dockins, and Appel [HDA10] explains the particular technique used (indirection theory). For purposes of this thesis, the step-indexed details of the Verifiable C model are not critically important. When I wish to indicate that rmap ϕ contains resource res at location l, I will use syntax $\phi \mathbin{@} l = res$.

The resources themselves are defined inductively as:

$$\begin{array}{l} res \in \textbf{Inductive}\ \mathsf{resource} : \mathsf{Type} \triangleq \\ \quad \mid \mathsf{NO} : \mathsf{resource} \\ \quad \mid \mathsf{YES} : \mathsf{pshare} \to \mathsf{kind} \to \mathsf{preds} \to \mathsf{resource} \\ \quad \mid \mathsf{PURE} : \mathsf{kind} \to \mathsf{preds} \to \mathsf{resource}. \end{array}$$

A NO resource indicates no access to a location. $\phi \mathbin{@} l = \mathsf{YES}\ \pi\ k\ pp$ asserts a resource of kind k with program-logic permission π and (optional) predicates pp at location l. pshare stands for *positive* (*i.e.*, nonzero) share.

$$\begin{array}{l} k \in \textbf{Inductive}\ \mathsf{kind} : \mathsf{Type} \triangleq \\ \quad \mid \mathsf{VAL} : \mathsf{memval} \to \mathsf{kind} \\ \quad \mid \mathsf{LK} : \mathsf{Z} \to \mathsf{kind} \\ \quad \mid \mathsf{CT} : \mathsf{Z} \to \mathsf{kind} \\ \quad \mid \mathsf{FUN} : \mathsf{funsig} \to \mathsf{kind}. \end{array}$$

The kinds are either values VAL, for CompCert memval, function specifications FUN, which specify function pointers, or the special LK/CT kinds, which indicate that a particular (series of) locations in memory serves as a semaphore.[4]

The PURE resource is used primarily to store FUN kinds, with the predicates pp, which may quantify over the rmap itself, storing the function pre- and postconditions. YES is used primarily to represent actual bytes in memory.

For example, the program-logic representation of a CompCert memval v with Freeable permission is:

$$\mathsf{YES}\ \top\ (\mathsf{VAL}\ v)\ \mathsf{NoneP}$$

where $\top$ is the topmost share in Verifiable C's permission lattice and NoneP represents the empty list of predicates.

Juicy Memories: Implementation. In `veric/juicy_mem.v`, we define juicy memories as pairs of a memory m and an rmap ϕ. The rmap and memory must be *consistent with each other*, in a way we will make precise in a moment. In the code, we represent this pair with the following inductive type.

```
Inductive juicy_mem : Type ≜
  mkJuicyMem : ∀ (m: mem) (φ: rmap)
     (JMcontents : contents_cohere m φ)
     (JMaccess : access_cohere m φ)
     (JMmax_access : max_access_cohere m φ)
     (JMalloc : alloc_cohere m φ).
   juicy_mem.
```

We equip the type juicy_mem with accessor functions of the form

```
m_dry (jm: juicy_Mem) ≜ case jm of mkJuicyMem m _ _ _ _ _  → m
m_phi (jm: juicy_Mem) ≜ case jm of mkJuicyMem _ φ _ _ _ _  → φ
```

The four proof objects beginning JM ... enforce the four consistency requirements:

[4] LK/CT are used only in the Verifiable C extension to Concurrent Separation Logic. LK is the resource kind associated with the first byte of a 4-byte lock; the remaining three bytes of every lock contain CT kinds.

Contents. If $\phi @ l = (\mathsf{YES}\ \pi\ (\mathsf{VAL}\ v)\ pp)$ then $m\ l = v$ and $pp = \mathsf{NoneP}$. That is, a VAL in the rmap must have no "predicates in the heap" associated with it, and the v in the rmap must match the v in the CompCert memory. Predicates will only occur in PUREs, to give function specifications, and in locks $(\mathsf{YES}\ \pi\ (\mathsf{LK}\ k)\ (\mathsf{SomeP}\ \mathsf{R}))$ to give resource invariants.

Access. For all locations l, $m\ l = \mathsf{perm_of_res}\ (\phi @ l)$. The fractional share $\phi @ l$ must "erase" to that location's CompCert memory permission. perm_of_res is a simple function that erases Verifiable C's fractional shares to CompCert-style permissions. Chapter 42 of [ADH+14] provides additional justification. In particular, it explains why erasing to coarse-grained permissions when connecting to CompCert makes sense for Pthreads-style concurrency.

Max Access. For all locations l,

$\mathsf{max_access_at}\ m\ l \sqsupseteq \mathsf{perm_of_sh}\ \pi$	when $\phi @ l = \mathsf{YES}\ \pi$
$\mathsf{max_access_at}\ m\ l \sqsupseteq \mathsf{perm_of_sh}\ \bot$	when $\phi @ l = \mathsf{NO}$
$\mathsf{fst}\ l < \mathsf{nextblock}\ m$	when $\exists f\ pp.\ \phi @ l = \mathsf{PURE}\ f\ pp$.

Alloc. For all locations l, if $\mathsf{fst}\ l \geq \mathsf{nextblock}\ m$ then $\phi @ l = \mathsf{NO}$. CompCert treats addresses whose abstract base pointer is beyond nextblock as not-yet-allocated. Here we ensure that ϕ makes no claim to those addresses.

The juicy-memory consistency requirements are mostly straightforward. **Max Access** is a bit more complicated. It does case analysis on the resource $\phi @ l$, ensuring that the *maximum* permission in m at a given location is greater than or equal to the permission corresponding to the shares π or $\bot$. As I explained in Chapter 2, maximum permissions are a technical device used in version 2 of CompCert's memory model to express invariants useful for optimizations like constant propagation. The *current permission* in m at location l, or just *permission*, is always less than the maximum permission. When $\phi @ l$ contains a PURE resource, **Max Access** just ensures that l is a location that was allocated at some point ($\mathsf{fst}\ l < \mathsf{nextblock}\ m$). Here $\mathsf{nextblock}\ m$ is the next block in CompCert's internal free list, as in Chapter 2.

The consistency requirements together ensure that assertions expressed in the Hoare logic on the ϕ portion of the juicy memory actually say something about the CompCert memory m. For example, suppose we know—perhaps because ϕ satisfies the assertion $l \overset{\pi}{\mapsto} v$—that ϕ contains the value v with share π at location l. Then, in order to prove that a load from m

at location l will succeed, we would also like to be able to show that m contains v at l, with at least readable permission.

To validate that the consistency requirements described above satisfy laws of this form, we prove such a lemma for each of the basic CompCert memory operations: load, store, alloc, and free. For example, here is the lemma for mapsto with writable share.

Lemma mapsto_can_store :
  $\forall$ $ch\ v\ b\ z\ jm\ v'$.
    (address_mapsto $ch\ v \top (b, z)$ $*$ TT) (m_phi jm) $\implies$
    $\exists m'$, store ch (m_dry jm) $b\ z\ v' =$ Some m'.

