Compiler Correctness for Concurrency: from concurrent separation logic to shared-memory assembly language

Santiago Cuellar[1], Nick Giannarakis[1], Jean-Marie Madiot[1,2], William Mansky[1,3], Lennart Beringer[1], Qinxiang Cao[1,4], and Andrew W. Appel[1]

[1]Princeton University

[2]Currently at INRIA Paris

[3]Currently at University of Illinois

[4]Currently at Shanghai Jiao Tong University

Princeton University Computer Science

Technical Report TR-014-19

March 20, 2020

Abstract

Optimizing compilers change a program based on a formal analysis of its code, and modern processors further rearrange the program order. It is hard to reason about such transformations, which makes them a source of bugs, particularly for concurrent shared-memory programs where the order of execution is critical. On the other hand, programmers should reason about their program in the source language, which abstracts such low level details.

We present the Concurrent Permission Machine (CPM), a semantic model for shared-memory concurrent programs, which is: (1) sound for higher-order Concurrent Separation Logic, (2) convenient to reason about compiler correctness, and (3) useful for proving reduction theorems on weak memory models. The key feature of the CPM is that it exploits the fact that correct shared-memory programs are permission coherent: threads have (at any given time) noncompeting permission to access memory, and their load/store operations respect those permissions.

Compilers are often written with sequential code in mind, and proving the correctness of those compilers is hard enough without concurrency. Indeed, the machine-checked proof of correctness for the CompCert C compiler was a major advance in the field. Using the CPM to conveniently distinguish sequential execution from concurrent interactions, we show how to reuse the (sequential) CompCert proof, without major changes, to guarantee a stronger concurrent-permission-aware notion of correctness.

Chapter 1

Introduction

Write a shared-memory concurrent C program using Pthreads primitives such as mutexes or general semaphores. Now, reason about your program: is it correct? Once the optimizing C compiler transforms it, is it still correct? Once the multicore computer with weakly consistent memory executes it, is it still correct? Can you verify all that, with machine checked proofs?

When writing and compiling your C program, pay attention to the difference between *atomic* loads/stores and *nonatomic* loads/stores. Nonatomic memory operations are the "ordinary" ones that compilers (and hardware instruction-execution pipelines) should optimize: eliminate redundant loads, eliminate redundant stores, insert redundant loads (if that reduces register pressure), hoist loads above stores (if the addresses are different). Atomic memory operations are used for synchronization; the benign races. In a typical program, one expects that most memory operations are "ordinary" nonatomic accesses, amenable to optimization (and easier reasoning about source programs). In an ideal world, the bulk of program and compiler verification are done on on "ordinary" memory access, as if the program were sequential. Atomic memory operations, where synchronization magic happens, should be modularly separated to make reasoning easier.

This work aims to reason about such programs, prove them correct in the source code, compile them correctly and have them execute correctly in a realistic machine, what we call the *top-to-bottom correctness*. All of this, with modular proofs, connected in an entirely machine-checked proof.

The backbone of this work is a formalization of the execution model of C and assembly language programs that expresses, with great generality, what programs should be allowed to do (but not more); that allows modular reasoning about source programs, source-level program logics, source-level static analyses, compiler optimizations, and execution on relaxed memory models. Our semantic model, called the *Concurrent Permission Machine* (or CPM for short), carefully separates nonatomic operations from synchronizations. It is sufficiently *detailed* that we have a formal instantiation of it (in Coq) for the CompCert Clight language (and embed the CompCert Clight sequential operational semantics), and a formal instantiation of CompCert assembly languages such as x86 (in both 32 and 64-bit modes).

Contributions of this work.

1. We formalize the *Concurrent Permission Machine*, an operational model of execution of C, of assembly language, or any language in the CompCert chain of intermediate languages. The CPM abstraction is designed to capture the intent of the C11 standard with respect to lock-based concurrency; and be friendly to reasoning about the correctness of optimizing compilers. It has cooperative concurrency, and virtual "permissions" that enforce noninterference between threads. Each thread's execution can be reasoned about while largely ignoring the existence of other threads.

2. We show how the CPM model enables modular specification of concurrent compiler correctness that transports safety and partial correctness of multithreaded C programs to assembly code using the model of cooperative concurrency. The

CPM separates sequential semantics from concurrent semantics, so reasoning about the semantics of concurrent programs can be kept separate from correctness proofs of compiler optimizations. In particular, this modular approach allows us to reuse existing proofs of compiler correctness for sequential code.

3. We apply our framework to CompCert, a proven correct compiler for sequential code. With minimal modifications of the CompCert code, we lift its specifications and proofs into CPMCompCert, a proven correct compiler that supports concurrency. The semantic model of CPMCompCert is the CPM, and is compatible with the other parts of the top-to-bottom proof (in Coq).

4. The Clight-instantiated CPM model is sufficiently expressive to capture user-level reasoning principles, by giving (in Coq) a soundness proof for a modern concurrent separation logic (CSL) that supports "ghost state" that can express concurrency-interaction protocols and first-class (dynamically creatable) threads and locks. The CSL logic and its semantic model build on those of the Verified Software Toolchain (VST) [3], a framework for proving correctness of C programs in Coq, such that users can apply the CSL to their programs using VST's existing proof automation system.

5. Safety and correctness of an assembly-language CPM execution entails similar properties for fine-grained interleaving, and any instruction-level interleaved execution of the program is well synchronized. Therefore, all executions of the program on a machine with relaxed memory semantics will have the same behavior as in the CPM.

The idea behind the Concurrent Permission Machine is to lift the single-processor operational semantics into a multiprocessor operational semantics, with multiple *local* thread-states sharing a single memory. Our innovation is to add a per-thread *permission map*—a mapping from address to permissions—such that no two threads have competing permissions at the same address. A thread is *stuck* if it tries to read address a without read permission, or write without write permission—so races are impossible in nonstuck executions. Acquiring a lock *increases* the thread's permissions at some set of addresses; releasing a lock *decreases* permissions. Which permissions are increased or decreased? Intuitively, the lock controls access to some shared data and the the thread should gain access to that data when it acquires the lock and loose the access when it releases it. And finally, these permission maps have no physical manifestation in the execution of a machine-language program—at the end we prove an erasure theorem.

CPM versus race-freedom. It is conventional to ask, "is this execution data-race-free?" As this property refers to the whole execution trace of a program we believe that it is an overly complex invariant to propagate through a compiler-correctness proof. Our Concurrent Permission Machine establishes a simpler property: permission coherence. From this, we prove that executions in the CPM are *well synchronized*—nonatomic accesses are regulated by locks and the lock operations are used properly. Owens [29, Theorem 2] and Batty [5, pg. 178, Theorem 13] have notions of well synchronized programs for the TSO and C11 memory models respectively. They prove that any behavior observed in a weakly consistent execution (of a well synchronized program) can be observed in some *sequentially consistent* one.

Concurrency primitives. We demonstrate proofs for programs with semaphores. These are more general than "locks" because they permit "daring concurrency" [28] in which thread 1 may acquire lock A; pass the "hold" of A to thread 2 by releasing lock B; and then, after thread 2 acquires B, it may release A. For example, our system supports this "handshake" protocol:

```
while (1) {             while (1) {
   acquire(A);             acquire(B);           p is a shared array,
   p[0]=x;                 y=p[0];               x and y are local variables
   release(B);             release(A);
}                       }
```

In conventional terminology, "semaphores" permit daring concurrency, but the thread that releases a "lock" must be the same thread that acquired it. We will use the terms interchangeably, to mean a semaphore that permits daring concurrency.

In this work we support: thread spawn, lock creation/destruction, lock acquire, and lock release. We do not currently address lock-free concurrency, such as C11 atomic load and store operations, but our separation logic extends to atomic operations [27] and we expect that the CPM extends as well (see section 8.1).

Rich and usable specifications. Top-to-bottom correctness could be viewed as three separate, or modular, results: CSL soundness, compiler correctness, and weak memory correctness. A significant contribution of this thesis is, nonetheless, the achievement of compatibility among the three results, rendering the specification of each both strong enough to be useful and weak enough to be practical to prove. This dichotomy is particularly relevant for the compiler specification that must support the other two parts. At various points through development, we found the working specification of the compiler making assumptions that were either too strong to be established by the CSL, or too loose to support our results about weak-memory consistency. For example, a bug in the definition of permission coherence (section 3.2.2) made it impossible to commute pointer comparisons through the instruction interleaving (section 7.1). Together, the CPM and the notion of coherence are major steps in reaching this balance.

Why not use CSL directly? Why is it useful to define the Concurrent Permission Machine abstraction? Could we not use separation logic directly in reasoning about compiler correctness, or in proving well-synchronization of executions? The problem with that approach is that modern CSLs are quite complex for two main reasons. First, to represent higher-order features such as function-passing and dynamic lock creation, they have step-indexed models that manifest (in the logic) as modal operators and bifunctors. Second, to model resource invariants that specify protocol-correctness (and not just safety) they have "ghost resources" in arbitrary partial commutative monoids. Step-indexed models and ghost resources would unnecessarily complicate reasoning about compiler optimizations (which may change the number of execution steps, and should not interact with ghosts); and would not help in proving well-synchronization properties. The CPM abstracts all the higher-order separation logic into a first-order notion of permission coherence, directly useful in lower-level proofs.

Chapter 2

Related Work

In this section we review related work, paying special attention to the developments that bear technical similarities to ours.

Boehm wrote "Threads cannot be implemented as a library," [7] explaining that many compilers did optimizations that were unsafe for concurrency. He argued that better language specifications were needed. This motivated our work. Our new result shows that CompCert's specification (with our improvements relating to permissions and events) *does* permit threads to be implemented "as a library", in the sense that it decouples the compiler-correctness proof from the thread-library correctness proof.

2.1 Compiler specification

The Next 700 Compiler Correctness Theorems

Patterson and Ahmed [10] have laid out a comprehensive framework for compiler correctness and compiler compositionally. They focus only on separate compilation, but we can use their framework as a reference for the compositionally of our compiler proof with concurrency.

Horizontal compositionally. The horizontal compositionally in concurrent programs comes from the parallel composition of threads, instead of linking modules. We cannot express such compositionally syntactically, as the authors do with compilation units (i.e., $e_s \bowtie e'_s$), because threads are dynamic semantic objects instead of static syntactic units. However, one of our innovations is to show that our concurrent semantics (the Concurrent Permission Machine, section 3.2) supports language-independent compositionally (the Hybrid Machine section 5.2.1).

Vertical compositionally. We introduce two types of simulation relations: for thread local semantics *MOIST simulations* (section 4.5), which stands for Memory-explicit, Observable, Injectable, Startable, Trace, simluations; and for full program semantics *CPM simulations* (section 5.1). Both simulations compose transitively in the vertical sense but Patterson and Ahmed would not consider them "truly vertically compositional", because they expose the memory reorderings produced by the compiler. We also require that, for all intermediate languages, a CompCert-style memory may be produced from each state, even if the state may have an abstract representation of memory, or an enriched version of the memory. This approach is slightly more compositional than similar compiler theorems that require all languages to use the same memory model [22, 33].

Interaction semantics and Logical Simulation Relations

Originally, CompCert-compiled programs could not share memory (especially pointer-containing memory) with their contexts (such as system-calls to the O.S. or a hypothetical concurrency library) because the CompCert correctness specification/proof was too weak. To improve the compiler's specification, Beringer *et al.* introduced *Interaction Semantics* [6], reformulating CompCert's operational semantics to permit shared-memory interaction. This was powerful enough to characterize system calls

	CompCert 2.1	CompComp	**Added**	CompCert 3.2	Our work	**Added**
Cminorgen	2796	5018	**52%**	2261	2431	**7%**
RTLgen	1475	4883	**156%**	1593	1690	**6%**
Tailcall	628	1769	**125%**	608	705	**16%**
Stacking	2906	6657	**130%**	2907	2964	**2%**

Table 2.1: Comparing lines of code for selected compiler passes in CompCert, CompComp and our work.

that read or write the memory of the process, and was a step towards shared-memory concurrency.

Interaction Semantics was also a step towards separate compilation for CompCert, and the ability to link with assembly language programs. To achieve fully modular separate compilation and linking, *Compositional CompCert* (CompComp) [33] extended Interaction Semantics, strengthening CompCert's compiler correctness theorem with Logical Simulation Relations and *structured injections* that make the right "contract" between separately compiled modules. Unfortunately, this stronger theorem required substantial changes to the CompCert 2.1 correctness proof (table 2.1), so the CompCert maintainers declined to merge it into the master branch. In our current work, where we focus on concurrency rather than separate compilation, we have found a lightweight modification to CompCert's correctness specification, that will be easy to integrate into the trunk and to maintain. Our new *MOIST semantics* and *simulations*, which stands for Memory-explicit, Observable, Injectable, Startable, Trace, are closely related to Interaction Semantics and Logical Simulation Relations.

Memory. Every intermediate language of CompCert uses states that contain a memory of the same type mem because it is useful to reason about the changes in memory produced by compilation. Compositional CompCert (CompComp) makes that decision explicit, by requiring all *Core semantics*, to act on pairs of *cores* (i.e., state without the memory) and memories. To develop CompComp's we changed all type signatures in CompCert, to match this design decision: while the step relation in

CompCert has type $\mathsf{genv} \rightarrow \mathsf{state} \rightarrow \mathsf{trace} \rightarrow \mathsf{state} \rightarrow \mathbb{P}$ in Compositional CompCert it has type $\mathsf{genv} \rightarrow \mathsf{core} \rightarrow \mathsf{mem} \rightarrow \mathsf{trace} \rightarrow \mathsf{core} \rightarrow \mathsf{mem} \rightarrow \mathbb{P}$.

Our **M**OIST semantics only require that a state may be mapped to a pair of core and memory. Namely, every intermediate language only has to provide a function get_mem: state $\rightarrow$ mem, to extract the memory from the state. In all current languages of CompCert, this function is trivial. First, this approach minimizes the changes to CompCert's existing proofs since we do not have to adapt the type of all semantics. Second, our approach allows intermediate languages with richer notions of memory such as the juicy memory [3] or other kinds of abstract state in [15], as long as they can produce a get_mem.

Observable. By design, Core semantics [33] don't take a step when they call an external function; there is a function at_external that describes such states and another after_external that reestablishes the state after the external function is executed (e.g. writes the result in the right place), but there is no step between the two. In essence, core semantics are precisely the small steps taken by the module/thread/core executing; the other steps are introduced after semantic linking.

We believe that these semantics are very useful for semantic linking (and we use core semantics to build our concurrent machine), but it is not the best model for compilation. To adapt compiler correctness to the core semantics, Stewart et al. [33] define logical simulation relations which are quite complicated for the at_external and after_external cases. These relations form a rely-guarantee between callers and callees which become pretty complex when both are being compiled. Our MOIST simulations and semantics, together with our technique to Compile One At a Time (COAT, section 5.2), avoid that complexity.

Like Core semantics, M**O**IST semantics makes states that call external functions **o**bservable, as described by a function at_external; however, MOIST semantics still step from those states (where Core semantics prohibits a thread's state from stepping at an external call). Our semantics can take one single 'big step' representing the entire execution of the external call, just like CompCert. In essence, MOIST semantics represent the thread/module local *view* of an execution, where internal steps are taken as small steps and external steps as big-step. This view is consistent with CompCert's semantics.

An important feature of MOIST semantics is that it requires no changes to the step relation of each intermediate language of CompCert; we only need to define at_external to observe the external calls and their arguments. In addition, we can derive a core semantics from a MOIST semantics, but not the other way around. We use this transformation to derive core semantics for Clight and Assembly, which is what we use to build our concurrent machine.

Moreover, MOIST simulations avoid having the complex relations for at_external and after_external; instead, we only need two clauses that say that (1) the compiler doesn't remove external function calls from executions and (2) the compiler can't change the number of steps taken by an external function call. This simplicity is enabled by the COAT technique (section 5.2), which allows us to model the context as external function calls, instead of parts of the program that are also compiling.

Startable. The MOIST semantics' entry_point predicates are very similar to Compositional CompCert's initial_core, with two main difference. First, Compositional CompCert does not really pass arguments on the stack (which means that calls to main(argc,argv) on x86-32 and other stack-based architectures are not modelled accurately). Interaction semantics [6] is designed to link any module of any language and, because different languages treat arguments differently, the authors decided to unify all semantics with an abstract calling convention. We avoid that idealization and provide concrete semantics for argument-passing, including passing arguments on the stack when calling external functions. Second, thanks to our COAT technique, the simulation diagram for initial states in MOIST simulations, is much simpler than the one in Logical Simulation Relations (as explained in section 4.5)

Lightweight separate compilation of CompCert

SepCompCert [22] is another enhancement of CompCert that supports syntactic linking of modules compiled with the same compiler (i.e., CompCert). The SepCompCert framework does not use any notion of semantic linking and, in particular, does not have semantics for linking modules in different languages. However, the development of SepCompCert is significantly simpler than that of Compositional CompCert and has since been adopted by the developers of CompCert in the trunk. SepCompCert does not address concurrency; but its success in finding a solution to separate compilation with such lightweight changes to CompCert itself inspired us to find a solution to concurrency with lightweight changes to CompCert.

External modules as internal steps. Compositional CompComp and SepCompCert have very different semantics for separately compiled modules. As we described above, the former uses interaction semantics for each module such that other modules are treated as external functions. Only the linker gives a whole-program semantics. The latter only supports syntactic linking, so the only semantics is that of the whole program. In this case, functions defined in other modules execute normally as internal steps and there is no need for interaction semantics. The upside of the second approach is that it allows to reuse most of CompCert's proofs. On the other hand, SepCompCert can't reason about modules compiled with other compilers, especially if the module is not written in C.

The present work is not about separately compiled modules but about synchronization libraries and threads running in parallel. However, we believe our MOIST semantics strikes a balance between the two approaches described above. We treat the execution of the context as external functions, but we use CompCert's existing machinery for those. More precisely, MOIST semantics are thread-local views of the execution of the whole program, executing the context with "big-step" semantics, just like CompCert treats external functions. This approach allows us to reuse most proofs in CompCert. On the other hand, we can also derive an interaction semantics from a MOIST one and we can "link" them together in the Concurrent Permission Machine, to provide precise small-step semantics for the whole program.

Optimizations in RTL. SepCompCert supports the linking of modules that have been compiled with different levels of optimization. Without a notion of semantic linking, the authors can only hope to achieve that if every "optional" optimization is from an intermediate language to the same intermediate language, and no new optimizations are added that introduce new intermediate languages. Fortunately, all optional optimizations in CompCert are done in the RTL language, but any future optimizations that break this rule will not be supported by SepCompCert.

We are working with threads and not separately compiled modules, however our COAT technique models compilation as if each thread were being compiled separately. Part of the process is to define Concurrent Permission Machines with threads in different languages (Hybrid Machines, section 5.2). Thanks to this multi-language semantics, we don't need to make any assumptions about optional optimizations.

2.2 Compiler models supporting concurrency

Podkopaev et al. [30] have proven a compiler correct with respect to the *promising semantics* of Kang et al. [21], which features a high-level relaxed model of memory. They compile down to ARMv8-POP and later to POWER, ARMv7 and ARMv8 [31]. Their work focuses on the complexity of relaxed memory models, but does not reuse existing sequential compilers and has not been yet connected to a full optimizing compiler. We would be interested to explore a combination of our work, to leverage the existing CompCert compiler, and their work, to include other synchronization atomics.