This lemma relies on the consistency requirements to prove that the store in m_dry jm will succeed. The lemmas for the other memory operations differ in the predicate on m_phi jm but are otherwise similar.

In addition to "progress" lemmas of the form mapsto_can_store, we prove "preservation" lemmas for juicy memories. That is, we would like to know that after each CompCert memory operation on m_dry jm, yielding a new memory m', it is possible to construct a new juicy memory jm' such that m_dry $jm' = m'$. The intuition here is that memory operations on m_dry jm never touch the *hidden* parts of m_phi jm, *e.g.*, the function specifications and lock invariants appearing in Hoare logic assertions. Thus it is possible to construct jm' generically from m' and m_phi jm, by copying hidden data unchanged from m_phi jm to m_phi jm', and by updating m_phi jm' at those locations that were updated by the memory operation.

For example, the function after_alloc$'$ defines the map underlying the new m_phi jm' after an allocation alloc (m_dry j) $lo\ hi$.

after_alloc$'$ ($lo\ hi$: Z) (b: block) (ϕ: rmap) (H: $\forall z.\ \phi$ @ (b,z) = NO)
  : address $\rightarrow$ resource $\triangleq$ fun l $\rightarrow$
    if adr_range_dec (b, lo) ($hi - lo$) l
      then YES $\top$ pfullshare (VAL Undef) NoneP
      else phi @ l.

Then the lemma

Lemma juicy_mem_alloc_at :
  $\forall jm\ lo\ hi\ jm'\ b$. juicy_mem_alloc $jm\ lo\ hi = (jm', b)$ $\implies$
  $\forall$ l. m_phi jm' @ l = **if** adr_range_dec (b, lo) ($hi - lo$) l
                  then YES $\top$ pfullshare (VAL Undef) NoneP
                  else m_phi jm @ l.

Semantics G C juicy_mem

$$\begin{aligned}
\text{initial_core} &\triangleq \text{initial_core } sem \\
\text{at_external} &\triangleq \text{at_external } sem \\
\text{after_external} &\triangleq \text{after_external } sem \\
\text{halted} &\triangleq \text{halted } sem
\end{aligned}$$

corestep ge $(c : C)$ $(jm$: juicy_mem$)$ $(c'$: C$)$ $(jm'$: juicy_mem$)$: Prop $\triangleq$
  corestep sem ge c (m_dry jm) c' (m_dry jm')
  $\wedge$ resource_decay (nextblock (m_dry jm)) (m_phi jm) (m_phi jm')
  $\wedge$ level jm $=$ level jm' $+$ 1.

Figure 7.1: Juicy interaction semantics, parameterized by an underlying semantics sem : Semantics G C mem.

gives an extensional definition of the contents of the juicy memory jm' that results. Here juicy_mem_alloc uses after_alloc$'$ to construct the new juicy memory jm' resulting from the allocation.

Juicy Semantics. It is possible to take the Clight semantics I presented in Chapter 3, which operated on CompCert memories, and *lift* it to operate on juicy memories instead. In fact, this process can be replicated for any interaction semantics operating on CompCert memories (Figure 7.1). Here is the generic construction (file `VST/veric/juicy_extspec.v`):

Assume as input an interaction semantics sem : Semantics G C mem operating on CompCert memories. We will construct a new interaction semantics, $\mathcal{J}(sem)$, by defining the new juicy step relation jstep:

jstep G C $(sem$: CoreSemantics G C mem$)$ $(ge : G)$
  $(c : C)$ $(jm$: juicy_mem$)$ $(c'$: C$)$ $(jm'$: juicy_mem$)$: Prop
  $\triangleq$ corestep sem ge c (m_dry jm) c' (m_dry jm')
    $\wedge$ resource_decay (nextblock (m_dry jm)) (m_phi jm) (m_phi jm')
    $\wedge$ level jm $=$ level jm' $+$ 1.

The new jstep relation embeds the corestep relation of the underlying semantics, projected to the m_dry components of the initial and final juicy memories jm and jm'. In addition, it asserts that

- m_phi jm' is resource_decayed from m_phi jm'; and
- the level, or age, of jm' is one less than the age of jm (stepping reduces the step index by one).

The resource_decay relation is a bit more complicated:

$$
\begin{array}{l}
\textsf{resource_decay}\ (\mathit{nextb} : \textsf{block})\ (\phi_1\ \phi_2 : \textsf{rmap}) \triangleq \\
\quad \textsf{level}\ \phi_1 \geq \textsf{level}\ \phi_2\ \wedge \\
\quad \forall\ l : \textsf{address}.\ \textsf{fst}\ l \geq \mathit{nextb} \implies \phi_1\ @\ l = \textsf{NO}\ \wedge \\
\qquad\quad (*\ \textsf{either no change, up to step indices}\ *) \\
\qquad\quad \textsf{resource_fmap}\ (\textsf{approx}\ (\textsf{level}\ \phi_2))\ (\phi_1\ @\ l) = (\phi_2\ @\ l) \\
\\
\qquad\quad (*\ \textsf{or write at location}\ l\ *) \\
\qquad \vee\ (\exists\ v\ v'.\ \textsf{resource_fmap}\ (\textsf{approx}\ (\textsf{level}\ \phi_2))\ (\phi_1\ @\ l) \\
\qquad\qquad\qquad = \textsf{YES}\ \top\ (\textsf{VAL}\ v)\ \textsf{NoneP} \\
\qquad\qquad \wedge\ \phi_2\ @\ l = \textsf{YES}\ \top\ (\textsf{VAL}\ v')\ \textsf{NoneP}) \\
\\
\qquad\quad (*\ \textsf{or}\ l\ \textsf{newly allocated in}\ \phi_2\ *) \\
\qquad \vee\ \textsf{fst}\ l \geq \mathit{nextb} \wedge \exists\ v.\ \phi_2\ @\ l = \textsf{YES}\ \top\ (\textsf{VAL}\ v)\ \textsf{NoneP} \\
\\
\qquad\quad (*\ \textsf{or}\ l\ \textsf{freed in}\ \phi_2\ *) \\
\qquad \vee\ \exists\ v\ pp.\ \phi_1\ @\ l = \textsf{YES}\ \top\ (\textsf{VAL}\ v)\ pp \wedge \phi_2\ @\ l = \textsf{NO}.
\end{array}
$$

The relation gives an extensional interpretation of the kinds of memory effects that may occur over steps from rmap $\phi_1 = \textsf{m_phi}\ jm$ to $\phi_2 = \textsf{m_phi}\ jm'$. At every location l, either $\phi_1\ @\ l = \phi_2\ @\ l$ (up to step-indexing, resource_fmap, *etc.*), or there was a write at l, or l was newly allocated in ϕ_2, or l was freed. resource_decay is often used, in the construction of the Verifiable C model, to reason by cases on the Clight jstep relation. Its justification is the fact that all operational semantics that respect the CompCert memory model's interface behave in this way.

Whole-Module Correctness. Now that I've introduced open-program safety, juicy memories, and the $\mathcal{J}$ operator for lifting standard CompCert interaction semantics such as the one for Clight to their juicy counterparts, I can finally state the correctness theorem for single-module proofs in the logic. It is:

Theorem 8 (Soundness of Judgment $V, G \vdash_{func} mod : G$).

```
1  ∀mod V G ω jm f f_id f_b f_body v⃗.
2  let ge ≜ globalenv mod in
3  let O ≜ funspecs_of G in
4  let τ⃗ ≜ sig_args f in
5  let τ ≜ sig_res f in
6  let sem ≜ J(CL_Sem) in
7  V, G ⊢_func mod : G ⟹
8  fun_id f = Some f_id ⟹
9  find_symbol ge f_id = Some f_b ⟹
10 find_funct ge (Vptr f_b Int.zero) = Some f_body ⟹
11 ∀x : ext_spec_type O f. ext_spec_pre O f x (genv_symb ge) τ⃗ v⃗ ω jm
12  ⟹ ∃c. initial_core sem ge (Vptr f_b Int.zero) v⃗ = Some c
13        ∧ juicy_safe sem (O with {ext_spec_exit ≜ ext_spec_post O f})
14                     ge ω c jm
```

Proof. The theorem is a corollary of the interpretation of semax_func.[5] The machine-checked proof is still in progress. □

Essentially, for each function $f \in mod$, if we initialize mod at entry point f in a state satisfying f's precondition, then the module will either (safely) infinite loop, under the definition of safety for open programs given in this chapter, or terminate in a state satisfying f's postcondition.

In detail, given

- a program module mod,
- variable and global function specifications V and G,
- external state ω,
- juicy memory memory jm,
- function f, and
- arguments $\vec{v}$

if f is defined by module mod (lines 8 to 10), and $\vec{v}$, ω, and jm satisfy f's precondition as given by the function specification $\mathcal{O}$ (which I will explain in a moment), then initializing the module at entry point f results in an initial core state c that is (juicy) safe in the specification that updates $\mathcal{O}$ to have ext_spec_exit predicate equal f's postcondition.

How is $\mathcal{O}$ defined? Take the empty set of specifications and add to it the function specifications given in G. For example, if G associates

[5] File `VST/veric/semax_prog.v`.

funspec (A, P, Q) with f_{id}, then ext_spec_pre (funspecs_of G) will map f to P, ext_spec_post (funspecs_of G) will map f to Q, and so on, for the other function specifiers in G.[6]

7.3 Composing End-to-End

How do we compose Theorem 8 with the results in Chapter 6, on verified separate compilation?

Figure 7.2 depicts the general strategy. At the top of the diagram, the user has proved a number of program-logic semax_func judgments of the form

$$V, G \vdash_{func} \; mod_{idx} \; : \; G$$

for each module index idx, with respect to the semantics $\mathcal{J}(\mathsf{CL_{Sem}})$. As I will prove in this section, these independent proofs, with respect to the shared function specifiers G, yield (open-program) safety with respect to the linked semantics

$$\mathcal{L}([\![mod_0]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, [\![mod_1]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, \ldots, [\![mod_{N-1}]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})})$$

$\mathcal{L}$ is the linking operator of Chapter 4, made parametric over the type of memories M shared by the semantics, here juicy memories.[7]

The linked program $\mathcal{L}(\ldots)$ may still call external functions—those specified by the oracle $\mathcal{O}$ but not defined by any of the modules 0 to $N-1$. The next step is to close over these external functions, by constructing a Gallina context $\mathcal{J}(C)$ (*cf.* Section 4.3) that "executes" the exteral functions, in the manner of Section 4.3. We must also ensure, for the next step, that $\mathcal{J}(C)$ is erasable, by which I mean the predicates "checked" by C are erasable to

[6]See `VST/veric/semax_ext.v` for the details. In that file, funspecs_of is called add_funspecs.

[7]Linking semantics as defined in Chapter 4 was specialized to CompCert memories, for concreteness. However, CompCert memories were by no means essential. The definition of parametric linking semantics is given in file `VST/linking/linking.v`.

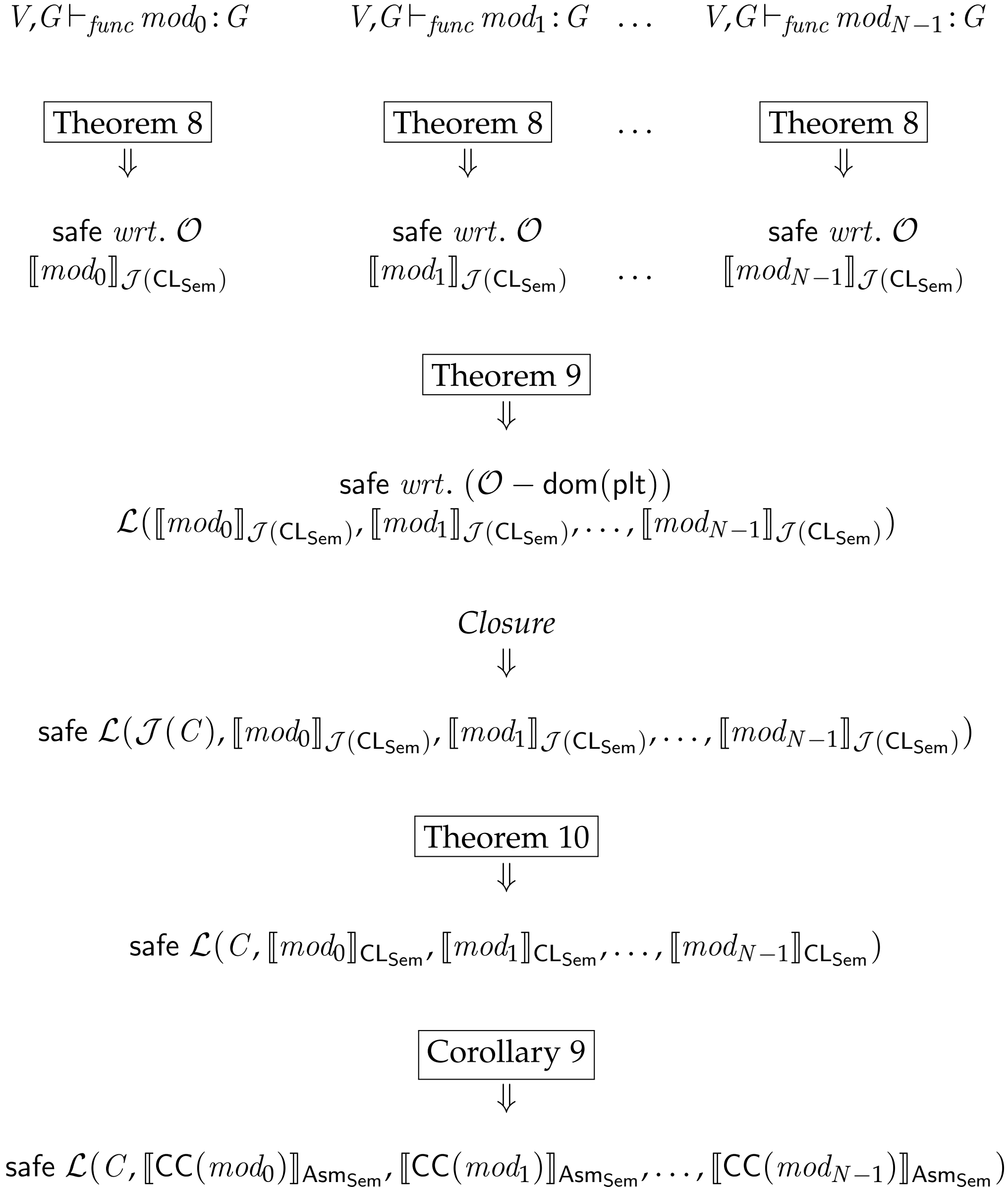


Figure 7.2: Composing the proofs. CC abbreviates CompCert. $\mathcal{O}$ is funspecs_of G. C is a program context compatible with $(\mathcal{O} - \mathsf{dom}(\mathsf{plt}))$, as described in the text.