Dodds et al. [12] developed an automatic checker for compiler optimizations that supports relaxed memory models. Our trace-preservation simulations mirror their denotation-preserving relation. However, they consider observations for all possible contexts while we only consider permission-coherent ones (section 3.2.2). In this way we can verify transformations of code that manipulates shared memory that they cannot.[1]

CompCertTSO [32] is a verified optimizing C compiler for programs that use atomic memory operations on x86 TSO. These atomics can be used to write high performance lock-free algorithms. But their semantics bakes TSO atomics into a fork of CompCert 1.5. We see no path forward for extending it to more relaxed memory architectures such as Power. In addition, it's a substantial modification to standard CompCert[2] that is incompatible with the CompCert's correctness proof for sequential programs and thus impractical for the CompCert team to adopt. Our approach generalizes to many architectures, and is much more compatible with standard CompCert.

Certified concurrent abstraction layers

CCAL [15] is a toolkit to support concurrency in the Certified Abstraction Layers framework [14]. The approach is general-purpose but the design is driven for verifi-

[1] For example, their tool cannot verify $\mathsf{l} := \mathsf{load}\ (\mathsf{x});\ \mathsf{store}\ (\mathsf{x},\mathsf{l}) \rightsquigarrow \mathsf{l} := \mathsf{load}\ (\mathsf{x})$.

[2] The paper [32] reports 86K lines of Coq, compared with 55K lines for the base CompCert 1.5 sequential-language compiler. The paper does not report how many lines were left unchanged; supposing it were 40K, then the ratio of new-or-modified lines (46K) to old lines (55K) would be over 80%.

cation of small operating system kernels and thus has some limitations, as explained below.

CPUs and threads. CCAL makes an explicit distinction between threads in the same CPU and threads in different CPUs, potentially supporting optimizations that exploit these facts. We consider that a commendable achievement. However, we believe that such distinctions are only useful in system design, where the "user" wants to have tight control over the contents of cache. General software programmers shouldn't need to worry about what threads are sharing a CPU and, in reasoning about the correctness of concurrent programs, it can be appropriate to abstract this. Our system abstracts all threads in such a way that it doesn't matter on what CPU they are running.

Shared memory. In CCAL, shared memory concurrency can only be achieved through external function calls. That is, reading or modifying shared memory can only be done through *shared primitive calls* which are specified in Coq and represent the behavior of some external modules that handle shared memory access. In that way, different threads in CCAL can interact with "shared memory", through function calls, but one cannot write code that directly manipulates shared memory. Our Concurrent Permission Machine also uses primitive functions to represent synchronization operations, but on only those functions that would normally be part of a concurrency library (such as pthread_mutex_lock or pthread_mutex_unlock in pthreads). Between the primitive calls to acquire and release, our programs are allowed to manipulate shared memory freely.

Sharing stack-allocated variables. The Certified Abstraction Layers framework does not support creating pointers to stack-allocated variables. In particular, threads cannot share stack-allocated variables. We don't impose such a restriction.

CASCompCert

Jiang et al. [18] have presented a framework for reasoning about compilers supporting concurrency. The authors apply their framework in CASCompCert, another extension of CompCert that supports race-free concurrent Clight programs. We find it encouraging that we have independently developed several concepts that closely resemble those in CASCompComp. However the two systems differ in significant ways:

Footprints. One of the main contributions of CASCompCert is to define footprints for every language, showing the locations accessed during execution, and to provide simulations that preserve these footprints. This notion of footprint is closely related to the effects introduced by Stewart et al. [33], but the difference is that footprints also include read accesses to memory. We completely avoid using footprints or effects by leveraging the power of memory permissions, which are already defined in CompCert and proven to be preserved by the compiler. In fact, for well defined languages (defined below), the permissions of a thread are a superset of its footprint. As we discuss in section 3.2.2, we prefer to reason about permission coherence instead of data race freedom and we can derive DRF preservation from permission-coherence preservation.

Sharing stack-allocated variables. The Coq development of CASCompCert does not support cross-module escape of pointers to stack-allocated variables. The authors describe how they would support it, which largely follows the approach of CompComp [33]—that is, using heavyweight structured injections. Our framework supports cross-thread and cross-module escape of pointers to stack-allocated data structures, without the need of structured simulations or extra infrastructure.

Well defined languages. CASCompCert requires that languages are *well defined*, which means their execution respects the memory footprint; in other words, the program only reads locations in the readable set and modifies locations in the writable set. We independently developed a similar notion which we call, more explicitly, *memory semantics*. Thankfully, because of our COAT technique, we only require that the source and target languages (Clight and assembly) are a *memory semantics*. This simplification has two benefits: first, it simplifies the proofs required of the intermediate languages and, second, it allows intermediate languages that are not memory semantics or well defined (as long as the compiler eventually removes all behavior that does not respect the memory interface).

Memory model. The CompCert memory model assumes that consecutive allocations along an execution produce consecutive blocks of memory. The CASCompCert framework departs from this allocation model since changing allocations in one thread will change the execution of other threads. The authors provide a new memory model and present a bijection between the two models to reuse some of the the existing proofs in CompCert. Our COAT technique ensures that a compiler can reason about a thread as if the context (i.e., other threads) is not changing. In this way we can directly use the allocation order of CompCert and we can use the existing memory model; however, our concurrency semantics (CPM, section 3.2) could support either model.

Synchronization primitives. Just like CASCompCert, we support benign races (such as lock-acquire and lock-release) as external modules that threads, written in the sequential languages, can call as functions. We go further to support synchronization primitives to create new threads (i.e., pthread_create), and to convert regular memory locations into semaphores and back (i.e., pthread_mutex_init and pthread_mutex_destroy).

Reasoning about correctness. The design of our Concurrent Permission Machine is driven by its connection to a practical separation logic that can prove programs correct and permission coherent in our semantics. In contrast, the semantic model of CASCompCert is very reasonable but not known to be practical for verification of programs.

Weak cache consistency. Another motivation for our design is to connect, at the assembly level, to a reduction theorem (theorem 7.2.3 and theorem 7.2.6). That is, a proof that a correct program in our assembly semantics (which is, by definition, permission coherent) will run correctly in a machine with weak cache consistency, such as TSO. The CASCompCert compiler presents a notion of data race freedom and safety for assembly programs, but no real connection to realistic memory model.

Modifications to CompCert. CASCompCert, together with the modifications necessary to support stack-allocated variables, are just as heavyweight as CompCompCert, and therefore are impractical. We have produced a practical way of doing concurrent compiler correctness, with little modification to CompCert.

Separate compilation. In section 5.2.3 we describe how to extend our techniques to separate compilation and we believe that our framework can support concurrency and separate compilation. CASCompCert, has already produced a machine checked proof for concurrency and compilation.

Chapter 3

Top to Bottom structure

The goal of this work is to connect the proofs of correctness of a source language, all the way to its correct execution in a real machine. We do this for a rich logic (CSL) to reason about a real language (C, via its high-level intermediate language Clight), through a realistic optimizing compiler (CompCert) and executing on a real weak-cache-consistent multicore processor (X86). Each of these parts has its own intricacies, so we aim to make our work modular.

The programmer should not have to know anything about the compiler or the computer architecture. She wants to use the Dijkstra model of semaphores controlling access to shared data, perhaps using Hoare-style monitors, or using other patterns not limited to a simple mutex. A mutex is a semaphore that is always unlocked by the same thread that locked it; we also support more general synchronization patterns in which one thread acquires a semaphore that some other thread then releases. The programmer also wants to program as if in cooperative concurrency: a thread executes until it performs an explicit synchronization operation (such as semaphore acquire or release), and then some other thread might execute. She should not need to reason about interleavings of instructions. Not only are interleavings difficult to think about, but they are also unsound, since they connote sequential consistency and today's machines are not sequentially consistent.

The compiler designer should not have to know anything about the concurrency libraries. She would like to pretend that there's only a single thread, and would like to ignore the nasty problem of weak cache consistency. Similarly, the compiler writer should not need to know the details of the logic used to prove programs correct. He only needs to know a concrete semantics for the compiling languages.

The Concurrent Permission Machine is the connecting thread of this top-to-bottom concurrency result: it provides the semantic interface to modularly compose the three main components of the framework. We model a concurrent program as a list of threads: each with a local-variable (or register-bank) state and each with a permission-map (partial function from address to read/write permission) and all sharing a single memory. The machine is equipped with a *schedule* determining the order of thread execution.

The structure of the entire project is as follows:

(top) Any C program, proven correct in Concurrent Separation Logic, runs correctly in the CPM.

(**Compiler**) The CompCert compiler preserves correctness of compiled programs with respect to the CPM semantics.

(bottom) Safe executions of the CPM, in the compiled program, run correctly on X86 machines.

Putting it all together, our main theorem is a top-to-bottom preservation of correctness.

Theorem 3.0.1 (Informal Main Theorem)**.** *A correct C program, compiled by CompCert, runs safely in an X86 machine.*

In the result we present here, we go down as far as assembly language, not machine language; this is because no one had formalized the assembly-to-machine level of CompCert. The preservation of *safety* we prove includes: no execution of undefined instructions or loading from inaccessible memory; no violations of the spinlock well-synchronized property. Owens [29] proved that a spinlock-well-synchronized execution behaves correctly on a TSO (total-store-order) weakly consistent multiprocessor. We have generalized Owens's definition of spinlock-well-synchronized to account for (1) semaphores that can be released by threads other than the one that locked them, and (2) creation of locks (from ordinary allocated memory, our MAKELOCK primitive) and (3) decommissioning locks (our FREELOCK which converts a lock back into ordinary memory); see section 7.2.

In principle, the same generalizations could be applied to Batty's definition of spinlock-well-synchronized for the Power architecture [5, pg. 178, Theorem 13], or to more modern definitions that might (for example) leverage the C11 standard to define an architecture-independent notion of semaphore-well-synchronized with make-lock/freelock.

In addition, we believe our results can be extended to correctness, meaning that any finite prefix of the trace of input-output communications satisfies a specification; the compiler is shown to preserve this traces (up to memory rearrangements by the compiler, section 4.1). In section 3.1.1 we formalize how this preservation of traces translates to correctness, but this has not yet been implemented in Coq.

In the rest of this chapter, we will present the details of the main theorem and describe the CPM formally.

3.1 Main theorem

In this subsection we will describe the main theorem of the top-to-bottom work. We present two top-to-bottom theorems for preservation of safety and preservation of correctness: the former is mechanized in Coq and the latter is formally proven in the following subsection but has not been mechanized as part of this work. We state the theorems in two parts: first we state the theorems and the proofs and, second, the Coq implementations of the theorem as well as the technical details of the definitions.

3.1.1 Formal definitions

Safety

The notion of *safety* we preserve means no execution of undefined instructions or loading from inaccessible memory, and no violations of the spinlock well-synchronized property.

Theorem 3.1.1 (Main Theorem, formal version of Theorem 3.0.1)**.** *Given a source program P satisfying a CSL specification S, and an x86 assembly language program Q obtained by CompCert compilation of P: Any execution of Q is safe (no undefined behavior) and well synchronized.*

Proof of Main Theorem. The full proof has been mechanized and can be found in the accompanying Coq code. The rest of the work expands on the different parts of the proof. The Coq statement of the theorem is presented in fig. 3.1 □

Correctness

Each part of the top-to-bottom proof (i.e., CSL soundness, compiler correctness and well-synchronization proof) is stronger than preservation of safety and, as we show

```
Theorem top2bottom_correctness:
    (* C program is proven to be safe in CSL*)
    ∀ (main:AST.ident), CSL_correct C_program main →

    (* C program compiles to some assembly program*)
    CompCert_compiler C_program = Some Asm_program →

    (* Statically checkable properties of ASM program *)
    ∀ (STATIC: static_validation Asm_program main),

    (* For all schedules, *)
    ∀ U : schedule,

    (*The asm program can be initialized with a memory and CPM state*)
    ∃ (m : mem) (cpm : CPM),
      initial_state Asm_program STATIC cpm m ∧

      (* The assembly language program
       is correct and well-synchronized *)
      spinlock_safe U cpm m.
```

Figure 3.1: Coq definition of our main theorem theorem 3.1.1.

here, can be composed to show a preservation of correctness. The proof presented below is not mechanized as part of this work and is left as future work.

Threads communicate by releasing and acquiring locks that control access to data regions. In terms of Concurrent Separation Logic, we say a program is *correct* if the data in such regions always satisfies an appropriate predicate—the *resource invariant*—when the corresponding locks are released. Therefore, any *observer* of the program—a thread who communicates with it by acquiring and releasing locks—will see only output that satisfies this correctness specification. Because our CSL has the appropriate partial-commutative-monoid ghost variables [19], resource invariants can specify general protocol-correctness properties.[1]

[1]Resource invariants in the original Concurrent Separation Logic [28] each individual lock release satisfies locally a resource invariant, but modern ghostly CSLs [19, 27] can connect together the entire history of interactions.

To model correctnes and not just safety we characterize the multithreaded program by its interaction with its context or an observer. We assume that it will interact with its context by acquiring and releasing a distinguished set of locks controlling some resources in the program. We assume these locks already exist at the moment main starts, and the program is allowed to synchronize with them.

We summarize a program's output by monitoring the data controlled by locks as they are released, and a program's input via data controlled by locks acquired. The event trace of locks acquired/released, and the contents of memory transferred by those locks, we preserve all the way from the top level (CSL) to the bottom level (assembly), so that we may state correctness properties of the assembly-language execution.

Theorem 3.1.2 (Correctness preservation)**.** *Let P, S and Q be given as in theorem 3.1.1. For any trace prefix T_Q of an execution of Q, there is an execution of P with trace T_P, which is equal up to some injection ($T_P \overset{j}{\hookrightarrow} T_Q$). In particular, any integer values transferred by releasing locks are equal in both programs.*

Moreover, for every lock l, defined in the precondition of main (in S) with some invariant R, every time l is realeased (in T_P and T_Q), the transferred resources satisfy R.

Proof. We look at how every part of the top-to-bottom work preserves traces (up to injection).

The instruction interleaving proof in section 7.1 constructs a new coarse-grain schedule and a new execution but preserves all synchronization operation in the same order. The transferred resources of each synchronization operation are also the same up to some reordering of memory.

The CPM simulations definition 5.1.1, given by the compile correctness specification, preserve synchronization operation in the same order and the transferred resources are equal up to the injection given by compilation.

The CSL proof, that P satisfies S, includes proving that every lock release satisfies the invariant of the lock. For any lock defined in the precondition of main, that invariant is exactly the one in S. □

3.1.2 Coq definitions

Figure 3.1 shows the Coq statement of our main theorem 3.1.1. We show how the code implements the theorem and explain the code below:

In what follows we explain how the code in fig. 3.1 implements theorem 3.1.1.

- "Given a source program P satisfying a CSL specification S ..." (lines 2-3):

  CSL_correct C_program: States that the program has been proven correct in CSL for some specification written in concurrent separation logic. This definition corresponds to the judgment $\Gamma \vdash_{\text{CSL}} \Psi : \Gamma'$ as described in section 6.1. We use CompCert's definition of program.

- "... and an x86 assembly language program Q obtained by CompCert compilation of P ..." (lines 5-9)

  - CompCert_compiler: States that the CompCert compiler translates C_program into the assembly program Asm_program.
  - static_validation: We validate the translation by statically checking a couple properties of the translated program Asm_program. All of them are known to be preserved by the compiler, but the fact that CompCert preserves them has not yet been proven in Coq. We leave removing these conditions as future work. These properties are:
    - ∗ limited_builtins: The program only uses the builtins we currently support: memcopy, mem_alloc, mem_free. These are the only builtins that CompCert inserts during compilation.

        * valid_mem: The initial memory has no dangling pointers.
        * ge_wd: The global environment is allocated in the initial memory.
        * main_ident_correct: The assembly program has an entry function named main.

- "...Any execution of Q is safe (no undefined behavior), well synchronized and correct." (lines 11-20)
    - initial_state: The program can be initialized with the initial memory m and the initial CPM cpm.
    - spinlock_safe: For all executions (and for all schedules, quantified in line 11), the initial state is safe and well synchronized, as defined in section 7.2.

3.2 The concurrent permission machine

The Concurrent Permission Machine is a language-agnostic operational semantics for concurrent programs. We use the CPM as our interface, at the C source level, between *proofs of source program properties* and *the operational semantics of the programming language*. It also serves as the operational model that allows *modular reasoning about compiler proofs*; as well as our interface, at the x86 assembly language level, between *permission safety* (derived from those proofs) and the *well-synchronized* property.

3.2.1 Overview

The main idea of a Concurrent Permission Machine is a simple one: annotate the small-step operational semantics–of C, of assembly language, or of any intermediate language–with per-thread permissions at every address. A thread is *stuck* if it tries to read address a without read permission, or write without write permission. No two threads ever have conflicting permissions to the same address, so races are impossible.

Acquiring a lock *increases* the thread's permissions at some set of addresses; releasing a lock *decreases* permissions. Which permissions are increased or decreased? Intuitively, the lock controls access to some shared data and the the thread should gain access to that data when it acquires the lock and loose the access when it releases it. It is the permissions of that data that increases or decreases. Of course, these permissions have no physical manifestation in the execution of a machine-language program, so we prove an erasure theorem. But we must wait until the very bottom level (of our proof) before erasing them, so as to be able to prove the kind of race freedom necessary on a weak-cache-consistent multicore processor.

We model a concurrent program as a list of threads, each with a local-variable (or register-bank) state, each with a permission-map (partial function from address to read/write permission), all sharing a single memory. The program is equipped with a *schedule* determining the order of thread execution.

To enable instantiations to C and assembly (or other languages), CPMs are built parametrically on top of *interaction semantics* [6], a common abstraction for languages operating over shared CompCert memories, each with its own code representation, thread-local state, and operational relation for internal steps. To support our proofs, we further developed Beringer et al.'s interaction semantics into MOIST semantics; see chapter 4. For CompCert′s languages (i.e., C and Asm), we use $\Psi \vdash_{\text{CompCert}} \langle \sigma, m \rangle \overset{\epsilon}{\mapsto} \langle \sigma', m' \rangle$ to refer to this small-step relation, where Ψ is the program; σ and σ' are thread-local states (a thread-local variable store in case of C, a register bank in case of assembly); m, m' are CompCert memories; and xϵ is a (possibly empty) trace of events that denote the nonatomic memory accesses performed. We use it to state theorems about the absence of races between nonatomic accesses. Thread-local execution proceeds ad infinitum, reaches a *Halted* state, or reaches an external function call. In the latter case—marked at_external—the CPM takes control, for example by executing a concurrency primitive (see below), and scheduling other threads. Internal execution is resumed by the primitive after_external, decorated by the (optional) return value.

The CPM supports external function calls for dynamic creation of threads and locks; thus we have a subset of C11 concurrency. A program can make these external function calls:

```
struct lock;
void spawn (void *func, void *arg); /* spawn a thread */
void makelock(struct lock *p); /* initialize a lock */
void freelock(struct lock *p); /* decommission a lock */
void acquire (struct lock *p); /* acquire a lock */
void release (struct lock *p); /* release a lock */
```

To specify the gain or loss in access rights that a thread experiences when making such external calls, CompCert and CPMs mark locations with *permissions*—instrumentation values that may take (in increasing order of permissiveness) levels None < Nonempty < Readable < Writable < Freeable [25]. In fact, CompCert's memory model associates permissions to each location that indicate the running thread's current access rights. A thread's small-step relation is *stuck* whenever a memory operation (such as load, store, free) is not supported by the thread's permission at the location in question. Thread-local execution does not alter existing permissions, with the exception of lowering it to None during stack frame deallocation. In contrast, external calls (and specifically, the concurrency primitives) may modify the permissions arbitrarily (though not greater than the max permissions). The CPM regulates permission transfers between different threads, maintaining *permission coherence* (in particular: the absence of conflicting write permissions by different threads), but also preserving safety of individual threads as these are compiled.

Example. Before moving on to the precise definitions of the CPM, let's attempt a first approximation of how the rule for Release should work. We review this example in the next section and the correct rule is shown in fig. 3.3.

The multithreaded CPM is guided by a *schedule* $\mho$; we write $i \cdot \mho$ for a schedule in which the ith thread is next to execute. If there are (so far) k spawned threads, the machine maintains a list $\vec{s}$ of k thread-states (local-variable sets) and a list $\vec{\pi}$ of k permission-maps; all threads share the memory m. To release a lock a, the current thread i must be at_external, ready to call Release with argument a. The machine must make sure that the lock is currently locked ($m(a) = 0$) and unlock it ($m[a \mapsto 1] = m'$).

$$\frac{\begin{array}{c}\text{at\_external } s_i = \text{Some}(\text{Release}, a) \\ m(a) = 0 \qquad m[a \mapsto 1] = m' \\ \text{guess } \delta \qquad \vec{\pi}[i \mapsto \delta/\pi_i] = \vec{\pi}'\end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}'), m'\rangle} \quad (\text{SIMPLIFIED RELEASE})$$

What permissions should the release transfer? Suppose a controls access to the linked list rooted at address p. Release should give up all permissions to that linked list—they are transferred to the lock. Unfortunately, different linked lists can be stored at p, so the data protected by a can be different every time the lock is released. In concurrent separation logic, we describe this with a resource invariant R; in any given memory, only one set of addresses[2] can satisfy R; this determines which addresses to transfer. In the top-to-bottom proof an oracle is constructed from the CSL proof. In the CPM the data is given by the oracle as a guess δ. The CPM updates the permissions of the current thread, to contain the old permissions updated with the permissions transfered δ/π_i.

[2]In general we do not require *precise* resource invariants, but the linked-list predicate happens to be precise.

3.2.2 Formal definition of the CPM

The CPM operates over states of the form $\langle \mho, (\vec{s}, \vec{\pi}, L), m \rangle$, where

$\mho$ is a (cooperative) schedule, a finite sequence of natural numbers indicating which thread to run next. We approximate infinite computations by quantifying over all finite prefixes.

$\vec{s}$ is a list of marked local states of the CompCert language the CPM is instantiated by; $s_i \in \{\text{Start}(f, a), \text{Run}(\sigma_i), \text{Blocked}(\sigma_i), \text{Resume}(v_i, \sigma_i)\}$, where σ_i is the ith thread's local state. $\text{Start}(f, a)$ denotes that the thread should start by calling function f with arguments a; $\text{Run}(\sigma_i)$ denotes that the thread is in the middle of sequential execution; $\text{Blocked}(\sigma_i)$ denotes that the thread has called a synchronization primitive, and is waiting for the CPM to execute it; $\text{Resume}(v, \sigma_i)$ means that an external call has returned value v to be fed back to the thread. A halted thread (one that has returned from its initially spawned function) is recognized by a *halted* predicate.

$\vec{\pi}$ is a list of permission map pairs, one pair for each thread (the components of these pairs are described below).

L is a function from address to option(option(permission)), indicating the state of each lock: $L(a) = \text{None}$ means that a is not a lock. Some(None) means that a is locked—permissions associated with the lock a are hence installed in the *Cur* component of m. Some(Some π) means a is unlocked and π is the permission that a thread would obtain by acquiring a.

m is the global memory, shared by all threads.

The CPM employs judgments $\Psi \vdash_{\text{CPM}} \langle \mho, (\vec{s}, \vec{\pi}, L), m \rangle \overset{\epsilon}{\mapsto} \langle \mho', (\vec{s}', \vec{\pi}', L'), m' \rangle$, as shown in fig. 3.2 and fig. 3.3.

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Acquire}, a) \quad m|_{\pi_i^2}(a) = 1 \\ m[a \mapsto 0] = m' \quad \text{guess } \delta \quad \delta/\pi_i = \pi' \quad \pi_i \oplus L(a) = \pi' \\ \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \quad \vec{\pi}[i \mapsto \pi'] = \vec{\pi}' \\ L[a \mapsto \text{Some } \emptyset] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Acq}_i\, a\, \delta}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \quad (\textsc{acquire})$$

$$\frac{s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Acquire}, a) \quad m|_{\pi_i^2}(a) = 0}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{AcF}_i\, a}{\mapsto} \langle \mho, (\vec{s}, \vec{\pi}, L), m\rangle} \quad (\textsc{acqfail})$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\;\sigma = \text{Some}(\text{Release}, a) \quad m|_{\pi_i^2}(a) = 0 \\ m[a \mapsto 1] = m' \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \quad L(a) = \emptyset \\ \text{guess } \delta \quad \text{guess } \delta_{\text{L}} \quad \delta_{\text{L}}/\emptyset \oplus \delta/\pi_i = \pi_i \\ \vec{\pi}[i \mapsto \delta/\pi_i] = \vec{\pi}' \quad L[a \mapsto \text{Some}(\delta_{\text{L}}/\emptyset)] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Rel}_i\, a\, \delta\, \delta_{\text{L}}}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \quad (\textsc{release})$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\;\sigma = \text{Some}(\text{mkLock}, a) \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \\ \pi_i[a \mapsto (\text{Nonempty}, \text{Writable})] = \pi_i' \quad \vec{\pi}[i \mapsto \pi_i'] = \vec{\pi}' \\ L(a) = \emptyset \quad L[a \mapsto \text{Some None}] = L' \quad m|_{\pi_i^2}[a \mapsto 0] = m' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \quad (\textsc{make lock})$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\;\sigma = \text{Some}(\text{freeLock}, a) \quad \vec{s}[i \mapsto \text{Resume}(0,\sigma)] = \vec{s}' \\ \text{guess } p \quad \pi_i[a \mapsto (p, \text{None})] = \pi_i' \quad \vec{\pi}[i \mapsto \pi_i'] = \vec{\pi}' \\ L(a) = \text{Some None} \quad L[a \mapsto \text{None}] = L' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}', L'), m'\rangle} \quad (\textsc{free lock})$$

$$\frac{\begin{array}{c} s_i = \text{Blocked}(\sigma) \quad \text{at\_external}\,\sigma = \text{Some}(\text{Spawn}(f,a)) \quad |\vec{s}| = j \\ \vec{s}[i \mapsto \text{Resume}(0,\sigma),\, j \mapsto \text{Start}(f,a)] = \vec{s}' \\ \text{guess } \delta \quad \text{guess } \delta' \quad \delta/\pi = \pi' \\ \delta' \oplus \pi' = \pi \quad \vec{\pi}[i \mapsto \pi',\, j \mapsto \delta'/\{\}] = \vec{\pi}' \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\text{Spa}_{ij}}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L), m\rangle} \quad (\textsc{spawn})$$

Figure 3.2: Concurrent Permission Machine, synchronization steps.

$$\frac{s_i = \text{Start}(f,a) \qquad \vec{s}' = \vec{s}[i \mapsto \text{Run}(\text{initialCore}(f, a))]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{start})$$

$$\frac{s_i = \text{Resume}(v, \sigma) \qquad \text{afterExternal}(\sigma, v) = \sigma' \qquad \vec{s}' = \vec{s}[i \mapsto \text{Run}(\sigma')]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{resume})$$

$$\frac{\begin{array}{c} s_i = \text{Run}(\sigma) \qquad \Psi \vdash_{\text{CompCert}} \left\langle \sigma, m|_{\pi_i^1} \right\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m' \rangle \\ \vec{s}' = \vec{s}[i \mapsto \text{Run}(\sigma')] \qquad \vec{\pi}' = \vec{\pi}[i \mapsto (\text{Cur}(m'), \pi_i^2)] \end{array}}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \overset{\epsilon_i}{\mapsto} \langle i \cdot \mho, (\vec{s}', \vec{\pi}', L), m'\rangle} \quad (\textsc{core})$$

$$\frac{s_i = \text{Run}(\sigma) \qquad \text{at\_external } \sigma = \text{Some}(f, \vec{x}) \qquad \vec{s}' = \vec{s}[i \mapsto \text{Blocked}(\sigma)]}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}', \vec{\pi}, L), m\rangle} \quad (\textsc{suspend})$$

$$\frac{(s_i = \text{Blocked}(\sigma) \wedge \text{at\_external } \sigma = \text{Some}(\text{Exit}, \_)) \ \vee \ \neg(0 \le i < |\vec{s}|)}{\Psi \vdash_{\text{CPM}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \mho, (\vec{s}, \vec{\pi}, L), m\rangle} \quad (\textsc{stutter})$$

$$\frac{}{\Psi \vdash_{\text{CPM}} \langle \text{nil}, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto \langle \text{nil}, (\vec{s}, \vec{\pi}, L), m\rangle} \quad (\textsc{done})$$

Figure 3.3: Concurrent Permission Machine: administrative and internal steps.

Without $\vec{\pi}$ and L, it would be a rather conventional model of a cooperative-concurrent machine, with threads $\vec{s}$ operating on a memory m. The type of thread-states s contains local variables and control-stack. The initial state has a single thread (at the beginning of the main function) and an empty lock pool.

Each thread s_i has its own pair π_i of (finite) permission maps: for each address l, $\pi_i(l)$ contains a *data* permission $\pi_i^1(l)$ and a *lock* permission $\pi_i^2(l)$. When scheduled, the thread's data permissions are installed as the current permission in CompCert's memory (operation $m|_{\pi_i^1}$ in rule core), thus regulating the dynamic access to shared locations.[3] In contrast, the thread's lock permissions are never installed in the CompCert memory of a running thread, but employed by the CPM to decide whether a thread's lock requests can be granted; in this context, Readable permission grants "permission to acquire/release" and Writable means "permission to decommission (freelock) the lock back into ordinary data". In the context of both data and lock permissions, Readable can be thought of as "Shared" and Writable as "Exclusive": data is read-only when shared and can be written when held exclusively, while locks can be acquired and released when shared and can only be turned back into data when held exclusively.

The lock pool L maps each address a to either None (not a lock), Some($\emptyset$) (when the lock is locked) or Some(π) (when the lock is unlocked and holds a resource whose permission map is π).

As in O'Hearn's original CSL [28], resource invariants R specify the memory regions (and assertions on them) regulated by locks. As in Gotsman *et al.* [13] and Hobor *et al.* [16], locks are dynamically creatable at addresses in memory; as in Hobor et al., resources are higher-order in that they may predicate over other locks and their resource invariants, or even over this lock and its resource invariant. To support higher-order impredicative resource invariants, we use a step-indexed model of CSL, but we want to avoid putting step-indexed resource invariants into the CPM itself. To avoid the use of higher-order resource invariants in the CPM, the CPM uses guesses δ, *partial permission maps* that specify which new permissions should be installed for every address in their domain. We write δ/π to mean a permission map in which δ "overrides" π. Like π, each δ is a pair of *data* and *lock* permission, at each address.

To find δ without knowing R (in particular: concurrency operations operating on the same lock may yield different permission transfers at different points in the

[3]The installation of permission maps happens at the granularity level of instructions / individual execution steps; this permits the reuse of the CPM framework in the setting of fine-grained interleaving, as described in section 7.1.

execution, as footprints are dynamic), the CPM uses *angelic nondeterminism*. A program is safe if there exists any sequence of angelic guesses δ such that the program does not get stuck. Chapter 6 describes how we prove that this sequence of guesses must exist: in particular, any CSL proof entails a sequence of guesses for which the CPM is nonstuck.

In two nonstuck executions of $\vdash_{\mathrm{CPM}}$ from the same start state but with different guesses δ, the values loaded and stored (and passed as parameters to functions) are the same. Guesses δ affect only the permissions of the memory. Insufficient permission can cause a stuck state, but extra permissions do not change values loaded or stored. Therefore the angelic nondeterminism does not affect any observable property of a nonstuck execution.

The operator $m|_{\pi_i^1}$ sets the current permissions of m to the *data* permission-map of the ith thread; respectively, $m|_{\pi_i^2}$ sets m to the *lock* permission-map.

We write $\pi \oplus \pi' = \pi''$ to indicate a kind of *join* on permission maps; it's a relation, not a function, because Readable$\oplus$Readable could "add up" to Readable or Writable.

The annotation ϵ is an *event trace* that records nonatomic memory operations such as acquires and releases of locks, creation and destruction of locks, and creation of threads. Later, we'll use this ordered sequence of operations to define *well-synchronized* executions.

Example. Following up from the example in the previous section, we address the changes in the RELEASE, to obtain the rule shown in fig. 3.3. First, the ith thread's state is Blocked(σ), which signals to the machine that the external function has not been executed. Second, when loading from m we adjust the current permissions of m to make the lookup according to the threads' lock permissions (π_i^2). Third, the CPM signals that the RELEASE has been executed by setting the ith thread to Resume($0, \sigma$). Fourth, the we have to guess δ_{L} representing the new resource held in the pool of unlocked resources and we must we make sure that the new permissions in the lock pool ($\delta_{\mathrm{L}}/\emptyset$) and in the thread ($\delta/\pi_i$) "add up" to the old permissions in the thread (π_i). Fifth, we require that the guessed $\delta, \delta_{\mathrm{L}}$ have not caused competing permissions. Finally, we modify L at address a to hold $\delta_{\mathrm{L}}/\emptyset$.

One might think that δ must uniquely determine δ_{L}; why do we need both? Remember that $\pi \oplus \pi' = \pi''$ is notation for a relation, which is not deterministic (two Readable permissions may add up to either a Readable or a Writable permission), so we use $\delta, \delta_{\mathrm{L}}$ to cope with this bit of nondeterminism.

Permission coherence

We say coherent($\vec{\pi}, L, m$) when π_i and π_j (or π_i and $L(l)$, etc.) do not give *competing* permissions at any address, nor treat any address as both *data* and *lock*. For example, $\pi_i^1(a) =$ Writable means thread i can do nonatomic reads/writes to address a; this is compatible with $\pi_j^1(a) \leq$ Nonempty (for $i \neq j$), meaning that thread j "knows that a is allocated," but not with $\pi_j^1(a) =$ Readable, nor with $\pi_j^2(a) \neq$ None. That is, if thread i sees a as a data location, then thread j cannot see the same address as a lock. Every execution of the CPM maintains coherence as an invariant.

Definition 3.2.1 (Competing permissions)**.** *Two permissions in the CompCert permission lattice, compete iff:*

1. *one of them is Freeable and the other is Nonempty or higher; or*

2. *one of them is Writable and the other is Readable or higher.*

Intuitively, Freeable means "no other thread even knows this data is allocated, so I can free it." Nonempty means "I know this data is allocated, even though I can't read or write it." With Nonempty *data* permission one can at least do pointer-equality tests; in C11, these are defined only on addresses of allocated data.

Definition 3.2.2 (Data-Lock coherent). *Two permissions in the CompCert permission lattice, respect the data-lock invariant iff: The lock permission is not Freeable; and if the lock permission is ≥Readable then the data permission ≤Nonempty.*

A Readable *data* permission gives permission to read, while a Readable *lock* permission gives permission to (attempt to) acquire the lock. To perform makelock, which converts a data block to a lock, one needs at least a *data* permission of Writable; the makelock takes away the Writable *data* permission (leaving either None or Nonempty *data* permission) and grants Writable *lock* permission. In turn, that *lock* permission can be split into several Readable parts, to be granted to various threads that want to contend for the lock. Eventually, these might be gathered together by a single thread into a Writable *lock* permission, which is enough to perform freelock, which converts a Writable *lock* permission to a Writable or Freeable *data* permission.

We lift definition 3.2.1 and definition 3.2.2 to permission-maps π. Recall that we write π_i^1 for thread i's *data* permission-map, and π_i^2 for thread i's *lock* permission-map. We write $L(a)^1$ for the *data* permission-map of the unlocked lock at address a, and $L(a)^2$ for its *lock* permission-map. Locked locks have empty permission-maps, since all their permissions have been (temporarily) added to the permission-map of the thread that acquired the lock.

Definition 3.2.3 (Coherent[4]). *We say that a state of the* CPM *machine is* coherent, *written as* $\text{coherent}(\vec{\pi}, L, m)$ *iff:*

1. *For all i, $\pi_i^1 \leq \text{Max}(m)$ and $\pi_i^2 \leq \text{Max}(m)$*
2. *For all $L(a)$, $L(a)^1 \leq \text{Max}(m)$ and $L(a)^2 \leq \text{Max}(m)$*
3. *For all i, j, permission-maps π_i^1 and π_j^2 are data-lock coherent; and if $i \neq j$, then π_i^1 and π_j^1 do not compete and π_i^2 and π_j^2 do not compete.*

[4]The Coq implementation details of this definition can be found in appendix A.3.1

4. *For all $L(a)$ and π_i, $L^1(a)$ and π_i^2 are data-lock coherent, π_i^1 and $L^2(a)$ are data-lock coherent, π_i^1 and $L^1(a)$ do not compete, and $L^2(a)$ and π_i^2 do not compete.*

5. *For all a, b, $L^1(a)$ and $L^2(b)$ are data-lock coherent; and if $a \neq b$ then $L^1(a)$ and $L^1(b)$ do not compete, and $L^2(a)$ and $L^2(b)$ do not compete.*

Safety

We use two notions of safety. First the intuitive one is as follows

Definition 3.2.4 (Weak CPM Safety). *A CPM state is* safe for k steps*, written* safe'_k*, when there exists a sequence of angelic guesses δ such that the machine does not get stuck within k steps.*

Using a schedule $\mho$ is a useful fiction, however oftentimes we like to express that the program safety does not depend on the schedule. It is not enough to quantify Weak CPM Safety over all possible schedules, since we want our safety to be preserved by execution. We need a slightly stronger notion of safety that quantifies over all schedules after every step

Definition 3.2.5 (CPM Safety). *A CPM state is* safe for k steps*, written* safe_k*, when there exists a sequence of angelic guesses δ such that the machine does not get stuck within k steps, even when it's allowed to change the tail of its schedule.*

Since a safe execution makes the "right angelic guesses", we can prove the following lemma

Lemma 3.2.6 (Safety implies coherence). *If a program is safe for all k, then every state in the execution is coherent.*

3.2.3 Generality of the CPM

The C11 standard defining the behavior of concurrent C programs has the following design principles:

- Memory locations can be classified as either nonatomic (accessed by normal operations in a well-synchronized manner) or atomic (accessed by synchronization operations, not necessarily well-synchronized).

- Compilers should optimize nonatomic operations.

- Ill-synchronized accesses to nonatomic locations (i.e., data races) lead to undefined behavior.

- Between synchronization operations, code in a thread is executed as if it were sequential.

- Correctly written code should execute correctly on any processor, regardless of its relaxed memory model.

- Correctly written code should execute correctly on any processor, regardless of its relaxed memory model.