the m_dry component of juicy memories.[8] This process gives us safety of

$$\mathcal{L}(\mathcal{J}(C), [\![mod_0]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, [\![mod_1]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, \ldots, [\![mod_{N-1}]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})})$$

Finally, we erase the juicy linked semantics, resulting in a proof of safety for semantics

$$\mathcal{L}(C, [\![mod_0]\!]_{\mathsf{CL_{Sem}}}, [\![mod_1]\!]_{\mathsf{CL_{Sem}}}, \ldots, [\![mod_{N-1}]\!]_{\mathsf{CL_{Sem}}})$$

in which C is the erasure of context $\mathcal{J}(C)$, and each module $mod_1 \ldots mod_N$ is now interpreted in the "dry" Clight semantics $\mathsf{CL_{Sem}}$ against which Compositional CompCert is proved correct.

As long as C is a well-defined context (Definition 8, Chapter 6), the results of Chapter 6 give us that C is safe when linked with the (independently) compiled assembly modules $\mathsf{CompCert}(mod_1), \ldots, \mathsf{CompCert}(mod_N)$, assuming compilation succeeds for each of the mod_1 through mod_N.

7.3.1 Safely Linking

Proving the erasure theorem I described above is relatively easy. Proving safety of the linked semantics, from the per-module program-logic proofs, requires a bit more ingenuity. Here is the statement of the main theorem:

---

[8] I have not yet implemented closure as a general theorem in Coq. The simplest strategy—and the one supported by the current erasure proofs—is to require that the specifications of those functions that "escaped" implementation by $mod_1 \ldots mod_N$ be expressible purely on the dry part of the state, *i.e.* the CompCert memory. These dry specifications can then be lifted to operate on juicy memories as was done for juicy semantics. "Juicy" specifications can still be used in the program logic to prove module specifications. In ongoing work on concurrency, I have had initial successes with erasure of more complex specifications.

Theorem 9 (Linking Safety)**.**

1 $\forall N\ plt\ (mod : I_N \to \mathsf{Prog}_{\mathsf{CL}})\ ge\ main\ id_{main}\ V\ G.$
2 **let** $\mathcal{O} \triangleq$ *funspecs_of* G **in**
3 **let** $\mathcal{O}' \triangleq \mathcal{O}$ **with** $\{$*ext_spec_exit* $\triangleq$ *ext_spec_post* $\mathcal{O}\ main\}$ **in**
4 **let** $sems \triangleq \lambda idx : I_N.$ mkModsem $\mathcal{J}(\mathsf{CL_{Sem}})$ *(globalenv* mod_{idx}*)* **in**
5 *(∗ Postconditions* in $\mathcal{O}$ *imply* return *values are well−typed... ∗)* $\Longrightarrow$
6 *(∗ The* plt is *well−formed wrt. global environments... ∗)* $\Longrightarrow$
7 *(∗ Modules contain equal global symbol tables* genv_symb *∗)* $\Longrightarrow$
8 $(\forall idx : I_N.\ V, G \vdash_{func}\ mod_{idx}\ :\ G) \Longrightarrow$
9 $\forall x\ \omega\ jm\ idx_{main}\ b_{main}\ \vec{v}.$
10 *ext_spec_type* $\mathcal{O}\ main =$ *unit* $\Longrightarrow$
11 *fun_id* $main = id_{main} \Longrightarrow$
12 *plt* $id_{main} =$ *Some* $idx_{main} \Longrightarrow$
13 *find_symbol (ge* $(sems_{idx_{main}})$*)* $id_{main} =$ *Some* $b_{main} \Longrightarrow$
14 *ext_spec_pre* $\mathcal{O}\ main\ x$ *(genv_symb* ge*) (sig_args* $main$*)* $\vec{v}\ \omega\ jm$
15 $\Longrightarrow \exists l$: *LinkedState* $N\ sems.$
16 *initial_core* ge *(Vptr* b_{main} *Int.zero)* $\vec{v}$ = *Some* l
17 $\wedge$ *safeN* $(\mathcal{O}' -$ *dom(plt))* ge *(level* jm*)* $\omega\ l\ jm.$

$\mathsf{Prog}_{\mathsf{CL}}$ is the type of Clight programs (program Clight.fundef type in the Coq code). Overall, the theorem states that if we have proved each module correct in the logic (line 8), then for an entry point $main$, any juicy memory state jm (and initial arguments $\vec{v}$, and external state ω, *etc.*), if jm satisfies the precondition for $main$ (line 14), then initializing the linked semantics at $main$ succeeds (line 16) and the initial state is safe for level-of-jm steps. Why is safe-for-level-of-jm-steps sufficient, as opposed to safe-for-all-n? We have proved[9] that for any CompCert initial memory m, we can construct a matching juicy memory jm such that m_dry $jm = m$, with arbitrary initial level n. Safety is with respect to specification $\mathcal{O}' - \mathsf{dom}(\mathsf{plt})$, the function specifications $\mathcal{O}'$ minus pre-/post-conditions for the implemented functions (those in the domain of plt).

The three assumptions I have elided are:

Postconditions imply well-typed return values. For each function f specified by $\mathcal{O}$, f's postcondition implies that the values f returns are well-typed (with respect to f's signature). This property is required for compiler correctness—*e.g.*, in register allocation, to determine in

[9]Definition `initial_jm` in file `VST/veric/initial_world.v`.

which class of registers to stick return values. It shows up here because the property is built in, operationally, in linking semantics $\mathcal{L}$.

The plt is well-formed. Whenever plt f_{id} = Some idx (that is, the plt claims that f_{id} is implemented by module idx), then module idx's global environment contains a binding for f.

Modules contain equal global symbol tables genv_symb. Module global environments contain equal symbol tables (genv_symb, mapping global identifiers to the addresses at which they are allocated).

This assumption is not unrealistic. Assume we have two independent proofs, in the Verifiable C logic, that modules mod_1 and mod_2 are safe with respect to specification oracle $\mathcal{O}$. If mod_1 and mod_2 declare different (but consistent) sets of global identifiers, we can pre-process mod_1 and mod_2 to include the exact same sets of global variable and function declarations, in the same order, without invalidating the associated Verifiable C proof scripts (these can be re-run unchanged in the larger global context).