The CPM gives an operational semantics for concurrent programs that obeys these principles, making it well suited for modeling lock-based concurrent programs in C and any other language that adheres to the same principles. The CORE rule of fig. 3.3 incorporates the sequential semantics of the language into the machine, and the remaining rules give semantics to the atomic operations. Because the CPM uses cooperative scheduling, the behavior of a thread between synchronization operations is exactly its sequential behavior, and compilers can optimize code in between synchronizations as if it were sequential.

In chapter 6 we show that the CPM serves as an operational model for Concurrent Separation Logic. But it models permission coherence more generally than just CSL. For example, Gu *et al.* describe a refinement method for proving correctness of shared-memory concurrent programs [15] with a “push/pull” model for releasing and acquiring locks. The *push* and *pull* can be modeled as angelic permission transfers in the CPM though it is unknown whether they can be modeled in concurrent separation logic.

To demonstrate that our CPM is more general than our specific concurrent separation logic, we show an example of a program that our CPM can model even though our CSL front-end cannot. The logic presented in chapter 6, and many others similar concurrent logics (e.g., [28], [16]), will fail to prove the correctness of the following example:

```
(thread 1)          (thread 2)
p[0]=x;             if (?)
release (A);        then acquire (A);
release (B);        else acquire (B);
                    y=p[0];
```

Some CSLs fail to reason about this example because we do not know whether the resource p[0] is transferred through lock A or lock B. Some CSLs with ghost state can prove the program correct: in fact, as Jung *et al.* [19] have shown, ghost state can be used to incorporate general rely-guarantee reasoning into CSL, by creating a more general notion of invariant that is not tied to atomic accesses to a specific location. In this case, the `release` and `acquire` rules must be interpreted as a different kind of atomic access: one that allows interaction with the global invariant. The above example can also be proven correct in a rely-guarantee logics such as VCC [9] or Iris [19]. Regardless of the means used to verify the program, it can still be executed in the CPM: for each choice of lock, there exists a CPM execution that performs the appropriate permission transfer.

The current presentation of the CPM is limited to Semaphores (coarse grain synchronization); however, we believe that it can be extended to support other C11 atomic operations in modes such as SC and release/acquire

Chapter 4

MOIST simulations and semantics of CompCert

The simple code in fig. 4.1 communicates with its environment in two main ways: (1) it takes an address as input, and (2) it reads from and writes to this location to increment the value stored there. We will see how the specification of CompCert prevents us from reasoning about such programs, either as compilation units or as external functions, and we will show how to extend the specification of a compiler to lift these limitations.

First, CompCert can't give any guarantees about compiling the code in fig. 4.1 because it is not a complete program. It is reasonable to expect that the function

```
int *buff;
void remember(int *p){
    buff = p;
}
void incr(void){
    ++(* buff);
}
```

Figure 4.1: The function remember records the address of some buffer, and incr increments it by one.

```
Inductive initial_state (p: program): state → ℙ :=
  | initial_state_intro: ∀ b f m0,
      let ge := Genv.globalenv p in
      Genv.init_mem p = Some m0 →
      Genv.find_symbol ge p.(prog_main) = Some b →
      Genv.find_funct_ptr ge b = Some f →
      type_of_fundef f = Tfunction Tnil type_int32s cc_default →
      initial_state p (Callstate f nil Kstop m0).
```

Figure 4.2: The initial_state in C and Clight describes a call to main. It also enforces that it takes no arguments (Tnil) and returns an integer (type_int32s).

remember runs safely, given some assumptions (e.g. ∗p is a valid address in memory). Unfortunately, CompCert's semantics assumes that a program starts executing with a call to main() with no arguments. CompCert characterizes the initial state by a predicate initial_state: state → ℙ that takes no additional arguments. You can see an instantiation of the predicate for Clight in fig. 4.2. So, even though CompCert correctly compiles the code, its specification gives no guarantees of any execution other than the one that starts by calling main with no arguments.

Second, imagine that the example in fig. 4.1 describes not the program being compiled but the semantics of two system calls remember and incr. Suppose CompCert compiles some program that calls incr(); then the compiler's specification gives no guarantee about the behavior of the compiled code. Indeed, CompCert's semantics allows calls to external functions that are assumed to be correct but, unfortunately, that specification of correctness is too strict. It assumes that the function's behavior is fully determined by (1) the state of memory, (2) the function arguments and (3) the events produced by the function.[1] The behavior of incr also depends on the value

[1] Leroy [23] claims that "inputs given to the programs are uniquely determined by their previous outputs", but this is not exactly correct. A more accurate representation of CompCert's specification would be to say "inputs given to the programs are uniquely determined by their most recent outputs". Indeed, the semantics of external calls extcall_sem : **Type** := env → list val → mem → trace → val → mem → ℙ are determined by the environment, the arguments to the call, the current memory and produce a trace, a return value and a return memory. As we will see in section 4.5, it would be much stronger to determine inputs based on all historic outputs.

```
Inductive event: Type :=
  | Event_syscall:
        string → list eventval →
        eventval → event
  | Event_vload:
        memory_chunk → ident →
        ptrofs → eventval → event
  | Event_vstore:
        memory_chunk → ident →
        ptrofs → eventval → event
  | Event_annot:
        string → list eventval → event
```

```
Inductive eventval: Type :=
  | EVint: int → eventval
  | EVlong: int64 → eventval
  | EVfloat: float → eventval
  | EVsingle: float32 → eventval
  | EVptr_global: ident → ptrofs → eventval.
```

Figure 4.3: The events in CompCert

in buff (which for system calls will not be in the program's accessible memory), so it is not correct, according to CompCert's specification. Certainly, incr could expose the pointer stored in buff as part of its trace but CompCert events, shown in fig. 4.3, can only contain integers, floats or pointers to global variables.

Third, in the CompCert semantics, the entire behavior of each external function call is bundled into one big step. Looking at an execution, internal steps and external function calls are uniform. This consistency is very useful when reasoning about the compilation of the program, where we want to abstract external calls. Nevertheless, when reasoning about a program in a context, it is more useful to replace the big step external calls with their small step semantics. Regrettably, the specification of CompCert does not even guarantee that the source and target programs call the same external functions. In theory, CompCert could replace an external function call with internal steps as long as they had the same (possibly empty) trace. In practice, obviously, CompCert does not do that, as this guarantee is not exposed in its specification.

Fourth, the correctness of CompCert is stated as a semantic preservation theorem, where the traces are the preserved behavior and the proof uses forward simulations[2].

[2]Forward simulation and determinism of the target language implies bisimulation and thus preservation of behavior.

At least two other works ([33], [18]) have proposed alternative CompCert specifications that make the simulations an exposed feature of the compiler's specification. In these papers, the authors view CompCert correctness modularly as a thread-local or module-local simulation, and they recover a simulation of the global program later. Moreover, from the exposed simulations, they can recover the relation between memories in source and target, another very useful feature in compositional compilers, which seems to be a key feature in compositionally. Following this line of work, we propose to expose the simulation as the specification of the compiler, deriving semantic preservation as a corollary.

Finally, in each CompCert intermediate language, every state of the small-step semantics contains exactly one memory. To reason parametrically about how a thread of execution evolves its memory, we need a uniform way to access the memory component of the small-step state.

We move, then, to lift these limitations according to the following richer notion of specification:

Definition 4.0.1 (MOIST simulations)**.** *We say that a compiler's specification uses Memory, Observable, Injectable and Startable Trace (MOIST) simulations if they satisfy the following:*

- *Memory: All intermediate languages have a unified memory model* mem*, and each language L_1 has a function* get_mem: state L_1 → mem*, that exposes the memory of a state. The simulation describes the relation between memories before and after compilation.*

- *Observable: Similarly, all intermediate languages are outfitted with a function* at_external *that identifies states about to make an external function call. For every language L_1,* at_external: state L_1 → **option** (f_ext, args) *returns the exter-*

	Percent change	Number of lines changed
Arguments in main	1.6%	3660
Injectable Traces	0.5%	1141
MOIST Semantics	0.2%	466
MOIST Simulations	1.3%	2911
Total	3.6%	8178

Table 4.1: Measures of changes to CompCert: changes are calculated from the number of lines added as given by running git diff between our code and the commit of CompCert we branched off from. For each feature, an estimated number of lines is provided.

nal function being called and its arguments. The simulation preserves external calls.

- *Injectable: The execution trace supports events that can describe locations in memory (i.e., pointers). Compilation may rearrange memory, which CompCert describes as an* injection, *so the trace will be preserved up to these injections. The simulation shows that the injection relating traces in source and target executions is the same injection that relates their memory. We call these new events* memory events.

- *Startable: The execution of a program can start in any of its public functions, including* main, *taking arguments.*

It is worth noting that, even though we require a unified memory model, in practice, a language can use a different memory model (or none at all) as long as it can construct a memory from its state with get_mem. In practice all CompCert languages use the same memory model, described in section 4.1, which we will refer to as mem from now on. Nevertheless, a future language could use *juicy memory* as in [3] or abstract state in [14] since a mem can be derived from them.

In the rest of the chapter, we describe how we develop MOIST specifications for CompCert. We first describe how to generalize initial_state to make the simulations Startable. Second we describe how to add memory events to CompCert. Then we show how to extend the semantics for every language in CompCert to include get_mem and at_external functions and, finally, we show how to put everything together in MOIST simulations for CompCert, which proves the following theorem:

Theorem 4.0.2 (CompCert MOIST specification). *Given a source program P and an x86 assembly language program Q obtained by CompCert compilation of P, the CPMs of the two programs are in a MOIST simulation relation.*

The changes described here represent only a 3.6% change to CompCert, as measured by running git diff in CompCert before and after our changes. The amount changed for every feature proposed is described in table 4.1.[3] Approximately 10% of all new additions are specifications and the rest are proofs.

4.1 Memory model and memory injections

All intermediate languages in CompCert use the same memory model [26]. This is a key feature of the CompCert's proof of correctness, since it facilitates reasoning about how the compiler lays out stack frames in memory. In this section we briefly describe the memory model and the *memory injections* which describe the evolution of the memories (particularly stack frames) through compilation.

CompCert's memory is represented by a two dimensional array of bytes, indexed by a block reference and an integer offset. Each single block is a one dimensional array that can be used differently for each intermediate language; in Clight, every stack-allocated variable is in a separate block while in Mach every stackframe sits in a single block. The bytes in each location are represented by abstract values such as integers, floats, pointers; or undefined, if its location has not been initialized.

[3] All these measurements are made with respect to a commit in CompCert from May 21 (f047fcb7852ff58c0c62f10d41f91f3f88552780)

	block	$::=$	$\mathbb{N}$
Blocks	b	$::=$	block
Offsets	ofs	$::=$	$\mathbb{Z}$
Values	v	$::=$	Vundef | Vint n | Vfloat n | Vptr b ofs
mem	m	$\in$	$\mathbb{N} \rightarrow \text{option}(\mathbb{N} * \mathbb{Z})$
perm	p	$::=$	None | Nonempty | Readable | Writable | Freeable
Max perm	Max	$\in$	$\text{mem} \rightarrow (\text{block} * \mathbb{Z}) \rightarrow \text{Perm}$
Cur perm	Cur	$\in$	$\text{mem} \rightarrow (\text{block} * \mathbb{Z}) \rightarrow \text{Perm}$
injection	j	$\in$	$\text{block} \rightarrow \text{option}(\text{block} * \mathbb{Z})$

Figure 4.4: CompCert memory model.

Every location in memory is also outfitted with permissions that regulate how a program can interact with memory. These permissions are cumulative, so each one grants the capabilities of all the permissions below:

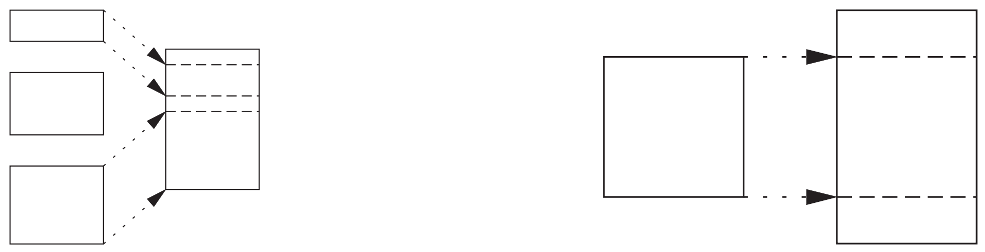
Freeable:	can free the location
Writable:	can write to the location
Readable:	can read to the location
Nonempty:	can only compare pointers to the given location
None:	Can't interact with the location

For example, a program may only free a piece of memory if it has Freeable permission of that location; but it can read any location where it has at least Readable permission. It is intended that a thread may gain or lose permissions to a location by doing synchronizations such as acquire and release; to keep track of this, there is a Cur (current) permission at each address. But to prove the correctness of a certain optimization (constant-folding the load of a global read-only variable), the semantics also has a Max permission at each address, above which Cur can never go. For example, all global variables have at most Writable Max permissions so they might change, but they can't be freed. Cur permissions represent local abilities of a module or thread. For example, a thread will have no permissions for variables in the stack of another thread, whose addresses are not taken. But no matter how much more Cur permission a thread gains by acquiring a lock (etc.), it can never go above the Max permission for the address.

The compiler must preserve the behavior of a compiling program, but it might reorder and change its accesses to memory; an optimization might swap two consecutive memory allocations, or it might add new ones during spilling. Even so, the memory can't change too much, lest it change the observable behavior of the program. In the CompCert memory model, the compiler can affect memory blocks by reordering them, deleting them, changing their internal offsets or even merging two of them. Leroy and Blazy [26] call these transformations *memory injections* and, to prove memory-changing optimizations are correct, they show that the observable behavior of a correct program is invariant under memory injections. We will describe injections further below.

Most CompCert passes don't change the memory behavior of the program. For example, the phase Cshmgen simplifies control structures, but preserves all memory accesses. In that way, the executions of the program before and after the pass display the same memories at every point. These are *equality passes*. Some passes might increase the size of a block by, for example, spilling variables to the stack, but they will not reorder the blocks. We call these *extension passes* fig. 4.5(b). Finally there are those that delete, reorder and merge blocks, such as the Csharpminor to Cminor pass which merges all stack-allocated variables into single block (the stack block). These are the *injection passes* fig. 4.5(a). All passes can be shown to be injections [33], but it is much easier to treat equality and extension phases as special cases.

An injection is described by a mapping j : block $\rightarrow$ **option** (block $*$ Z), that determines if a block is mapped and, if so, to which block and with what offset. For example in the Csharpminor to Cminor pass, where local variables are coalesced into one stack block, each variable will be offset such that it doesn't overlap with the others as shown in fig. 4.5(a). Beyond a permutation of memory, an injection induces



(a) Memory injection (b) Memory extension

Figure 4.5: Memory transformations of CompCert. [3]

$$\frac{}{\mathsf{Vint}(n) \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathsf{Vint}(n)} \qquad \frac{}{\mathsf{Vfloat}(n) \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathsf{Vfloat}(n)}$$

$$\frac{F(b_1) = \mathrm{Some}\ (b_2, \delta) \qquad i_2 = i_1 + \delta}{\mathsf{Vptr}(b_1, i_1) \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathsf{Vptr}(b_2, i_2)} \qquad \frac{v_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} v_2}{v_1 \overset{j}{\hookrightarrow} v_2} \qquad \frac{}{\mathsf{Vundef} \overset{j}{\hookrightarrow} v_2}$$

(a) Strong ($\lhook\joinrel\twoheadrightarrow$) and simpl ($\hookrightarrow$) value injections.

$$\frac{}{p \overset{j}{\hookrightarrow} p} \qquad \frac{}{\mathsf{None} \overset{j}{\hookrightarrow} \mathsf{Freeable}}$$

(b) Permission injections

Figure 4.6: Value and permission injections

relations between the source and target contents of memory and the permissions. We abuse the notation $\overset{j}{\hookrightarrow}$ to denote all these relations induced by injections, including several others we define in the following sections.

An injection imposes a relation between values before and after the compiler pass, as described in fig. 4.6(a). In the stronger injection, constants are preserved and pointers are renamed according to the injection. In the simple injection, undefined values are also allowed to map to any concrete value because, in compiler passes such as register coalescing, uninitialized local variables can map to initialized ones with concrete values. Similarly, the injection imposes a relation between permissions as shown in fig. 4.6(b).

Definition 4.1.1 (memory injection). [4] *Two memories m_1 and m_2 are* injected *by j ($m_1 \overset{j}{\hookrightarrow} m_2$) if, for all locations b_1 mapped by j $b_1 = Some\ (b_2, ofs)$:*

- *Permissions are injected on all offsets x:*

$$Max\ m_1\ (b_1,\ x) \overset{j}{\hookrightarrow} Max\ m_2\ (b_2,\ x + ofs) \quad and$$

$$Cur\ m_1\ (b_1,\ x) \overset{j}{\hookrightarrow} Cur\ m_2(b_2,\ x + ofs)$$

- *Values are injected on all visible offsets x:*

$$Cur\ m_1\ (b_1,\ x) \geq Readable \longrightarrow m_1\ (b_1,\ x) \overset{j}{\hookrightarrow} m_2\ (b_2,\ x + ofs)$$

- *Only allocated blocks are mapped and the image of mapped blocks don't overlap.*

- *mi_representable and mi_perm_inv [3]*

4.2 Passing arguments to main.

CompCert can compile programs where main takes arguments, but its correctness theorem gives no guarantees about their translation. That is because its semantic model assumes that main takes no arguments (See fig. 4.2). But real C programs can take up to two arguments: argc, the argument count, and argv, the argument vector. Also, all executions in the semantics of CompCert start with a call to main(), even though main is nothing but an agreed upon term for startup. We aim to generalize this to any function, not just main(). In this section we define a new predicate entry_point, generalizing initial_state (fig. 4.2), that characterizes starting states which includes calls to main with arguments and calls to any other function.

[4]We omit a couple extra properties such as *not overlapping* and *memory alignment*.

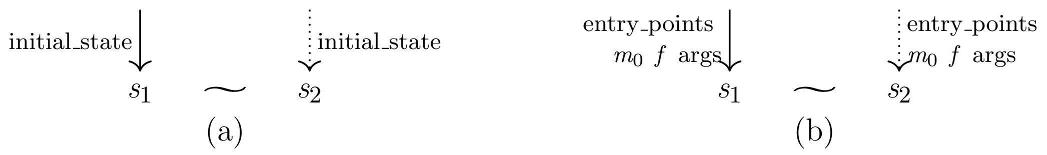


Figure 4.7: Entry simulation diagrams: (a) if s_1 is an initial state for the source program, then there exists some state s_2 that is related to s_1 and is an initial state for the compiled program. (b) Just like the diagram for initial states, but it generalizes and exposes the initial memory m_0, the entry function f and the arguments args. The entire simulation is parametric on the realation $\sim$.

Passing arguments to the entry-function is particularly important for our work with concurrency because spawning new threads behaves much like starting a program by calling main. The library function that spawns a new thread (e.g. pthread_create(foo)) must create a new stack, push the arguments to stack, and then call foo just like a program initialization would do. Our new predicate entry_point is general enough to capture both process-start and thread-start, as well any kind of incoming function call that follows the appropriate calling convention.

It is worth pointing out how important it is to allow newly spawned functions to take arguments. If we restricted our semantics to spawning threads with no arguments, threads wouldn't be able to share pointers (or would have to do it clumsily through global variables) and thus they would all execute in disjoint pieces of memory with no communication. That would be a much easier and less interesting result.

Once we define a new starting point for executions, we must prove that compilation preserves the predicate entry_point (fig. 4.7(b)) in the same way that CompCert's simulation preserves initial_state (fig. 4.7(a)). The proof largely follows the simulation of internal function calls which is already proven in CompCert, so we omit the details here. However, some interesting relevant details are presented later in section 4.2.2.

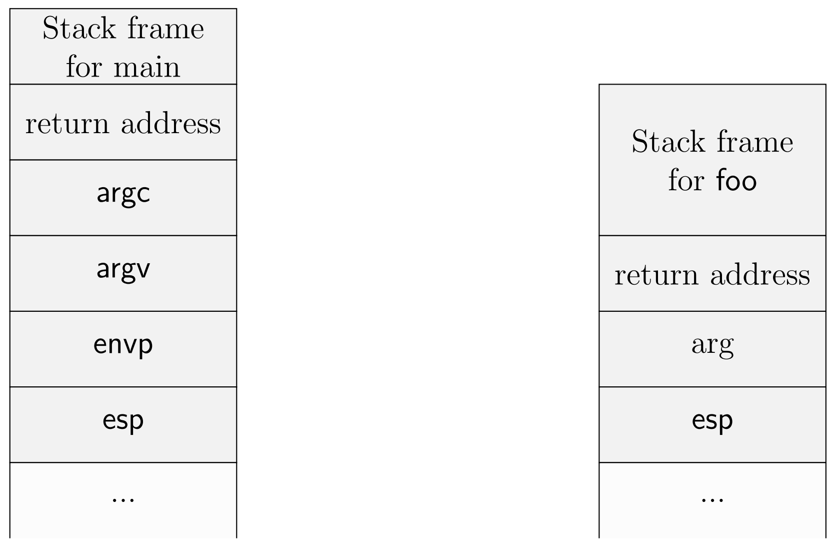
4.2.1 The prestack and the initial memory

When execution starts, CompCert semantics assumes that the stack is empty and the memory contains only the global variables. However, in reality, when main starts executing, there is more content already pushed on the stack and in memory that is particularly important to argument passing. Part of the entry_point predicate is to describe this initial state of memory, as we explain below.

Let's take, as an example, the moment main is called from the initialization function _libc_start_main. At this point the top of the stack will contain the return address and the arguments to main (beyond the first K that are passed on registers; on some RISC machines, $K = 4$, while on the x86-32, $K = 0$). We call that tip of the stack *prestack*, which is depicted in fig. 4.8(a), is relevant to the execution of main, and does not correspond to any function in the program. The rest of the memory, at that point, also contains the NULL-terminated argument vector, all the global variables, and possibly stacks of other threads or other initialization functions such as start. We call the entire memory at this point the *initial memory*. Our new predicate entry_point, unlike an empty stack, describes how arguments are set up in the prestack and, different from an almost empty initial memory, it allows memories to have arbitrary things.

As mentioned before, spawning a thread behaves like executing main in many ways. For instance, the stack of a thread, before the spawned function executes, looks just like a prestack before calling main as shown in fig. 4.8(b). Indeed, when pthread_create starts a new thread, it sets up the stack to pass arguments to the spawned function. The initial memory, at this point, contains the stacks of other threads and all other memory used by their executions. entry_point is general enough to characterize this prestack and initial memory too.

If we could pass all arguments on registers, we would not need to reason about the prestack at all. Unfortunately, in architectures such as $x86$ in 32-bit mode, all



(a) Stack at the start of main. (b) Stack created by pthread_create() to start a thread running foo.

Figure 4.8: Prestacks examples for X86-32 architecture: (a) Stack right before main executes. (b) Stack right before a function foo is executed in a new thread. Notice that in X86-64 (or ARM, etc.), the first K arguments will be passed in registers.

arguments are passed on the stack. As the comments in the CompCert code put it, "Snif!" [24]. Even architectures that allow argument passing in registers, such as $x86$ in 64-bit mode, have a limited number of registers and will pass arguments on the stack after those run out. Consequently, if we want to describe argument passing for entry functions in general, we must describe the prestack.

The characterization of the prestack is language-dependent, and we will describe it in greater detail in the next subsection.

4.2.2 The entry_point: a more permissive starting state

The predicate entry_point: $\mathsf{mem} \to \mathsf{state} \to \mathsf{val} \to \mathsf{list\ val} \to \mathbb{P}$ takes an initial memory m_0, an initial state s, a pointer to the entry function fun_ptr of type val, and a list of arguments args. This predicate is language-dependent, and is divided into three parts:

1. Checks that the global environment genv is allocated correctly. It also makes sure that the pointer fun_ptr points to a function in genv.

2. Checks that memory m_0 is well formed. That is, it contains no ill-formed pointers to invalid addresses. CompCert generally maintains that well-formed programs do not create dangling pointers.[5]

3. Checks that arguments are well-formed. Among other things, they have the right types for the function being called, they have no ill-formed pointers, they fit in the stack, and they correspond to the prestack.

In the rest of this section, we explore the definition of entry_points for different languages and, when interesting, we explain how we prove that different CompCert passes preserve the predicate as in fig. 4.7(b).

C frontend

All of the C-like languages (*Clight, Csharp, Csharpminor*) have similar entry_point, so we present here the one for Clight in fig. 4.9. Lines 4-6 ensure that the environment is allocated in memory and that it contains the function f with the right type signature. Line 7 states that the initial memory has no dangling pointers. Lines 8-11 say that the arguments have the right type and have no dangling pointers. The predicate bounded-args, enforces that the arguments fit in the stack, which is architecture dependent (generally around 1 Gigabyte). Finally, the entry state, in line 13, is defined as a call to f with an empty continuation.

In CompCert, continuations abstract the program's call stack with each Kcall describing a stackframe. Standard CompCert describes the "stop" continuation as simply Kstop without an argument. This does not permit description of a prestack

[5]A well-formed program, should not compare, read or write to invalid pointers. Hence, dangling pointers behave semantically as undefined values and could be modeled that way.

```
Inductive entry_point (ge:genv): mem → state → val → list val → ℙ :=
| initi_core: ∀ f fb m0 args targs,
        let sg:= signature_of_type targs type_int32s cc_default in
        type_of_fundef (Internal f) = Tfunction targs type_int32s cc_default →
        Genv.find_funct_ptr ge fb = Some (Internal f) →
        globals_not_fresh ge m0 →
        Mem.mem_wd m0 →
        Val.has_type_list args (typlist_of_typelist targs) →
        vars_have_type (fn_vars f) targs →
        vals_have_type args targs →
        Mem.arg_well_formed args m0 →
        bounded_args sg →
        entry_point ge m0 (Callstate (Internal f) args (Kstop targs) m0).
```

Figure 4.9: The entry_point predicate in Clight

frame containing arguments to main. Consequently, during later phases of compilation, we cannot design a simulation relation that properly relates Kstop to the prestack frame with arguments allocated in memory. It is precisely for this reason that we use Kstop targs instead of Kstop; from the types of the arguments targs, we can determine the (architecture-dependent) shape of the prestack. In our model, when Kcall gets translated to a predicate Stackframe that describes a stack frame, Kstop will be translated to Prestack, which is a special kind of Stackframe, depicted in fig. 4.10.

Register transfer languages

In the Cminorgen phase, CompCert coalesces all function variables into a stack frame. Some functions might get empty stack frames (i.e., a zero-sized memory block), if none of their variables have their address taken. These stack frames are important, even the empty ones, because that is where spill variables will be written after register allocation in the Allocation phase. We follow suit and create an empty stack frame for _start(). The stack is empty because the compiler has not yet decided which arguments will be passed in memory and which in registers. Even for architectures that pass all arguments in memory, this is not done until the Stacking pass. Hence, for languages

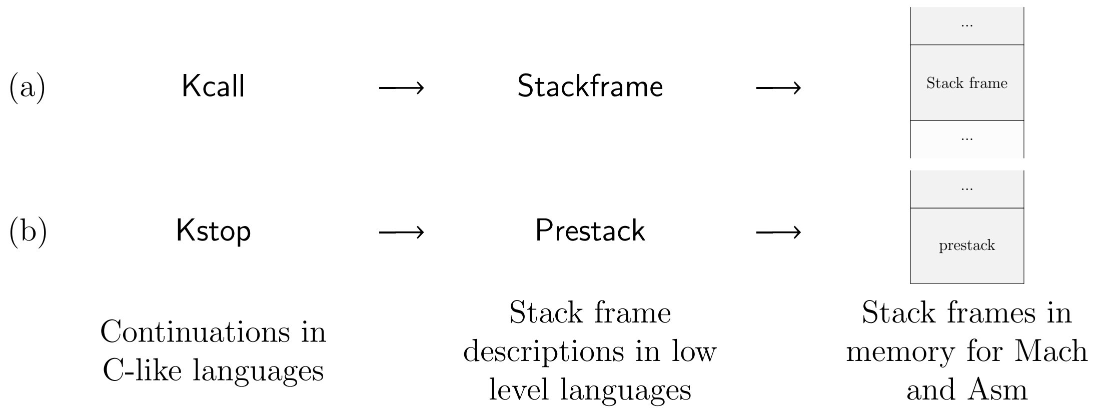

(a)
Kcall
Stackframe
...
Stack frame
...
(b)
Kstop
Prestack
...
prestack
Continuations in C-like languages
Stack frame descriptions in low level languages
Stack frames in memory for Mach and Asm

Figure 4.10: Abstractions of stack frames become more concrete through compilation. (a) Kcall is a high level abstractions of stackframes for C-like languages. It gets translated to Stackframes, which are low level descriptions of stack frames. Finally the stack is laid in memory. (b) Kstop, is a concise, high level, description of the the prestack. It gets compiled to Prestack, which is a special type of Stackframe, and finally laid in memory

between Cminorgen and Stacking (Cminor, CminorSel, RTL, LTL, Linear), there is an extra line in entry_point to make sure that the empty stack frame is allocated:

Mem.alloc m0 0 0 = (m1, stk)

The rest of the predicate is almost identical to the one in fig. 4.9.

These languages have a list of stack frame descriptors (called Stackframe), instead of continuations. Accordingly, the prestack is characterized by the predicate Prestack, a frame descriptor with just enough information to know the size of the prestack and where main should return. Prestack is generated from Kstop and, once the Stacking pass is done, it is translated to an actual prestack as shown in fig. 4.10.

Machine languages

In the machine languages (Mach and Asm), a function expects a certain shape from the stack frame of its caller. We replace the empty stack frame allocation with the construction of the prestack as shown in fig. 4.11. The function stack-defs is an architecture dependent function that calculates the layout of the stack and returns the size stk_sz, the offset of the return address ret_ofs, and a back link to parent frame

```
1          ⋮
2          let '(stk_sz,ret_ofs,parent_ofs) := stack_defs (fn_sig f) in
3          Mem.alloc m0 0 stk_sz = (m1, spb) →
4          let sp:= Vptr spb Ptrofs.zero in
5          store_stack m1 sp Tptr parent_ofs Vnullptr = Some m2 →
6          store_stack m2 sp Tptr ret_ofs Vnullptr = Some m3 →
7          make_arguments (Regmap.init Vundef) m3 sp
8                         (loc_arguments (funsig (Internal f))) args = Some (rs, m4) →
9          ⋮
```

Figure 4.11: Part of the entry_point predicate in Mach

parent_ofs. The last two values are unused, but the stack must have space for them. Line 3 allocates the stack of the correct size. Lines 5-8 store the return address, the link to the parent and the arguments in the stack.

Summary: entry points

In summary, we have augmented the CompCert semantics to permit the entry point to be a function of more than zero arguments; in a memory that can contain more than just the extern initialized global variables. These changes also required augmenting the Kstop continuation of the Clight language, to (abstractly) describe the initial stack frame.

4.3 Memory events.

The observable behavior of the CompCert semantics (for all languages) is a trace of events, as described in fig. 4.3. It records interactions with the outside world. For example, the results of a read system call will record an Event_syscall together with the name of the system call, its parameters, and its result. Lets consider a system call putbuf(**void** $*$a) that takes a pointer to a buffer and reads from it. In fig. 4.12(a), a points to the value 3, and putbuf then reads that value. What other contents x can a program put in a and communicate to putbuf?

We already mentioned that x must not be a pointer in CompCert (other than a global pointer), such as in fig. 4.12(b), “because these are not preserved literally during compilation” [23]. In fact, for this reason, CompCert does not have any event that exposes memory state or pointer values. Unfortunately, this limitation rules out several reasonable programs such as one calling incr (fig. 4.1) and the one in fig. 4.12(b). Moreover, these events are poorly suited to express any sort of shared memory interactions, such as concurrency or separate compilation. For example, a call to pthread_mutex_ulock(l) not only changes the state of the lock l, but also conceptually it gives away control to the data in memory protected by l. The location and the content of the data might expose memory state, so it cannot be expressed in CompCert semantics.

Can x be an uninitialized value? The answer here is also no. It turns out that uninitialized values also “reveal memory state” in a subtle way that we explain with the example in fig. 4.12(c). In the source code x is not initialized so, in the semantics of Clight, the content of a is Vundef. However after register allocation x and y might share a register, in which case the content of a will be 3. It is certainly reasonable to require that programs not communicate uninitialized variables to the external world as that can be a security risk. But output is not always going to the external world. If putbuf were a verified library, it would be acceptable to reveal undefined values to it. For another example, consider the code in fig. 4.12(d). In this case the code is part of a multithreaded program communicating through a semaphore l. In this case, again, it would be acceptable to reveal undefined values.

It is worth pointing out that we could forbid undefined variables from the trace. This would indeed make the compiler specification simpler and it’s verification easier. We can also describe and verify those properties of our programs using VST [3].

```
x = 3;            y = 3;            y = 3;            y = 3;
*a = x;           x = &y;           z = y + 1;        z = y + 1;
putbuf(a);        *a = x;           *a = x;           *a = x;
                  putbuf(a);        putbuf(a);        pthread_mutex_ulock(l);

      (a)               (b)               (c)               (d)
```

Figure 4.12: Four excerpts of code passing output through buffer a.

```
Inductive event: Type :=                Inductive mem_effect: Type :=
  ...                                   | Write : ∀(b : block) (ofs : Z)
  | Event_acq_rel:                                    (bytes : list memval), mem_effect
        list mem_effect →               | Alloc: ∀(b : block)(lo hi:Z), mem_effect
        delta_perm_map →                | Free: ∀(l: list (block ∗ Z ∗ Z)), mem_effect.
        list mem_effect → event
  | Event_spawn:
        block →
        delta_perm_map →
        delta_perm_map → event.
```

Figure 4.13: The new events in CompCert: mem_effect reflects changes to memory and delta_perm_map represents transfer of Cur permissions.

However, doing so would rule out some reasonable programs that we would like to verify (such as the one in fig. 4.12(b)), and it would add more complexity for the user who has to prove source programs correct.

We propose to include 2 new types of events which we call *memory events*, as described in fig. 4.13. As their name suggests, they expose the state of memory and they expose pointer values. The first one, Event_acq_rel, represents a generic memory interaction where the external function performs some arbitrary changes to memory, recorded by a list of mem_effects, and transfers some permissions recorded by delta_perm_map. The second one, Event_spawn, represents the creation of a new thread or a new module. It records the function being called, as a block number, and the change in permissions by two delta_perm_maps: one representing the permissions given and the other the starting permissions of a new thread/module. Notice that, in the mem_effects, Write contains a list of memvals, which can be uninitialized (i.e., Undef). In the example fig. 4.12(d), the Event_acq_rel would have the mem_effect, Write(&a,Undef).

The correctness of the CompCert compiler is formulated as a preservation of traces. However, our new traces expose the state of memory, which is not identically preserved. Fortunately, we know that compilation preserves an injection of the memories, so we can state the CompCert correctness as a preservation of traces, up to some rearrangement by an injection. We define the injection relation for mem_effects in fig. 4.14(d), from where the injection relation for traces can be derived. The full simulation, with the injection-related traces, will be later explained in section 4.5.

The injection relation for values, defined by CompCert, is not deterministic (i.e., it's does not represent a function). As shown in fig. 4.6(a), the value Vundef can be injected to any other value. Consequently, the injection relations for memories, mem_effects, events and traces are not deterministic. We do, however, include in all of those definitions a stronger notion of injection relations named *strong injection*, denoted by $\hookrightarrow\!\!\!\!\!\rightarrow$, which is determinisitic.

We discuss simulations for traces with non deterministic relations in section 4.5.1.

4.4 MOIST Semantics

As described in the introduction, we need more expressive semantics to distinguish the current memory, during program execution, and the points where external functions are called. We call this expanded semantics *MOIST semantics* (just like our simulations) and it extends CompCert semantics with the following:

- Memory: get_mem: The state of every language in CompCert can be interpreted as a pair of a *core* and a memory [6] and get_mem is the projection that returns the memory inside the state.

$$\frac{j\ b_1 = \mathrm{Some}\ (b_2, \delta) \qquad \mathrm{lo}_2 = \mathrm{lo}_1 + \delta \qquad \mathrm{hi}_2 = \mathrm{hi}_1 + \delta}{(b_1, \mathrm{lo}_1, \mathrm{hi}_1) \overset{j}{\hookrightarrow} (b_2, \mathrm{lo}_2, \mathrm{hi}_2)}$$

(a) Injection of ranges.

$$\frac{}{\mathrm{nil} \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{nil}} \qquad \frac{\mathrm{ls}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{ls}_2 \qquad x_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} x_2}{x_1 :: \mathrm{ls}_1 \overset{j}{\hookrightarrow} x_2 :: \mathrm{ls}_2}$$

$$\frac{}{\mathrm{nil} \overset{j}{\hookrightarrow} \mathrm{nil}} \qquad \frac{\mathrm{ls}_1 \overset{j}{\hookrightarrow} \mathrm{ls}_2 \qquad x_1 \overset{j}{\hookrightarrow} x_2}{x_1 :: \mathrm{ls}_1 \overset{j}{\hookrightarrow} x_2 :: \mathrm{ls}_2}$$

(b) Injection of lists (for any type that has an injection relation)

$$\frac{(b_1, \mathrm{lo}_1, \mathrm{hi}_1) \overset{j}{\hookrightarrow} (b_2, \mathrm{lo}_2, \mathrm{hi}_2)}{\mathrm{Alloc}\ b_1\ \mathrm{lo}_1\ \mathrm{hi}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{Alloc}\ b_2\ \mathrm{lo}_2\ \mathrm{hi}_2} \qquad \frac{j\ b_1 = \mathrm{Some}\ (b_2, \delta) \qquad \mathrm{locs}_1 \overset{j}{\hookrightarrow} \mathrm{locs}_2}{\mathrm{Free}\ b_1\ \mathrm{locs}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{Free}\ b_2\ \mathrm{locs}_2}$$

$$\frac{j\ b_1 = \mathrm{Some}\ (b_2, \delta) \qquad \mathrm{vals}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{vals}_2}{\mathrm{Write}\ b_1\ \mathrm{vals}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{Write}\ b_2\ \mathrm{vals}_2}$$

(c) Strong injection of mem_effects

$$\frac{\mathrm{me}_1 \overset{j}{\lhook\joinrel\twoheadrightarrow} \mathrm{me}_2}{\mathrm{me}_1 \overset{j}{\hookrightarrow} \mathrm{me}_2} \qquad \frac{j\ b_1 = \mathrm{Some}\ (b_2, \delta) \qquad \mathrm{vals}_1 \overset{j}{\hookrightarrow} \mathrm{vals}_2}{\mathrm{Write}\ b_1\ \mathrm{vals}_1 \overset{j}{\hookrightarrow} \mathrm{Write}\ b_2\ \mathrm{vals}_2}$$

(d) Regular injection of mem_effects

Figure 4.14: More injection relations, including strong and regular injections for mem_effects. Strong injections () can only map Vundef values to themselves, while injections () can map.

```
Definition at_external (c: state) : option (external_function * list val) :=
  match c with
  | Callstate fd args k _⇒
      match fd with
      | External ef targs tres cc ⇒ if ef_inline ef then None else Some (ef, args)
      | _⇒ None
      end
  | _⇒ None
 end.
```

Figure 4.15: at_external definition for Clight. The function checks that (1) the current state is about to make a function call, (2) the function is an External function and (3) the external function cannot be inlined. The compiler is allowed to inline specific functions such as memcpy and certain builtins.