Proof. The proof[10] depends on Theorem 8. The main difficulty, as in the proofs of the linking theorems of Chapter 6, is in devising an invariant on the stack-of-cores runtime states of linking semantics that is strong enough to prove safety of the overall linked program. Unlike in Chapter 6, which employed binary invariants on source–target linked states, the invariant here is unary (applies to a single program state). The invariant all_safe (defined in `VST/linking/safety.v`) has shape:

$$\mathsf{all_safe} : \Omega \to \mathsf{LinkedState}\ N\ mod \to \mathsf{juicy_mem} \to \mathsf{Prop}$$
$$\mathsf{all_safe}\ \omega\ l\ jm$$

and is defined:

$$\begin{array}{l} \mathsf{all_safe}\ \omega\ (l : \mathsf{LinkedState})\ jm \triangleq \\ \quad \exists fs : \mathsf{list}\ (\mathsf{ident{*}typelist{*}type}). \\ \qquad \mathsf{last_frame_main}\ fs \wedge \mathsf{stack_safe}\ fs\ (\mathsf{stack}\ l)\ \omega\ jm \end{array}$$

Existentially quantified is fs, the stack-trace of functions that have been called up to this point. last_frame_main, the first conjunct of the invariant, asserts that the bottom-most function in the stack is main.

[10]File `VST/linking/safety.v` states the safety invariant and contains the majority of the proof; `VST/linking/semax_linking.v` applies the theorem to the Verifiable C logic.

$$\begin{array}{l} \textsf{last_frame_main}\ fs \triangleq \\ \quad \textbf{case}\ fs\ \textbf{of} \\ \quad\quad \mid \textsf{nil} \rightarrow \textsf{True} \\ \quad\quad \mid f :: \textsf{nil} \rightarrow f = \textsf{main} \\ \quad\quad \mid _ :: fs' \rightarrow \textsf{last_frame_main}\ fs' \end{array}$$

stack_safe asserts safety of the current stack of cores s by distinguishing the head core c (on top of the stack) from the remaining cores s', all of which are at_external:

$$\begin{array}{l} \textsf{stack_safe}\ fs\ s\ \omega\ jm : \textsf{Prop} \triangleq \\ \quad \textbf{case}\ fs,\ s\ \textbf{of} \\ \quad\quad \mid \textsf{nil},\ \textsf{nil} \rightarrow \textsf{True} \\ \quad\quad \mid f :: fs',\ c :: s' \rightarrow \\ \quad\quad\quad\quad \exists x : \textsf{ext_spec_type}\ \mathcal{O}\ f. \\ \quad\quad\quad\quad\quad \textsf{head_safe}\ f\ x\ c\ \omega\ jm \wedge \textsf{tail_safe}\ f\ x\ fs'\ s'\ jm \\ \quad\quad \mid _,\ _ \rightarrow \textsf{False} \end{array}$$

Safety of the topmost core c (head_safe) is defined:

$$\begin{array}{l} \textsf{head_safe}\ f\ (x : \textsf{ext_spec_type}\ \mathcal{O}\ f)\ c\ \omega\ jm \triangleq \\ \quad \textbf{let}\ idx \triangleq c.\textsf{idx}\ \textbf{in} \\ \quad \textbf{let}\ ge \triangleq \textsf{ge}\ sems_{idx}\ \textbf{in} \\ \quad \textbf{let}\ sem \triangleq \textsf{sem}\ sems_{idx}\ \textbf{in} \\ \quad \textbf{let}\ \mathcal{O}' \triangleq \mathcal{O}\ \textbf{with}\ \{\textsf{ext_spec_exit} \triangleq \textsf{ext_spec_post}\ \mathcal{O}\ f\}\ \textbf{in} \\ \quad\quad \textsf{safeN}\ sem\ \mathcal{O}'\ ge\ (\textsf{level}\ jm)\ \omega\ c\ jm \end{array}$$

Recall that each module semantics $sems_{idx}$ defines two projections, ge for the global environment associated with module idx and sem for the interaction semantics of idx. head_safe states that state c is safe, for level jm steps, with respect to its associated semantics and global environment.

The predicate tail_safe is a bit more involved:

```
1  tail_safe f x fs s jm ≜ case fs, s of
2  | nil, nil → True
3  | f_top :: fs', c :: s' →
4    let idx ≜ c.idx in
5    let ge ≜ ge sems_idx in
6    let sem ≜ sem sems_idx in
7    let O' ≜ O with {ext_spec_exit ≜ ext_spec_post O f_top} in
8    tail_safe f_top x_top fs' s' jm ∧
9    ∃v⃗ ω jm_0 (x_top : ext_spec_type O f_top).
10     R (level jm) jm_0 jm
11     ∧  at_external sem c = Some (ef, v⃗)
12     ∧ ext_spec_pre O ef x (genv_symb ge) (sig_args ef) v⃗ ω jm_0
13     ∧ (∀v_ret jm' ω'.
14          R (level jm') jm_0 jm'  ⟹
15          ext_spec_post O ef x (genv_symb ge) (sig_res ef) v_ret ω' jm'  ⟹
16          ∃c'. after_external sem v_ret c = Some c'
17            ∧  safeN sem O' ge (level jm') ω' c' jm')
18 | _, _ → False
```

The invariant is defined recursively on fs and s. It states that, for each core c in s (recall that the s here is the tail of the overall linked-program stack), (i) c is at_external (line 11), and (ii) c is safe for any return value v_{ret} and states jm', ω' with which the environment may return (line 17), assuming the return values/states are in relation R and satisfy the postcondition of function ef (lines 14 and 15). The relation R is the same as that defined in Section 7.1.2 (essentially, level $jm <$ level jm_0).

Once the all_safe invariant has been defined, the brunt of the work of the proof is to show the following progress/preservation property:

Lemma all_safe_invariant ω (l : LinkedState N mod) jm :
(* if invariant holds initially, then *)
all_safe $\omega\ l\ jm \implies$ level $jm > 0 \implies$

(* either (i), linked semantics takes a corestep and invariant is reestablished *)
$\exists l'\ jm'.\ ge \vdash l, jm' \Longmapsto l', jm' \wedge$ **all_safe** $\omega\ l'\ jm'$

(* or (ii), semantics is halted *)
$\vee\ \exists v_{ret}.$ halted l = Some v_{ret}

(* or (iii), can reestablish invariant over external calls *)
$\vee\ \exists ef\ \vec{v}.$ at_external l = Some $(ef, \vec{v})$
$\wedge\ \exists x$: ext_spec_type $\mathcal{O}\ ef$.
ext_spec_pre $\mathcal{O}\ ef\ x$ (genv_symb ge) (sig_args ef) $\vec{v}\ \omega\ jm$
$\wedge\ \forall v_{ret}\ jm'\ \omega'\ n''.$
$n'' \leq$ level $jm' \implies$
R (level jm') $jm\ jm' \implies$
ext_spec_post $\mathcal{O}\ ef\ x$ (genv_symb ge) (sig_res ef) $v_{ret}\ \omega'\ jm' \implies$
$\exists l'.$ after_external $v_{ret}\ l$ = Some $l' \wedge$ **all_safe** $\omega'\ l'\ jm'$

Assume all_safe $\omega\ l\ jm$ initially. Then either:

the linked semantics takes a corestep, in which case we can reestablish the all_safe invariant,

the linked semantics is (safely) halted, or

the linked semantics makes a truly external call, *i.e.*, to a function defined by none of the modules in the program. In this case, we must be able to reestablish the invariant for any return values and memories satisfying the function postcondition.