- Observable: at_external : This function exposes when a program is about to call an external function and it returns the function and the arguments being passed. The instantiation for Clight is shown in fig. 4.15.
- Injectable: It uses the injectable traces described in section 4.3.
- Startable: entry_point: This is a generalization of initial_state, as described in section 4.2.

Our semantics is very closely related to *interaction semantics* [33] with two main differences. First, we don't need to define after_external for every language. We only define it for Clight and Assembly, the languages that show up in the specification. In those languages, we prove that we can replace the one-step CompCert external function calls with the small step execution of that function, using at_external and after_external. There is therfore no need for the compiler or the intermediate languages to know about after_external. Second, we allow states that are at_external to take a step in the CompCert semantics; namely, the execution will continue by calling the external function. The CompCert semantics, in this case, represents the *thread-local* view (or *module-local*), where external functions, other threads, and modules are all abstracted into oracles that execute in one step.[6]

In fact, given a Startable factorable-state semantics we can derive an interaction semantics, if only we define after_external. The step relation is constructed by removing steps from states that make external function calls, as described by at_external. We use this feature to define interaction semantics for Clight and Assembly, as shown in chapter 5.

4.5 Definitions for MOIST simulations

CompCert's compiler specification is stated as the following semantic preservation theorem:

Theorem 4.5.1 (CompCert semantic preservation)**.** *Let S be a source program and C be its compiled version. For all behaviors B that do not go wrong, if S has behavior B, then C too has behavior B. In short:*

$$\forall B \notin \mathit{Wrong}.\ S \Downarrow B \Rightarrow\ C \Downarrow B \tag{4.1}$$

Here a behavior is a a trace and a termination or divergence. If a specification *spec* is a function of behavior, then it also holds that CompCert preserves specifications in the sense that:

$$S \models \mathit{spec} \Rightarrow C \models \mathit{spec} \tag{4.2}$$

Such specification fails to preserve richer notions of behavior that can be described in higher-order separation logic and can be proven for Clight programs by tools like VST [8]. Moreover, the high level specification in eq. (4.1) is not well suited for modular reasoning to support shared memory concurrency or compositional compilation [33].

[6]In CompCert the oracle, called external_functions_sem, is passed as a parameter to the correctness proof and gives the semantic of external functions.

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\sim"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t"', "*"] \\
s_1' \arrow[r, phantom, "\sim"] & s_2'
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"', "*"] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}
\qquad
\begin{array}{c}
t \xhookrightarrow{j'} t' \\
j \sqsubseteq j'
\end{array}$$

(b)

Figure 4.16: Step simulation diagrams. (a) if s_1 takes a step to s_2 with trace t and s_1 is related to some s_2, then s_2 can take a number of steps with trace t' to a new state s_2' related to s_1'. (b) The new diagram exposes the memory injections j and j' and the traces t and t', are equivalent up to injection (inject_trace_strong j' t t').

The simulations that CompCert uses to prove eq. (4.1) are better suited for these purposes. CompCert proves a forward simulation between its source and target executions which, together with the determinism of the target language, imply eq. (4.1). These simulations, encoded in the record fsim_properties, state that (1) public global variables and functions are preserved, (2) initial_states are preserved (fig. 4.7), (3) final_states are preserved and (4) execution is preserved (fig. 4.16(a)). The simulations are parametric on a *match relation* (noted as $\sim$), an invariant of related states in source and target which the relation is established at initial states and preserved by the step simulation.

For all CompCert phases, the *match state* relation describes how the memory changes after compilation. In some passes, memory does not change at all (e.g. Cshmgen or Linearize) and sometimes the memory is extended by increasing the size of existing memory blocks with new values (e.g. Allocation, Tunneling). In other cases, memory is reordered, memory blocks are coalesced, and some are unmapped. CompCert expresses this reordering with *memory injections* that map memory blocks to their new block with some offset. For example, in Cminorgen, the compiler coalesces all stack-allocated local variables of a function into a single stack block. We use this same injection to describe how traces with memory events evolve through compilation (fig. 4.16(b)).

We propose a more expressive simulation inject_sim that improves the CompCert simulations. The full definition can be found in appendix A.4.1 and we describe the key points below:

- The simulation exposes how the memory changes: We expose the memory injection j that describes how memory changes after compilation. For simplicity of the proofs, for compiler passes that either preserve the memory or simply extend it, we also define the simpler simulations eq_sim and extend_sim respectively. These simulations follow immediately from the ones already proven in CompCert. All of the simulations we define compose to inject_sim as shown by the composition lemmas 4.5.2, 4.5.3 and 4.5.4.

  Lemma 4.5.2. *For all semantics L_1 and L_2 if eq_sim L_1 L_2 then c L_1 L_2.*

  Lemma 4.5.3. *For all semantics L_1, L_2, L_3, if extend_sim L_1 L_2 and inject_sim L_2 L_3, then inject_sim L_1 L_3.*

  Lemma 4.5.4. *For all semantics L_1, L_2, L_3, if inject_sim L_1 L_2 and inject_sim L_2 L_3, then inject_sim L_1 L_3.*

- The simulation admits traces with memory events and preserves the traces up to memory injection. That is, the memory events in the trace are appropriately renamed according to the permutations described in the memory injection.

- The simulation preserves external function calls. The original CompCert simulation only preserves traces such that, for example, a compiler could replace an external function call with internal code that produces the same event. In fact the compiler does exactly that with some special external calls such as memcpy and certain built-ins. However the compiler does not do that with arbitrary external functions (of course not!), but the simulation specification does not rule it out. We add preserves_atx to the simulation, which says that if a source at_external s_1 = Some (f,args)

$$\exists t'.\ t \overset{j'}{\hookrightarrow} t'$$

$$\exists j'.\ j \sqsubseteq j'$$

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"'] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}$$

Figure 4.17: At external step diagram (simulation_atx). Exclusive for external function calls, this diagram follows the simulation diagram in fig. 4.16, but enforces that the compiled execution takes only one step. The entire code is presented at the end of appendix A.4.1.

$$m_s \overset{j'}{\hookrightarrow} m_t$$

$$args_s \overset{j'}{\hookrightarrow} args_t$$

$$\begin{tikzcd}
{} \arrow[d, "\begin{array}{r}\text{entry\_points}\\ m_s\ f\ \text{args}_s\end{array}"'] & {} \arrow[d, dotted, "\begin{array}{l}\text{entry\_points}\\ m_t\ f\ \text{args}_t\end{array}"] \\
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] & s_2
\end{tikzcd}$$

Figure 4.18: Diagram for initial states with injected initial memories and arguments. Given an entry point in the source, for any target memory m_t and arguments args_t, that are related to the source by an injection, there exists an entry point for the targert with those arguments. The new source state is related to the target one by the $\sim$ relation.

state is at_external, then any target state it matches is also at_external with the same functions and related arguments (i.e., equal up to memory injection). We consider the simulations of built-ins in later sections.

- The simulation preserves the number of steps taken by external functions. This fact was already proven in each CompCert phase, but it was hidden in the less expressive simulation theorems exposed by each phase. We include a new diagram (fig. 4.17, code in appendix A.4.1), simulation_atx, which says that if a source state that is at_external takes exactly one step, then the matching target state does the same (as opposed to any number of steps as in fig. 4.16), and the two resulting states match.

  We further expand the notion of simulation_atx at the end of this subsection.

- The simulation can start executions with functions that take arguments and are not main. We replace the initial_state diagram with the diagram for entry_point as described in fig. 4.7.

Our simulation diagrams for entry_points is strikingly simple compared to that of other authors. In Compositional CompCert [33] and in CASCompCert [18], the initial state simulation accepts arbitrary (but injected) memories which, in our notation, would look like 4.18. That is, the initial state has to exist for any memory m_t and arguments args_t, as long as they are injected. Such diagram makes the proof of compiler passes harder and unnecessarily complicates passes that do not inject memory[7]. For the simple task of compiling whole sequential programs with arguments to main, our diagram is enough. We further show that our simpler diagram is enough to achieve concurrency in chapter 5. We also conjecture that the diagram is enough for separate compilation, which we discuss in section 8.1.

Summary regarding **inject_trace_strong**.

CompCert is carefully designed so that internal steps are deterministic, to simplify compiler-correctness proofs. However, external steps can be nondeterministic, as long as they manifest their nondeterministic choices in their events. We preserve this design decision and extend it.

In particular, when we allow Vundef values in traces, and we allow traces to be injected, we want simultaneously that

1. CompCert correctness proofs can mostly behave as if trace-injection is deterministic, so Vundef injects to Vundef;

[7]Stewart et al. [33] modify all passes to use injections. In other work [6] the authors propose using nine different composition theorems to compose all possible memory relations pairs (i.e., equality, extension and injection).

at_externals_1 = Some (f,args)

$\exists j', j \sqsubseteq j'$ $\qquad \exists t.\ t \overset{j'}{\hookrightarrow\!\!\!\!\twoheadrightarrow} t'$ $\qquad\qquad \forall t.\ t \overset{j'}{\hookrightarrow} t'$

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"'] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"'] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}$$

(a) $\qquad\qquad\qquad$ (b)

Figure 4.19: Two simulation diagrams needed to establish behavioral refinment for non deterministic trace relations. (a) is exacctly the same as fig. 4.17. (b) adds a simulationfor all other posible images of the inje‘elation.

2. Event_acqrel can inject from Vundef to defined values, which permits programs where semaphores control shared access to uninitialized buffers.

lemma 4.5.5, along with the work in chapter 5, permits both off these to be true at once. That is, compiler-correctness proofs are still reasonably simple, but external events are more expressive.

4.5.1 Simulations for traces without deterministic relations.

Dockins [11] proved that forward simulations, with a deterministic target language and a receptive source language, are enough to establish behavioral refinement. But this is only true if the relation between traces is deterministic. In CompCert, the relation between traces is equality, which is deterministic. When the traces' relation is not deterministic, the simulation diagram (as in fig. 4.19(a)) relates the source execution with only one of the possible target executions. As we saw in section 4.3, the injection of traces is, unfortunately, not deterministic. In order to establish a behavioral refinement we also need the diagram in fig. 4.19(b), which states that for all other images of t (by the same injection j'), the semantics can also take a step to a related state. The full code for the *double external call diagram* is shown in fig. 4.20, where the bolded code corresponds to the second diagram fig. 4.19(b).

Keep in mind that we could avoid needing the second diagram in fig. 4.19 if we keep uninitialized variables out of the trace, just like CompCert does. We have defined *strong injections* (the $\hookrightarrow\!\!\!\!\rightarrow$ in fig. 4.14), which are really deterministic, and we provide a simpler simulation diagram for external function calls that uses the strong injection to relate source and target traces. This diagram is enough to handle all external functions that do not reveal uninitialized values in the trace.[8] However, we want (and can!) support functions that expose uninitialized variables. These function calls are not input/output but a thread/module local view of the context (e.g. the execution of other thread) so, once these external calls have been replaced with their small step execution, these events are not external to the program anymore and they disappear from the trace. We can then establish the behavioral refinement just like Dockins did.[9]

The double diagram might seem complex and harder to prove but, remember, trace events only appear at external calls. Therefore, the diagrams are easy to prove as long as they hold for external calls. CompCert already requires that external calls commute with injection; for external functions with memory events, the only way to commute with injections is to satisfy the two diagrams in fig. 4.19.

We must wonder if there are reasonable functions that do not satisfy both diagrams in fig. 4.19. The answer is yes. Imagine a function show_a interacting with the code in fig. 4.12 that acquires lock l and then outputs the content in a. The trace of that function will contain either Vundef or 3, but not any other value. Does this mean that we cannot support show_a and other similar functions? No. We can

[8]The diagram with the strong injection, even supports some external functions with Vundef in their code, as long as that value does not change during compilation. In which case, we say that the function's semantics commutes with strong injections.

[9]We can go even further and support input/output events that revel uninitialized locations in memory. Even in those cases, one can establish refinement using the double diagram simulations and a strengthened version of Dockins' theorem.

```
 1  Definition simulation_atx_stronger {index:Type} {L1 L2: semantics}
 2                    (match_states: index → meminj → state L1 → state L2 → ℙ ) :=
 3              ∀ s1 f args,
 4                at_external L1 s1 = Some (f,args) →
 5                ∀ t s1' i f s2,
 6                Step L1 s1 t s1' →
 7                match_states i f s1 s2 →
 8                    ∃ f', Values.inject_incr f f' ∧
 9                            (∃ i' s2' t',
10                              Step L2 s2 t' s2' ∧
11                              match_states i' f' s1' s2' ∧
12                              inject_trace_strong f' t t') ∧
13                                    (∀ t', inject_trace f' t t' →
14                                     ∃ i', ∃ s2',
15                                       Step L2 s2 t' s2' ∧
16                                       match_states i' f' s1' s2') .
```

Figure 4.20: Stronger simulation for external steps, that universally quantifies over all injected traces. Lines 8-11 describe the existentially quantified diagram as described in fig. 4.19(a). Lines 12-15, in bold, describe the diagams for all other images of thee trace injection relation as described in fig. 4.19(b). inject_trace is the predicate that allows undefined values to be mapped to defined ones.

define a behavior[10], denoted $\overline{\text{show\_a}}$, showing the ouput with value in a, unless it is uninitialized, in which case it just outputs any value. The key observation is that, when the value in a is undefined, the calling program cannot distinguish between show_a and $\overline{\text{show\_a}}$. That is because a safe program cannot compare an undefined value. Similarly, if the compiler is correct for a program "calling" $\overline{\text{show\_a}}$, which has more behaviors, then it is correct for show_a. In summary, we should always prove the compiler correct with respect the those more liberal external functions to support all functions that have undefined values in their trace.

Finally, we define two simulations, inject_sim and inject_sim_strong, with the external step diagrams simulation_atx (fig. 4.21) and simulation_atx_strong (fig. 4.20), respectively. We prove that they compose, according to the lemma below, so we can

[10]A behavior is not a C function, it's just a Coq inductively defined predicate. Unlike C programs, it can test if a variable is undefined.

```
Definition simulation_atx {index:Type} {L1 L2: semantics}
                (match_states: index → meminj → state L1 → state L2 → ℙ ) :=
        ∀ s1 f args,
          at_external L1 s1 = Some (f,args) →
          ∀ t s1' i f s2,
          Step L1 s1 t s1' →
          match_states i f s1 s2 →
              ∃ f', Values.inject_incr f f' ∧
                      (∃ i' s2' t',
                        Step L2 s2 t' s2' ∧
                        match_states i' f' s1' s2' ∧
                        inject_trace_strong f' t t') .
```

Figure 4.21: Coq definition for the external step simulation diagram as described in fig. 4.17

simplify most proofs by using the simpler inject_sim. As long as the last pass of the compiler satisfies inject_sim_strong, we can show that the entire compiler does too.[11]

Lemma 4.5.5. *For all semantics L_1, L_2, L_3, if inject_sim L_1 L_2 and inject_sim_strong L_2 L_3, then inject_sim_strong L_1 L_3.*

4.5.2 Simulation diagrams for built-ins.

Our simulations require that external function calls be preserved 1-to-1 by the compiler (fig. 4.21). That is, if the source is at_external, it takes exactly one step to cross the external call; then the target is also at_external, and also takes exactly one step to cross the external call. But CompCert's compilation of built-ins (between Clight through various ILs to RTL) does not have this property. The *bigstep* semantics of built-ins includes evaluating their arguments, which is language-dependent and takes a different number of steps in different languages. We have two proposals to solve this problem: (1) a satisfying solution that we implemented for the purpose of this

[11]We can go one step further to simplify the passes of all compiler passes. If we prove that the assembly language satisfies a self_simulation (section 5.3), then all compiler proofs can be specified according to the simpler inject_sim, and we still get the stronger property for the entire compiler.

work, and (2) a long-term solution that is more principled, but involves more changes to the compiler and the proof.

Solution 1. Although the execution of built-ins is not preserved 1-to-1, we can still support built-ins that are not "external functions". For example, a system call like memcpy does not communicate with other threads and can be viewed as an "internal function" for the purpose of concurrency. For this to work, we must separate real "external functions" from those that are not. We assume that all built-ins are "inlinable" (ef_inline) in the sense that they can really be inlined in the code. All other external functions should be called using the external function protocol. This is what CompCert's parser does. On the other hand, built-ins can be called as external functions and the compiler inlines them as built-ins. We do not consider these inlinable built-ins as "external functions" (i.e., at_external returns None) and we enforce that the semantics of calling built-ins is "stuck" if it tries to execute non-inlinable external functions. For our concurrency application, we also require that all of these inlined built-ins respect the memory interface (respects permissions, etc.) but other applications can relax this condition.

This solution permits truly internal built-ins such as memcpy that have no external event trace, and it permits built-ins that have event traces, such as volatile-variable store to a device register that causes external output. But it does not permit built-ins that have a synchronizing effect such as lock-acquire and lock-release. In solution 1, the C program must make a function-call to an assembly-language implementation of acquire or release (or spawn). That is, our at_external predicate really means "at a synchronization/spawn/etc. call," whereas I/O can be done through CompCert's existing "Event-trace" mechanism. We would eventually want to allow synchronization operations to be implemented as inlineable built-ins. For example, in the context of concurrency, a built-in atomic store can be used as a semaphore. This will become even more important if a future extension of our concurrency research permits more of the C11 atomic load/store operations. Our solution 1 does not permit inlineable synchronizers. To support those we propose:

Solution 2. We know that in Clight, the evaluations of external calls and built-ins are almost identical, except external functions executed in three steps (one to evaluate arguments, one to execute the function and one to return), while built-ins do it in one step. In fact, in the Selection pass, all *inlinable* external functions are translated to built-ins, and it is proven that such transformation preserves the program's semantics. This transformation reduces the number of steps taken by the program; however, the following transformation, RTLgen, increases the number of steps used to execute a built-in to evaluate its arguments. This second trasformation violates the 1-to-1 requirement of simulation_atx.

We propose to delay the inlining of built-ins until after function arguments have been evaluated. This can be easily done with a simple *RTL to RTL* compiler pass that only inlines external functions and that since the arguments have already been evaluated, it satisfies the diagram simulation_atx. In this proposal, the languages above RTL (C, Clight, Cminor, etc.) would not even have an Sbuiltin command; built-ins (even inline assembly) would be expressed as if they were function calls, then rewritten as Ibuiltin in the RTL. The semantics and the generated code would each end up identical; and the CompCert compiler and proof would end up smaller than it is now.