□

7.3.2 Squeezing the (Princeton) Orange

One can "squeeze the orange",[11] in order to extract the juice from

$$\mathcal{L}(\mathcal{J}(C), [\![mod_0]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, [\![mod_1]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})}, \ldots, [\![mod_{N-1}]\!]_{\mathcal{J}(\mathsf{CL_{Sem}})})$$

[11] David Walker refined this metaphor. Andrew Appel originally suggested the name "juicy memories".

by proving that the juicy linked semantics is simulated by its "dry" analog

$$\mathcal{L}(C, [\![mod_0]\!]_{\mathsf{CL_{Sem}}}, [\![mod_1]\!]_{\mathsf{CL_{Sem}}}, \ldots, [\![mod_{N-1}]\!]_{\mathsf{CL_{Sem}}})$$

The general form of the theorem is:

Theorem 10 (Linked Erasure)**.** *Let $sem_0, sem_1, \ldots, sem_{N-1}$ be interaction semantics operating on CompCert memories, with $\mathcal{J}(sem_i)$ the corresponding lifted juicy semantics. Then there is a whole-program simulation*

$$\begin{array}{ll} & \mathcal{L}(\mathcal{J}(sem_0), \mathcal{J}(sem_1), \ldots, \mathcal{J}(sem_{N-1})) \\ \leq & \mathcal{L}(sem_0, sem_1, \ldots, sem_{N-1}) \end{array}$$

Proof. In Coq.[12] □

As a corollary of Theorem 10 and Corollary 4, we get that erasure is safety-preserving.

[12]File `VST/linking/erase_juice.v`.

Chapter

8 Application to CompCert

The techniques of this thesis have been applied to the CompCert certified C compiler (version 2.1). The result is Compositional CompCert, the codebase of which is open source and freely available on GitHub.[1]

8.1 Compositional CompCert

The proved-correct phases of the Compositional CompCert compiler are shown in Figure 8.1, 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); the adaptation of their proofs is ongoing work. The toplevel theorems we prove are the following.

Theorem 11 (Compiler Correctness)**.** *Let* $\mathsf{CompCert}$ *denote the compilation function that composes the phases in Figure 8.1. If* $\mathsf{CompCert}(S) = \mathsf{Some}\ T$*, for Clight module* S *and x86 module* T*, then* $[\![S]\!] \preceq [\![T]\!]$.

---

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

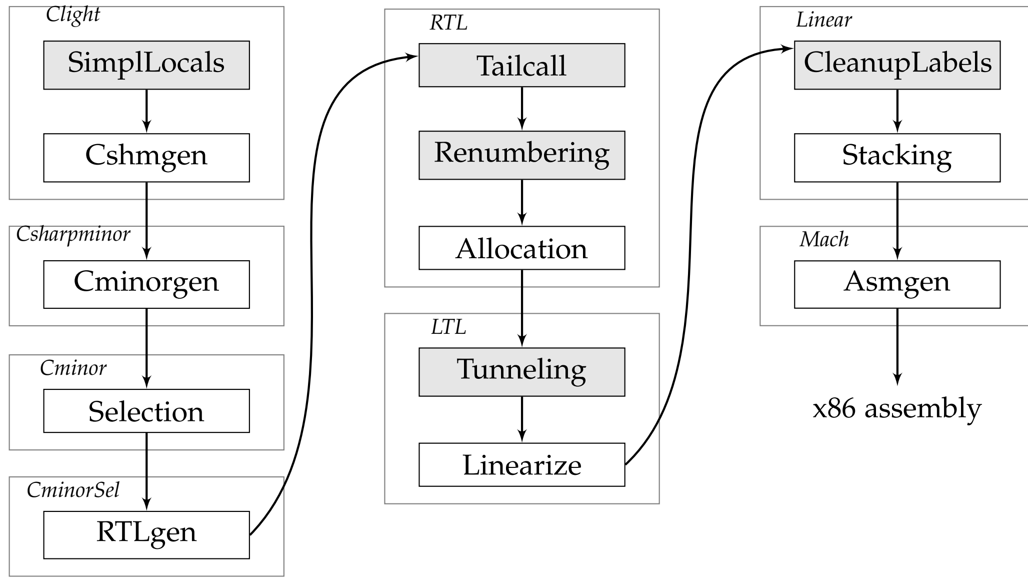


Figure 8.1: The phases of Compositional CompCert. Boxes in gray are optimization passes. Outer boxes indicate source languages.

Proof. By transitive composition of the simulation proofs for the individual phases in Figure 8.1 using Theorem 4.[2] □

Corollary 9 (Compositional Compiler Correctness). *Let $P_S = S_0, S_1, \ldots, S_{N-1}$ be a set of Clight modules with equal global domains such that* $\mathsf{CompCert}(S_i) = \mathsf{Some}\ T_i$ *for each i. Let P_T abbreviate the target program $T_0, T_1, \ldots, T_{N-1}$. Then $P_S \sim_{\mathsf{rc}} P_T$.*

Proof. Theorem 11 establishes the simulations $[\![S_i]\!] \preceq [\![T_i]\!]$. By Corollary 7, we get that $P_S \sim_{\mathsf{rc}} P_T$.[3] The side conditions of Corollary 7 are:

- for all i, $[\![S_i]\!]$ is reach-closed (Theorem 1, Clight is reach-closed);
- for all i, $[\![T_i]\!]$ is valid (Theorem 2, x86 is valid); and
- for all deterministic contexts C, the linked semantics $\mathcal{L}(C, [\![P_T]\!])$ is also deterministic (follows by Theorem 3 and determinism of CompCert x86 assembly).

□

We also get the following contextual refinement.

---

[2]The Coq proof is file `compcomp/driver/CompositionalCompiler.v`.

[3]The Coq proof is file `compcomp/linking/CompositionalComplements.v`.

Corollary 10 (Contextual Refinement for CompCert)**.** *Let $P_S = S_0, S_1, \ldots, S_{N-1}$ be a set of reach-closed Clight modules with equal global domains such that* $\mathsf{CompCert}(S_i) = \mathsf{Some}\ T_i$ *for each i. Let P_T abbreviate the target program $T_0, T_1, \ldots, T_{N-1}$. Then $P_T \sqsubseteq_{\mathsf{rc}} P_S$.*

Proof. By Theorem 11 and Corollary 8.[4] The side conditions are the same as in Corollary 9. □

8.2 Anatomy of a Phase

Converting a CompCert phase to structured simulations typically proceeded as follows: Refine CompCert's internal match-state relation $\sim_f$ (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 equality (or extension) phases had to be reformulated as injections. Particular care was needed to assign correct ownership and visibility information to compiler-introduced memory blocks.

In addition, add to each $\sim_\mu$ relation the clauses: vis_μ 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.

In total, porting the CompCert phases in Figure 8.1 to structured simulations took approximately 10 person-months, though much of this time was spent at the "boundaries" of the proof, updating the interfaces that connected, in particular, our linking semantics and proofs to structured simulations. In general, the porting time decreased as 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 on structured injections and simulations, which will remain useful as we continue to adapt the last few optimization passes.

---

[4]The Coq proof is file `compcomp/linking/CompositionalComplements.v`.

8.3 Anatomy of the Proof

As an (albeit imperfect) measure of the amount of effort involved in building the mechanized development that accompanies this thesis, I report lines-of-code for selected representative files in the development (Figure 8.2),

For Compositional CompCert, proofs of individual phases ("new") were on the order of 5klocs. By contrast, CompCert 2.1's ("old") proofs are about $2\times$ 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.

The underlying theories are on the order of $4,000$ lines of specifications and approximately $32,000$ lines of proofs, spread across more than $1,400$ distinct lemmas. The code and proofs corresponding to Chapter 7 (not shown in the table) total approximately $3,500$ lines.

	Specs.		Proofs		Lemmas	
Compiler Phases:	old	new	old	new	old	new
SimplLocals	725	979	2168	4670	71	126
Csharpminorgen	1201	1634	1450	3207	65	96
Cminorgen	1619	1635	2796	5041	85	112
Selection	1663	1463	3239	5926	145	248
RTLgen	961	1364	1475	4812	48	97
Tailcall	441	643	628	1713	19	32
Renumbering	441	643	267	1428	13	38
Allocation	765	1273	2197	4390	93	124
Tunneling	324	630	417	1941	14	51
Linearize	606	1359	750	2432	35	73
Cleanup Labels	282	729	372	2067	15	54
Stacking	712	1685	2906	6713	107	182
Asmgen	1326	1970	2863	5370	105	125
Theories:						
Interaction Sems. (Chapter 3)		75		167		16
Trace Sems. (Chapter 3)		270		-		-
Gallina Sems. (Chapter 3)		58		-		-
Linking (Chapters 4 and 6)		2454		8469		481
Whole-Program Sims. (Chapter 5)		118		594		13
Structured Injs. (Chapter 5)		55		2099		182
Structured Sims. (Chapter 5)		349		8056		487
Transitivity (Chapter 6)		105		5285		49
Valid Semantics (Chapter 6)		234		-		-
Reach-Closed Semantics (Chapter 6)		270		-		-
Well-Defined Contexts (Chapter 6)		89		-		-
Clight Well-Defined (Chapter 6)		-		7035		181

Figure 8.2: Lines of code for selected parts of the development. “Lemmas” is number of theorems proved.

Chapter

9 Conclusion

9.1 What Has Been Achieved?

This dissertation set out to give a semantic characterization of the program contexts for which an optimizing C compiler is sound, answering the question *For which program contexts is an optimizing C compiler correct?* In order to do so, it developed

interaction and linking semantics (Chapters 3 and 4), which made it possible to state compiler correctness as cross-language contextual equivalence (Section 4.2, refined in Chapter 6), showing *how to achieve language-independence*; and

structured simulations (Section 5.3.2), an extension of CompCert's forward simulation proof method that composes both transitively, across compiler phases (Chapter 6, Theorem 4), and horizontally, across separately compiled modules (Chapter 6, Theorem 5), answering the question *How to reason about equivalence of open modules?*

In answer to the question *Do the techniques scale to realistic languages like C and to real systems like the CompCert?* the above techniques were applied to build Compositional CompCert (Chapter 8 and [SBCA14]), a verified separate compiler for C. In addition, I showed (Chapter 7) how to connect the Verifiable C program logic [ADH+14] to Compositional CompCert, yielding a method for modular proving of compiled C/assembly programs.

9.2 Discussion

The techniques of this dissertation have applicability beyond just C/assembly-language programs, separate compilation, or just CompCert.

Interaction semantics are a natural tool for expressing more complex modes of interaction than the linking semantics of Chapter 4. For example, the following code

```
Record ConcurrentState (threads : thread_idx → ThreadSem) ≜
  mkConcurrentState {
    schedule : ℕ → thread_idx;
    permissionOracle : ℕ → Set location;
    threadStates : ∀tid : thread_idx. option (CoreState (threads tid))
  }
```

sketches a possible adaptation of the LinkedState record of Chapter 4, which modeled the state of a linked program, to a collection of concurrent threads. Thread semantics are defined by a parameter $threads$ that maps thread indices $thread_idx$ to records giving each thread's interaction semantics. The components of ConcurrentState might include the schedule, a stream of thread indices associated with each timestep of the concurrent execution, a permission oracle (derived from, *e.g.*, a program safety proof in Concurrent Separation Logic) describing the ownership transfers that must occur at each lock/unlock operation, as well as a map, threadStates, of the core states associated with each active thread. It is not difficult to imagine an adaptation of the interaction semantics $\mathcal{L}$ of program linking to this (coarse-grained) concurrent setting. An important point is that even in coarse-grained Pthreads-style concurrency, interactions among threads occur only at external function call points (`lock`/`unlock` are themselves just external functions). It is likely that most, if not all, "external-function-call-like" protocols could be modeled in the interaction semantics framework.

My work on separate compilation for C-like languages also exposed some warts of C. For example, much of the complexity of structured simulations (Chapter 5) was driven by C-language "features" such as addressed stack-allocated local variables. When pointers to compiler-managed data such as stack frames escape to external modules, one must keep careful track, in compiler invariants, of those parts of memory that may be updated by external functions. The invariants and proofs would have been simpler in a more restrictive language setting, in which compiler-managed data was instead guaranteed private.

From an engineering standpoint, the adaptation of CompCert 2.1 to interaction semantics and structured simulations (Chapter 8) could have

been simplified considerably, in retrospect, if my colleagues and I had first built a uniform interface to CompCert's compiler-phase proofs. It is not immediately obvious that such an interface is possible. However, the proofs do share many features. For example, CompCert's simulation invariants often have three constructors: One for the normal "running state" case, one for making function calls, and one for returning from calls. Each of these cases usually decomposes into a predictable set of sub-invariants on, *e.g.*, the stack frame and temporaries environment. Spending some engineering effort up front to expose this structure might have made it possible to adapt the compiler proofs to the Compositional CompCert framework in a more uniform fashion.

9.3 Future Directions

Heterogeneous Verified Systems. The verification techniques I described in Chapter 7—with which modules are proved independently in the Verifiable C logic, against a common specification $\mathcal{O}$, and then proved sound with respect to the $\mathcal{L}$ semantics of Chapter 4—have so far targeted mostly-C programs. The approach supports more heterogeneous systems, in which some modules are in C, some are in assembly, and others are in a third language such as Coq's Gallina. However, the verification techniques are not optimized for such highly heterogeneous programs—modules in languages other than C (*e.g.*, those in CompCert x86 assembly) must be proved directly from their operational semantics.