Technical note: Our proposed pass does indeed transform three steps in the source program into one in the target (just like the Selection pass). However, this is proven in three different diagrams: two diagrams that take one step in the source and none in the target, and one diagram that takes one step in the source and target, satisfying simulation_atx.

4.5.3 Full injections

Most passes in CompCert preserve the contents in memory. Even injection passes, such as Cminorgen, Stacking and Inlining, only reorder memory and coalesce blocks but do not remove any content from memory. Only two passes currently remove contents out of memory: SimplLocals, which pulls scalar variables whose address is not taken into temporary variables, and Unusedglob, which removes unused static globals. For those injection passes where memory content is preserved, we make it explicit by adding a predicate full_injection that states that an injection maps all valid blocks in memory. In the remaining of this subsection, we explain the current limitations of the way CompCert specifies unmapped parts of memory. In our version of CompCert, a compiler that skips SimplLocals and Unusedglob can expose full_injection and overcome those limitations. Certainly, requiring all memory to be mapped is also a strong limitation. In what remains of this chapter, we will make the problem clear and propose a solution, acknowledging its implementation is beyond the scope of this thesis. We further discuss solutions for this limitation in related work in chapter 2 and in our future work section 8.1.

Consider the remember() and incr() functions from fig. 4.1. As we discussed before, the execution of incr depends on the location in memory $*$buff. We already mentioned that if external functions behave this way, they cannot satisfy the strict "correctness" requirements of CompCert, so we have corrected this problem with memory events. The second problem with this simple function, however, is that it relies on the fact that the compiler does not remove buff from memory. CompCert does, in fact, preserve that piece of memory since its address has escaped. This fact however is not part of the compiler's specification.

As a second example, consider shared memory concurrency. When two threads are interacting through memory, each thread needs to know that the memory to which it gains acces is not unmapped. A thread can only use the locations over which it has permissions: a superset of the locations accessed by the thread. This approach allows us to ensure that the memory does not change when other threads execute. Unfortunately, if part of the memory is unmapped (by a compiler phase, because the program contains no accesses to it), we cannot ensure that the threads execute correctly. This problem is surprisingly close to the inr() example, and many of the solutions for that problem will also apply here.

In his original paper about CompCert, Leroy [23] claims that “inputs given to the programs are uniquely determined by their previous outputs”. That seems to suggest that functions like remember/incr would be safe but, in its implementation, CompCert rather requires that “inputs given to the programs are uniquely determined by their last outputs” (i.e., the arguments to the external function call). However, we could implement the former (stronger) specification by allowing external functions to depend on the entire args_hist. Moreover, one should be able to prove that args_hist is not unmapped by SimplLocals or Unusedglob, since it only contains escaping pointers. These changes are beyond the scope of this work, so we temporarily use full_injection and we skip the two problematic passes. We discuss this solution further in section 8.1.

Chapter 5

Compiler Correctness

In this chapter we present the compiler specification that supports concurrent code, and we show that CompCert satisfies this specification. Just as we did for MOIST semantics, the specification of correctness for CPMs is given as a simulation relation. As we explained in section 3.1.1, this simulation is enough to derive safety and correctness preservation. We start the proof by introducing our novel technique to support threads as if we could *Compile One At a Time*, and then we show how to prove the simulation for the different steps in the CPM.

5.1 Compiler Specification

A compiler that correctly compiles a concurrent program must preserve a simulation between the source and compiled programs, as CPM semantics. The simulation describes how executions relate on both programs: it very explicitly shows that permissions transfers are preserved (up to injection), and that nonstuck programs compile to nonstuck programs. These CPM simulations are similar to MOIST simulations except that instead of differentiating external function calls, they differentiate steps executed by the machine (i.e., synchronization operations).

We state that the CompCert compiler satisfies the specification of correctness given here, and then we prove that fact in later sections of this chapter.

Definition 5.1.1 (CPM simulation). *Two concurrent permission machines, CPM_1 and CPM_2 are in a simulation relation $CPM_1 \gtrsim CPM_2$ if there is a* match relation *for states* $\gtrsim$[1] *and a well ordered measure such that*

1. *at each point of the execution, CPM_1 and CPM_2 have the same number of threads,*
2. *at each point of the execution, CPM_1 and CPM_2 have the same running threads (i.e., local states marked with Run),*
3. *If CPM_1 halts (the schedule runs out), then so does CPM_2.*
4. *The initial state in CPM_1 has a matching initial state in CPM_2, related by $\gtrsim$.*
5. *Each core step in CPM_1 is simulated by some number of core steps in CPM_2, preserving the match relation $\gtrsim$ (as depicted in fig. 5.1(a)). If CPM_2 takes 0 steps, the measure decreases according to a well founded order.*
6. *Each administrative step (resume, suspend, stutter, done) in CPM_1 is simulated by an equivalent administrative step in CPM_2, preserving the match relation $\gtrsim$ (as depicted later in this chapter in fig. 5.3).*
7. *Each synchronization step (release, acquire, fail_acquire, spawn, make_lock, free_lock) in CPM_1 is simulated by an equivalent synchronization step in CPM_2, preserving the match relation $\gtrsim$ (as depicted in fig. 5.1(b)). The produced events are equal, up to injection.*

[1] We use the notation $\gtrsim$ for match relations of CPM states and $\sim$ for match relations of states in MOIST semantics.

$\exists\ \Sigma_2'\ j', j \sqsubseteq j'$

$$\begin{tikzcd}
\Sigma_1 \arrow[d, "\text{core}"'] \arrow[r, phantom, "\gtrsim_j"] & \Sigma_2 \arrow[d, dotted, "\text{core}", "*"'] \\
\Sigma_1' \arrow[r, phantom, "\gtrsim_{j'}"] & \Sigma_2'
\end{tikzcd}$$

(a) Core steps

$\exists\ \Sigma_2'\ j'\delta_2, j \sqsubseteq j'$

$$\begin{tikzcd}
\Sigma_1 \arrow[d, "\text{sync } \delta_1"'] \arrow[r, phantom, "\gtrsim_j"] & \Sigma_2 \arrow[d, dotted, "\text{sync } \delta_2"] \\
\Sigma_1' \arrow[r, phantom, "\gtrsim_{j'}"] & \Sigma_2'
\end{tikzcd}$$

$$\delta_1 \xhookrightarrow{j'} \delta_2$$

(b) Synchronization steps

Figure 5.1: CPM internal and synchronization step diagrams. The new core transition can be any number of steps but the synchronization steps happen in lock-step.

The last three items in the definition of CPM simulation are the most difficult to prove, and we will spend significant portion of this chapter proving similar diagrams, such as the one depicted in fig. 5.3. For that, we spell out the details of a CPM step simulation diagram in the following definition.

Definition 5.1.2 (CPM step diagram)**.** *Let $\gtrsim_j$ be a given match relation for CPM states and let P describe some kind of CPM step such as core, administrative or synchronization steps. Also, let there be two CPM states in a match relation, $\Sigma_1 \gtrsim_j \Sigma_2$. If Σ_1 takes a P step to some Σ_1', then Σ_2 takes some steps in P to some Σ_2' such that the match relation can be reestablished for a larger injection $\Sigma_1' \gtrsim_{j'} \Sigma_2'$.If the steps produce events, these events are injected by j'. If Σ_2 takes 0 steps, the measure decreases according to its well founded order.*

We say the diagram is in lockstep *if Σ_2 is required to take exactly one step.*

We will define a match relation $\gtrsim$ for CPMs in definition 5.2.4 (with definition 5.4.1), and we will use that relation for all the diagrams in this chapter.

Following the previous definitions, the last three items in a of CPM simulation can be reworded as:

5'. The core steps of CPM_1 and CPM_2 form a simulation diagram.

6'. The administrative steps of CPM_1 and CPM_2 form a lockstep simulation diagram.

7'. The synchronization steps of CPM_1 and CPM_2 form a lockstep simulation diagram.

It is often important to prove that simulation relations like the one in definition 5.1.1 are transitive in order to achieve modularity. For example, the CompCert correctness proof uses transitivity of the simulation relation to verify every compiler pass separately and then compose them into a full compiler simluation. Indeed, we can establish transitivity of definition 5.1.1, but we exploit it in a novel way to get a new type of modularity (Compile One At a Time, section 5.2) in our compiler correctness proof.

Lemma 5.1.3. *For all machines CPM_1, CPM_2 and CPM_3, if $CPM_1 \gtrsim CPM_2$ and $CPM_2 \gtrsim CPM_3$ then $CPM_1 \gtrsim CPM_3$.*

Armed with the definition of simulation for CPMs, we can state the specification of the CompCert compiler. The rest of this chapter describes the proof of how CompCert satisfies this specification.

Theorem 5.1.4 (Compiler Correctness)**.** *Given a source program P and an x86 assembly language program Q obtained by CompCert compilation of P, the CPMs of the two programs are in a simulation relation:*

$$CPM_{Clight}(P) \gtrsim CPM_{Asm}(Q)$$

5.2 Compiling One At a Time (COAT)

Beringer et al. [6] explained how CompCert does not guarantee the properties it requires from external functions for its modules. For example, CompCert requires that external functions do not modify locations in memory that are not allocated in the target program, while a CompCert function can modify a local variable that is never used and is removed by the compiler. This problem appears with recursive modules in separate compilation, and with threads (modeled as external functions) in concurrent programs. To solve this asymmetry, Beringer et al. developed Logical Simulation Relations. Stewart et al. [33] later improved them with structured simulations. Kang et al. [21] avoid the problem altogether by using the same compiler for all modules, but they therefore cannot support different compilers or concurrency.

We propose a completely different approach. We observed that the problem disappears if external functions are not compiling while we compile internal functions. But how can we support mutually recursive modules and concurrency (modeled as external functions) if the external functions cannot be compiled? We mathematically model a program where only one thread has been compiled and prove a simulation between the source program and the "partially compiled" program. We then repeat the process with one thread at a time proving a simulation at each step. Finally, we compose all the simulations to obtain a simulation between the source program and the compiled one.

To model an execution where only some threads have been compiled, we construct a *Hybrid Machine*. This is a CPM such that some threads execute assembly code from the compiled program, and some threads execute C code from the source program. Our use of interaction semantics in the CPM ensures that the different semantics compose properly in the Hybrid Machine.

$$\frac{\begin{array}{c}\text{hb} \le i \qquad h_i = \text{Start}(f,a) \\ \vec{h}' = \vec{h}[i \mapsto \text{Run}(\text{initialCore}_{\text{Clight}}(f,a))]\end{array}}{\Psi, \Phi \vdash_{\text{HM}_{\text{hb}}} \langle i \cdot \mho, (\vec{h}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{h}', \vec{\pi}, L), m\rangle} \quad (\textsc{start src})$$

$$\frac{\begin{array}{c}i < \text{hb} \qquad h_i = \text{Start}(f,a) \\ \vec{h}' = \vec{h}[i \mapsto \text{Run}(\text{initialCore}_{\text{Asm}}(f,a))]\end{array}}{\Psi, \Phi \vdash_{\text{HM}_{\text{hb}}} \langle i \cdot \mho, (\vec{h}, \vec{\pi}, L), m\rangle \mapsto \langle i \cdot \mho, (\vec{h}', \vec{\pi}, L), m\rangle} \quad (\textsc{start tgt})$$

$$\frac{\begin{array}{c}\text{hb} \le i \qquad h_i = \text{Run}(\sigma) \qquad \Psi \vdash_{\text{Clight}} \langle \sigma, m|_{\pi_i^1}\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m'\rangle \\ \vec{h}' = \vec{h}[i \mapsto \text{Run}(\sigma')] \qquad \vec{\pi}' = \vec{\pi}[i \mapsto (\text{Cur}(m'), \pi_i^2)]\end{array}}{\Psi, \Phi \vdash_{\text{HM}_{\text{hb}}} \langle i \cdot \mho, (\vec{h}, \vec{\pi}, L), m\rangle \overset{\epsilon_i}{\mapsto} \langle i \cdot \mho, (\vec{h}', \vec{\pi}', L), m'\rangle} \quad (\textsc{core src})$$

$$\frac{\begin{array}{c}i < \text{hb} \qquad h_i = \text{Run}(\sigma) \qquad \Phi \vdash_{\text{Asm}} \langle \sigma, m|_{\pi_i^1}\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m'\rangle \\ \vec{h}' = \vec{h}[i \mapsto \text{Run}(\sigma')] \qquad \vec{\pi}' = \vec{\pi}[i \mapsto (\text{Cur}(m'), \pi_i^2)]\end{array}}{\Psi, \Phi \vdash_{\text{HM}_{\text{hb}}} \langle i \cdot \mho, (\vec{h}, \vec{\pi}, L), m\rangle \overset{\epsilon_i}{\mapsto} \langle i \cdot \mho, (\vec{h}', \vec{\pi}', L), m'\rangle} \quad (\textsc{core tgt})$$

Figure 5.2: Hybrid Machine semantics: internal steps and start thread.

5.2.1 The Hybrid Machine

The Hybrid machine is a two-language Concurrent Permission Machine, indexed by a *hybrid bound*, $\text{hb} \in \mathbb{N} + \infty$. It operates over states of the form $\langle \mho, (\vec{h}, \vec{\pi}, L), m\rangle$, where $\mho, \vec{\pi}, L$ and m are the schedule, the list of permissions, the lock permissions and the global memory just like in the CPM. The hybrid list of states $\vec{h}$ contains marked local states, such that the first hb states are in Assembly and the rest in Clight.

The semantics of the Hybrid Machine employs judgments similar to that of the CPM, $\Psi, \Phi \vdash_{\text{HM}_{\text{hb}}} \langle \mho, (\vec{h}, \vec{\pi}, L), m\rangle \overset{\epsilon}{\mapsto} \langle \mho', (\vec{h}', \vec{\pi}', L'), m'\rangle$, with some rules shown in fig. 5.2. Ψ and Φ are the source and compiled programs, respectively. When a new thread is spawned, if its index is above the hybrid bound hb, it will run the code in the source program Ψ and the compiled code otherwise. The internal steps will execute according to the index of the thread. Above the hb, it will use the Clight semantics; bellow it, Assembly.

The hybrid machine presented here only contains two programs in two languages and is indexed by a hybrid bound. This is the form we use in the proofs presented in this work. However, the hybrid machine can be instantiated with an arbitrary number of programs (or modules), each in a different language, and threads can be spawned from any of the modules and execute in the corresponding language. For example, in section 8.1.1, we propose a hybrid machine that keeps the first thread in Clight to reason about transformations of the global environment. In section 5.2.3, we propose a hybrid machine with arbitrary languages to tackle separate compilation.

Notice that the the hybrid machine with hb $= 0$ will never consult the compiled program, and will always execute in the source language, just like CPM in Clight. Similarly, the hybrid machine with hb $= \infty$ behaves just like the CPM in Assembly. We can almost say the "CPM $(P) =$ HM$_0$ (P, Q)" and "CPM $(Q) =$ HM$_\infty$ (P, Q)", except that the types do not match, but we can formalize it as follows:

Remark 5.2.1. *For all programs Ψ and Φ, there is an isomorphism between CPM (Ψ) and HM$_0$ (Ψ, Φ) and another isomorphism between CPM (Φ) and HM$_0$ (Ψ, Φ). The isomorphisms map states and steps to identical states and steps, but change the types appropriately. If an operating system has a bound n to the number of threads that a program can spawn, then there is also an isomorphism between CPM (Φ) and HM$_n$ (Ψ, Φ).*

Moreover, the simulation relation in definition 5.1.1 extends naturally to hybrid machines.

5.2.2 COAT proof of compiler correctness

Following remark 5.2.1, the proof of our compiler correctness theorem 5.1.4 can be reduced to the following theorem

Theorem 5.2.2. *Given a source program P and an x86 assembly language program Q obtained by CompCert compilation of P, the source and target hybrid machines are in a simulation relation:*

$$HM_0(P, Q) \gtrsim HM_\infty(P, Q).$$

The proof proceeds in three steps, which we formalize in lemmas below. First we show that "adjacent" hybrid machines are in a simulation relation. Second, we compose the simulations shown in the previous step to prove that the source machine (HM_0) simulates any of the finite machines (HM_n). Third, we show that simulating all "finite" hybrid machines is enough to simulate the target machine (HM_∞). For the rest of this chapter, we assume that P and Q are the source and compiled programs as defined in theorem 5.2.2.

Lemma 5.2.3 (Compile one thread)**.** *For all $n \in \mathbb{N}$, $HM_n(P, Q) \gtrsim HM_{n+1}(P, Q)$.*

As the first step to proving the simulation in lemma 5.2.3, we must define a match relation $\gtrsim_j$ for CPM states. We do this below, with the Coq implementation details in appendix A.5.

Definition 5.2.4[2]**.** *We say two CPM states are in a match relation (indexed by a memory injection j), $\langle \mho_1, (\vec{s}_1, \vec{\pi}_1, L_1), m_1 \rangle \gtrsim_j \langle \mho_2, (\vec{s}_2, \vec{\pi}_2, L_2), m_2 \rangle$, when they satisfy the properties below:*

(a) The schedules are equal $\mho_1 = \mho_2$.

(b) Both states have the same number of threads ($|\vec{s}_1| = |\vec{s}_2|$).

(c) Both states are coherent (definition 3.2.3).

(d) For every location mapped by the memory injection j, the permissions of each thread for this location, is preserved.

[2]The coq implementation is presented in appendix A.5.1

(e) *The memories restricted to the permissions of a thread are injected,*

$$\forall i, m_1|_{\pi^1_{1,i}} \overset{j}{\hookrightarrow} m_2|_{\pi^2_{2,i}} \quad \textit{and} \quad \forall i, m_1|_{\pi^2_{1,i}} \overset{j}{\hookrightarrow} m_2|_{\pi^2_{2,i}}$$

(f) *Locks are preserved by the injection with the same state (locked or unlocked) and their ressources are injected:*

$$\begin{aligned} \forall l_1\ z\ \delta_1,\ L_1\ (l_1, z) = \textit{Some}\ \delta_1 \quad &\rightarrow \\ \exists l_2\ d\ ,\ j\ l_1 = \textit{Some}\ (l_2, d) \quad &\wedge \\ L_2\ (l_2, z + d) = \textit{Some}\ \delta_2 \quad &\wedge \\ \delta_1 \overset{j}{\hookrightarrow} \delta_2 \end{aligned}$$

(g) *Each thread $i < n$ is in Assembly for both CPMs and the thread states are related by an injection relation. The injection of Assembly states is given by the self simulation of the assembly languge in lemma 5.3.2.*

(h) *Each thread $i > n$ is in Clight for both CPMs and the thread states are related by an injection relation. The injection of Clight states is given by the self simulation of the Clight language in lemma 5.3.2.*

(i) *The nth thread in $\vec{s}_1$ is in Clight and in $\vec{s}_2$ is in Assembly. A Clight and an Assembly state are in a match relation, as defined by the MOIST specification of CompCert in theorem 4.0.2. We lift this match to marked states in definition 5.4.1.*

Proof of lemma 5.2.3. This proof relies on several lemmas that will be detailed through the rest of this chapter (sections 5.3 and 5.5). We use the match relation defined in definition 5.2.4 and we consider each condition necessary to establish a simulation in order (definition 5.1.1):

1, 2, 3. The first three properties follow directly from the match relation.