It would be convenient to provide support for other program logics, besides just Verifiable C. For example, one might prove assembly modules correct in a variant of XCAP [NS06] or in a modified Bedrock [Chl11], adapted to CompCert x86 assembly. There are no major technical limitations here. Chapter 7's semantics preservation proofs were designed to be mostly independent of the particular program logic used to establish per-module safety. The major interdependency is the shared specification language $\mathcal{O}$.

More radical is to provably compile programs that include modules written in a high-level language like Gallina, in addition to C and assembly. One could use Coq's current code extraction to compile the Gallina modules to OCaml, and then further compile with `ocamlopt`. However, this process yields an unverified toolchain (Coq extraction, `ocamlopt`, and the OCaml runtime must all be trusted).[1] Another (better) solution is to build a verified compiler for Coq itself, a project currently underway at Prince-

[1]One could argue that, as Coq users, we must trust the latter two already.

ton. This "CertiCoq" could then be fruitfully combined with Compositional CompCert to yield a certified compiler for Gallina/C/assembly programs.

The motivation here is efficiency of the program verifier, by which I mean the human actually guiding the proof assistant. Certain low-level software components, such as garbage collectors and OS kernels, are more suited to implementation at the C level of abstraction. However, the cost of verification is proportionately higher at this level. Other components—such as the "glue" code that composes large sections of many software systems—are more conveniently implemented and proved in a purely functional language like Gallina with a clean proof theory.

Syntactic Linking. The linking operator $\mathcal{L}$ first introduced in Chapter 4 is *semantic*, in the sense that it takes as arguments not the syntax of its input modules but instead their associated interaction semantics, and produces a new interaction semantics as result. There is an alternative kind of *syntactic* linking that operates directly on the syntax of modules, *e.g.*, by syntactically concatenating a number of program fragments, all of which must be in the same language. In general, semantic linking provides more flexibility; for instance, it enables linking of modules in a variety of languages, as long as the domain of interpretation is shared across modules. On the other hand, cross-language syntactic linking does not make sense (a C module cannot be concatenated to an assembly module; the types do not match).

Syntactic linking is nevertheless useful. It would support certain cross-module optimizations such as external function inlining (compile two modules to a common intermediate language; link syntactically; then do standard intramodule inlining). A syntactic linking proof at the x86 assembly level would also provide additional justification—beyond the arguments already presented in Chapter 3—of Compositional CompCert's "copying" x86 interaction semantics. This assembly-level syntactic linking proof might, in addition, provide a means of integration into projects such as Shao *et al.*'s CertiKOS certified microkernel [GVF+11, GKR+15], which currently uses a modified CompCert compiler to translate C functions to assembly code, in the context of assembly-level callers. This modified CompCert compiler does not yet support address-taken local variables—one of the aspects of C that so complicated the Compositional CompCert proofs (*cf.* Chapters 1 and 5).

I have done initial experiments with syntactic linking.[2] The idea—rather than reproving syntactic linking for each language—is to define a general proof infrastructure that depends on (i) a monoid $c_1 \circ c_2$ on corestates of the

[2] File `compcomp/linking/stacking.v`.

language, modeling abstract stack-frame composition in linking semantics $\mathcal{L}$;[3] and (ii) proofs that the corestep relation, at_external, after_external, *etc.* are compatible with the monoid. The simulation that relates the abstract $\mathcal{L}$ activation-record stack s to the actual stack s' (of the syntactically linked whole program) states that s' is the fold of the monoid operator $\circ$ over s (lifted to option, with unit None).

9.4 Conclusions

Tony Hoare, in 2005, called the verifying compiler one of the "grand challenges" [HM05] of computer science—on par with Fermat's last theorem (in math) and P vs. NP. Such comparisons are perhaps a bit overblown. Regardless, Leroy's CompCert, the first *verified* optimizing compiler for a realistic language, was a definitive milestone. This thesis, while definitively *not* the last word on compiler verification, has advanced the art by a few (small) steps.

[3] In a language like C, the monoid is defined, for example, as list append on activation-record stacks.

Bibliography

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Annual* IEEE *Symp. on Logic in Computer Science*, pages 75–86, June 2002.

[AB11] Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the International Conference on Functional Programming*, 2011.

[ADH+14] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[AMRV07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jerôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, January 2007.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[BH09] Nick Benton and Chung-Kil Hur. Biorthogonality, Step-Indexing and Compiler Correctness. In *Proceedings of the International Conference on Functional Programming*, 2009.

[BH10] Nick Benton and Chung-Kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, 2010.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified Compilation for Shared-memory C. In *Proceedings of the European Symposium on Programming*, 2014.

[Chl07] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2011.

[Com] Verification of Separate Compilation for CompCert. URL: http://sf.snu.ac.kr/compcertsep. Last accessed January 30, 2015.

[Dav03] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Software Engineeering Notes*, 28(6), 2003.

[Doc12] Robert W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608. ACM, 2015.

[GT12] Dan Ghica and Nikos Tzevelekos. A system-level game semantics. In *Proceedings of Mathematical Foundations of Programming Semantics*, 2012.

[GVF+11] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 3. ACM, 2011.

[HD11] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[HDA10] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–185, January 2010.

[HDNV12] C.K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[HM05] Tony Hoare and Robin Milner. Grand Challenges for Computing Research. *The Computer Journal*, 48(1):49–52, 2005.

[HNDV13] C.K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. Parametric bisimulations: A logical step forward. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[Int] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html. Last accessed September 5, 2014.

[ISO11] ISO. C11 Draft Standard. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, April 2011.

[KR98] Brian W Kernighan and Dennis M Ritchie. The C Programming Language, 1998.

[LABS14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model. In Andrew W. Appel, editor, *Program Logics for Certified Compilers*. Cambridge, 2014.

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1), 2008.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LFF12] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[Loc12] Andreas 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.

[LWN13] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[Man14] William E. Mansky. *Specifying and Verifying Program Transformations with PTRANS*. PhD thesis, University of Illinois, 2014.

[McK14] Matthew McKay. Compiler correctness via contextual equivalence. Undergraduate thesis, Carnegie Mellon University, May 2014.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–10. ACM, 2007.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*. British Computer Society, 1971.

[Moo89] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 1, 1967.

[NLWSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the European Symposium on Programming*, 2014.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 320–333, 2006.

[PA14] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Proceedings of the European Symposium on Programming*, 2014.

[PIL] Compositional Compiler Verification via Parametric Simulation. URL: http://www.mpi-sws.org/~neis/pils/. Last accessed January 30, 2015.

[RSW+15] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jeremie Koenig, and Yuchen Fu. A compositional semantics for verified separate compilation and linking. In *Proceedings of the ACM SIGPLAN Conference on Certified Programs and Proofs*, 2015.

[SBCA14] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew Appel. The Compositional CompCert Proof Development. URL: https://github.com/PrincetonUniversity/compcomp, 2014.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2015.

[SVN+13] Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[WCC14] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *Proceedings of OOPSLA*, 2014.