4. The initial state of a machine is a single thread with state Start(main,args) and a global memory with the global environment allocated. As long as the target program has a a main function (corresponding to the same identifier) and a global environment that is injected from the source one, then the simulation of initial states follows. Both those facts follow from the MOIST simulations in theorem 4.0.2.

5. There are two kinds of internal steps: the steps from the thread that is compiling (thread id $= n$) and the steps from other threads (thread id $\neq n$). The diagram for the compiling thread follows from the the internal diagram of the MOIST simulation in theorem 4.0.2. The step diagram of the other threads is proven with lemma 5.3.2.

6. The simulation diagram for synchronization steps is exactly lemma 5.4.2.

7. The simulation diagram for synchronization steps is exactly lemma 5.5.1. □

Lemma 5.2.5 (Compile many threads). *For all $n \in \mathbb{N}$, $HM_0(P, Q) \gtrsim HM_n(P, Q)$.*

Proof. By induction on n, using the transitive composition from lemma 5.1.3 and the stepwise simulation in lemma 5.2.3. □

For systems with hard bound on the number of threads, lemma 5.2.5 is enough to derive a simulation between source and compiled CPMs. Most real systems have a hard bounds on the number of threads that can be initialized, For example, the number of threads created with pthread_create, in Linux, is bounded by the kernel variable pid_max [1], which for linux in 32-bit platforms is bounded above by 2^{15} (for 64-bit architectures the limit is 2^{22}) . However, we want to prove a simulation with the full generality of theorem 5.2.2 (i.e., supporting infinitely many threads).

Proof of theorem 5.2.2. The key observation for this proof is that during execution any process has only a finite number of threads. In fact, a process can only spawn one thread per small step so, after n steps, the processes has at most n threads.

Moreover, with an execution of n steps it is impossible to distinguish between the hybrid machines with hybrid bounds above n, $\{ \mathrm{HM}_i \mid i \in \mathbb{N} + \infty \ , \ i > n \}$. Consequently, we get that every step of an execution HM_∞ is undistinguishable from infinitely many other machines (namely $\{ \mathrm{HM}_i \mid i > n \}$) that are in simulation relation with HM_0.

Let $\gtrsim_i$ be the match relation for states, given by the simulation $\mathrm{HM}_0(\mathrm{P}, \mathrm{Q}) \gtrsim \mathrm{HM}_i(\mathrm{P}, \mathrm{Q})$ (proven in lemma 5.2.5). Also let $|\Sigma|$ be the number of threads in state Σ. We define a match relation $\gtrsim_\infty$, between states Σ_0 and Σ_∞ of HM_0 and HM_∞, respectively, as follows:

$$\Sigma_0 \gtrsim_\infty \Sigma_\infty \quad \triangleq \quad \forall i > |\Sigma_\infty|.\ \ \Sigma_0 \gtrsim_i \Sigma_\infty{}^{3}$$

In other words, Σ_∞ can be seen as state of infinitely many finite hybrid machines, each of which is in simulation relation with $\mathrm{HM}_0(\mathrm{P}, \mathrm{Q})$. Then, Σ_∞ matches Σ_0, if they match for all those simulations[4].

Premises 1, 2, 3 and 4 from definition 5.1.1 follow trivially from the definition of $\gtrsim_\infty$. The simulation diagrams (premises 5,6 and 7), follow from the simulation diagrams of any the simulations $\mathrm{HM}_0(\mathrm{P}, \mathrm{Q}) \gtrsim \mathrm{HM}_i(\mathrm{P}, \mathrm{Q})$, for $i > |\Sigma_\infty|$.

Notice that if the diagram includes spawning a new thread, one of the machines we are considering will take a step that does not match the rest (i.e., $\mathrm{HM}_{|\Sigma_\infty|}(\mathrm{P}, \mathrm{Q})$ spawns a thread in Clight). However, because the number of threads has grown, by definition, we do not have to track the machine $\mathrm{HM}_{|\Sigma_\infty|}$ anymore to establish $\gtrsim_\infty$. □

[3]The expression $\Sigma_0 \gtrsim_i \Sigma_\infty$ is not well typed; the state Σ_∞ is part of the HM_∞, while the right hand side of $\gtrsim_i$ expects a state of HM_i. However, because Σ_∞ has less than i threads, there is a state of the right type that is identical to Σ_∞.

[4]It turns out that, for all $i > |\Sigma_\infty|$, all the relations $\gtrsim_i$ are equivalent but with different types, so it suffices to just check the smallest one, $\Sigma_0 \gtrsim_{|\Sigma_\infty|} \Sigma_\infty$.

Finally we can establish our proof of Compiler Correctness.

Proof of Compiler Correctness . This is a direct corollary of theorem 5.2.2. □

5.2.3 Separate compilation

We propose that the COAT technique can be exploited in separate compilation to simplify the compiler proofs. In this section we present the structure of the proof, but we leave the implementation detail as future work.

We will consider the linking semantics, from [33], and define a *Hybrid Linker* by analogy to our Hybrid Machine which is also indexed by a *hybrid bound*, $\mathrm{hb} \in \mathbb{N} + \infty$. Given a list of modules $S_0,\ S_1, \ldots S_{n-1}$ and their compiled versions $T_0,\ T_1, \ldots T_{n-1}$, a hybrid linker is given by $\mathcal{HL}\ ((\llbracket S_0 \rrbracket, \llbracket T_0 \rrbracket), \ldots (\llbracket S_{n-1} \rrbracket, \llbracket T_{n-1} \rrbracket)$. Just like the linking semantics, the hybrid linker models the execution of the ("selectively compiled") linked program by maintaining as its own state a stack of the modules' core states (i.e., individual instantiations). It also has the following new properties

- It keeps a counter ec, for the number of external calls. Every time an external function is called (and a new runtime invocation of a module is created), the counter increases by 1. The counter never decreases.
- When an external function to module i is executedt if $\mathrm{ec} \geq \mathrm{hb}$, then an instance of the funtion in S_i is created; otherwise it uses T_i.

We use $\gtrsim$ to denote contextual refinement, as extended to hybrid linkers.

Following the same structure as in section 5.2.2, we present the three key results to obtain separate compilation using COAT. In all the following lemmas, we assume the hybrid linkers are all instantiated with the same modules, as described above, and we omit them.

Conjecture 5.2.6 (Compile one core)**.** *For all $n \in \mathbb{N}$, $\mathcal{HL}_n \gtrsim \mathcal{HL}_{n+1}$.*

The proof of conjecture 5.2.6, is beyond the scope of this work, but it should not be harder than the proof of Compiler refinement in [33]. We expect the proof to be easier and allow significant simplification of structured simulations.

Lemma 5.2.7 (Compile many cores)**.** *For all $n \in \mathbb{N}$, $n \in \mathbb{N}$, $\mathcal{HL}_0 \gtrsim \mathcal{HL}_n$.*

Proof. By induction on n, using the transitive composition of contextual refinement (or structured simulations) and the stepwise simulation in conjecture 5.2.6. □

Theorem 5.2.8 (Compile all cores)**.** $\mathcal{HL}_0 \gtrsim \mathcal{HL}_\infty$.

Proof. Notice that at any point in an execution, the hybrid linker has only instantiated a finite number of new cores (as recored by the counter ec). Using ec in place of the number of threads, the proof follows in direct analogy to the proof of theorem 5.2.2.

□

Corollary 5.2.9 (Separate compilation correctness)**.** *If $P_S = S_0, S_1, \ldots, S_{N-1}$ is a multimodule program with N translation units compiled to $P_T = T_0, T_1, \ldots, T_{N-1}$ (possibly by different compilers) then there is a simulation relation $\mathcal{L}(S_0, S_1, \ldots, S_{N-1}) \gtrsim \mathcal{L}(S_0, S_1, \ldots, S_{N-1})$, between the linked source and compiled programs.*

5.3 Self simulations

When one thread is compiling (using a COAT model), what happens to the other threads? Their code is not changing but the context is, and that might change the memory observable by the thread. For example, if a compiling thread allocates less memory (or more), the other threads will see locations in memory change, even their own allocations.[5] We must prove that the behavior of threads that do not compile is preserved, under the compilation of other threads. This preservation of behavior

[5]CompCert's memory model allocates blocks in order, so the block numbers are not preserved by interference of other allocations. However, the memory structure is preserved up to the memory injection.

can be even stronger than the one preserved by the compiler. For example, we can expect that threads that do not compile allocate the same number of blocks in the same order (even if the block numbers change). We formalize this strong preservation of behavior under changes to the context as *self simulation*.

Definition 5.3.1. *We say that a language L* simulates itself *if there is an* injection relation *between its states and indexed by a memory injection, such that:*

- *States related by the injection relation take steps that have equivalent (up to memory injection) effects on memory and preserve the injection relation,*
- *The injection relation preserves* at_external*, and*
- *The injection relation is monotonic on the memory injection.*

We prove in Coq that Clight and Assembly self-simulate.

Lemma 5.3.2. *Clight and x86 assembly self-simulate.*

Proving self simulation for source and target languages is a necessary step of the COAT technique. However, we only need to prove it twice (for source and target languages)—that is, we do not need to prove it for each intermediate language of CompCert. On the other hand, the COAT technique simplifies the simulation relations that we need to require from every pass of the compiler[6], of which there are more than 16. In general this makes the proof effort easier, and much easier to extend. This is precisely why we find the COAT technique so powerful.

5.4 Simulations for administrative steps

The administrative steps of a CPM (namely RESUME, SUSPEND and STUTTER) are the simplest ones. They do not change the memory, the thread permissions, or the

[6]When only one thread is compiling, we model other threads as the standard external calls that CompCert supports; thus, we do not need the complicated at_external and after_external of [33]. We also simplify the simulation of initial states as explained in section 4.5.

locks. They also leave the thread states mostly unchanged, only modifying the marker (e.g., changing Blocked to Run). The only difficulty, which we solve in this section, is that the RESUME step has to turn the thread-local state from a call state, to a return state by calling afterExternal$(\sigma, v) = \sigma'$ (as it is returning from a synchronization function call).

First, why is afterExternal not called when the synchronization function is executed? That is because other threads interfere with the memory between the execution of the synchronization function and the RESUME step and we want to model all of that behavior as part of the external function. However, at the time the synchronization function executes, we already know how to reestablish the state. Our strategy, detailed in the proof of lemma 5.4.2, is to prove the diagram for the REI.E., SUME step when the function is executed and record that fact in the match relation.

We start by presenting the match relation for marked states (i.e., with Blocked $\cdot$, Run $\cdot$ and Resume $\cdot$). For Blocked states it records the interference of other threads; for Resume states it promises to reestablish the match relation after future interference.

Definition 5.4.1. *Let $\sim$ be a match relation between states of some languages L_1 and L_2. We extend the relation to marked states as follows:*

Init: For any function f and arguments a_1 and a_2 such that $a_1 \overset{j}{\hookrightarrow\!\!\!\!\rightarrow} a_2$, then $Run\ (f,\ a_1) \sim_j Run\ (f,\ a_2)$

Run: If two states match, so do their running version $\sigma_1 \gtrsim \sigma_2 \rightarrow Run\ \sigma_1 \gtrsim Run\ \sigma_2$

Resume: If two states match and some threads interfere with the memory, the blocked states still match.

Resume: Two Resume states match if, for every possible interference of other threads with the memory, we can reestablish a match relation with the resulting states.

Now the definition of match relation for Resume states bakes in the step simulation relation, which now makes the lemma below easy to prove. The definition above

$$\exists\ \Sigma_2'\ j', j \sqsubseteq j'$$

$$\begin{tikzcd}
\Sigma_1 \arrow[r, phantom, "\gtrsim_j"] \arrow[d, "\text{admin}"'] & \Sigma_2 \arrow[d, dotted, "\text{admin}"] \\
\Sigma_1' \arrow[r, phantom, "\gtrsim_{j'}"] & \Sigma_2'
\end{tikzcd}$$

Figure 5.3: CPM administrative step diagrams stated in lemma 5.4.2

shifts the burden of proof to the execution of the synchronization functions (which we consider in section 5.5).

Lemma 5.4.2. *For two hybrid machines with consecutive hybrid bounds ($hb_2 = hb_1 + 1$) the administrative steps form a simulation diagram in lockstep, as depicted in fig. 5.3.*

Proof. We look at each step individually

STUTTER This step is trivial and the simulation is trivial as well.

SUSPEND This is trivial and the simulation is easily constructed. To reestablish the match relation after changing the states to $\mathsf{Blocked}\ \cdot$, we provide an empty interference (case (b) in definition 5.4.1).

RESUME This simulation diagram follows directly from the match relation. After the RESUME step, the new states are marked with Run, so the new simulation is easy to establish (case (a) in definition 5.4.1).

It is worth noting that for this step we are using the (c) case in definition 5.4.1, which is established when the synchronization function is executed, as we show in lemma 5.5.1. □

5.5 Simulations for synchronizations

The synchronization steps (acquire, acqfail, release, make_lock, free_lock and spawn), are the most involved of the CPM steps because they build in the functionality of the synchronization operations. These steps can change the permission maps ($\vec{\pi}$), the lock permissions and locations (L), the memory (m) and even extend the number of threads ($\vec{s}$). In this section we describe the simulation diagrams for these steps, an overview of the mechanized proof is outlined in appendix A.5.2 and the full proof can be found in the Coq code.

Lemma 5.5.1[7]**.** *For two hybrid machines with consecutive hybrid bounds ($hb_2 = hb_1 + 1$) the synchronization steps form a simulation diagram in lockstep.*

Proof. We prove each step separately. The proof for spawn is detailed in lemma 5.5.2. The proofs for steps acquire and release are almost identical and we explain the proof for the former in lemma 5.5.3. The steps make_lock, free_lock and acqfail are relatively simple and do not present any difficulty that is not solved in the cases above, so we omit their proof here. □

5.5.1 Simulation of the Spawn step

In this section we state and prove the simulation of the spawn step for Hybrid machines.

Lemma 5.5.2. *For two hybrid machines with consecutive hybrid bounds ($hb_2 = hb_1 + 1$) the spawn steps form a simulation diagram in lockstep.*

Proof. Let $\Sigma_1 = \langle i \cdot \mho, (\vec{s}_1, \vec{\pi}_1, L_1), m_1\rangle$ and $\Sigma_2 = \left\langle i \cdot \mho, (\vec{s}_2', \vec{\pi}_2', L_2), m_2\right\rangle$, such that

$$\Psi, \Phi \vdash_{\text{HM}} \langle i \cdot \mho, (\vec{s}_1, \vec{\pi}_1, L_1), m_1\rangle \overset{\text{Spa}_{ij}}{\mapsto} \left\langle \mho, (\vec{s}_1', \vec{\pi}_1', L_1), m_1\right\rangle$$

[7]The coq implementation is presented in appendix A.5.2

Then we know that, by the spawn rule in fig. 3.2,

1. $s_1[i] = \mathrm{Blocked}(\sigma_1)$,
2. $\mathrm{at\_external}\,\sigma_1 = \mathrm{Some}(\mathrm{Spawn}(f, a_1))$,
3. $\vec{s}_1' = \vec{s}_1[i \mapsto \mathrm{Resume}(0, \sigma_1),\ j \mapsto \mathrm{Start}(f, a_1)]$ and
4. $\vec{\pi}_1' = \vec{\pi}_1[i \mapsto \pi_1',\ j \mapsto \delta_1'/\{\}]$.

From the match relation $\Sigma_1 \gtrsim_j \Sigma_2$, defined in definition 5.2.4, we can infer:

5. $s_2[i] = \mathrm{Blocked}(\sigma_2)$,
6. $\mathrm{at\_external}\,\sigma_2 = \mathrm{Some}(\mathrm{Spawn}(f, a_2))$ such that $a_1 \overset{j}{\hookrightarrow\!\!\!\rightarrow} a_2$ and
7. there exists π_2' and δ_2', such that $\pi_1' \overset{j}{\hookrightarrow\!\!\!\rightarrow} \pi_2'$, $\delta_1' \overset{j}{\hookrightarrow\!\!\!\rightarrow} \delta_2'$.

So we can construct

8. $\vec{s}_2' = \vec{s}_2[i \mapsto \mathrm{Resume}(0, \sigma_2),\ j \mapsto \mathrm{Start}(f, a_2)]$ and
9. $\vec{\pi}_2' = \vec{\pi}_2[i \mapsto \pi_2',\ j \mapsto \delta_2'/\{\}]$

such that

$$\Psi, \Phi \vdash_{\mathrm{HM}} \langle i \cdot \mho, (\vec{s}_2, \vec{\pi}_2, L_2), m_2 \rangle \overset{\mathrm{Spa}_{ij}}{\mapsto} \left\langle \mho, (\vec{s}_2', \vec{\pi}_2', L_2), m_2 \right\rangle .$$

Finally, we must establish $\left\langle \mho, (\vec{s}_1', \vec{\pi}_1', L_1), m_1 \right\rangle \gtrsim_j \left\langle \mho, (\vec{s}_2', \vec{\pi}_2', L_2), m_2 \right\rangle$, which we do by considering all the requirements of the match relation for CPMs, defined in definition 5.2.4:

(a), (b), (f) are trivial,

(c) Coherence follows by construction.

(d) The injection of permissions follows by construction too.

(e) The memory has not changed: the only permissions that change were those in threads j and i. By 7, above, the injection of the new permissions follows.

(g),(h),(i) The new thread has not been initialized, so the new thread j, is matched in both machines by definition definition 5.4.1 and 6 above. The match relations for all the other threads follow from the old relation $\Sigma_1 \gtrsim_j \Sigma_2$. $\square$

5.5.2 Simulation of the **Acquire** step

In this section we state and prove the simulation of the acquire step for Hybrid machines.

Lemma 5.5.3[8]**.** *For two hybrid machines with consecutive hybrid bounds ($hb_2 = hb_1 + 1$) the acquire steps form a simulation diagram in lockstep.*

Proof. Let i be the thread that is executing the acquire step. We consider three cases for this proof: (1) $i = \mathrm{hb}_1$, (2) $i < \mathrm{hb}_1$ and (3) $i > \mathrm{hb}_1$. In what follows we present the proof for the first case where the executing thread is the one compiling. The other two cases follow the same structure without the complication of being compiled. We split the proof of the diagram in two parts: first we construct Σ_2' and show that Σ_2 can take an acquire step to it, second, we prove $\Sigma_1' \overset{j}{\sim} \Sigma_2'$:

Construction of Σ_2' and the step Let $\Sigma_1 = \langle i \cdot \mho, (\vec{s}_1, \vec{\pi}_1, L_1), m_1\rangle$ and $\Sigma_2 = \left\langle i \cdot \mho, (\vec{s}_2', \vec{\pi}_2', L_2), m_2\right\rangle$, such that

$$\Psi, \Phi \vdash_{\mathrm{HM}} \langle i \cdot \mho, (\vec{s}_1, \vec{\pi}_1, L_1), m_1\rangle \overset{\mathrm{Acq}_i\, a_1\, \delta}{\mapsto} \left\langle \mho, (\vec{s}_1', \vec{\pi}_1', L_1), m_1\right\rangle$$

Then we know that, by the acquire rule in fig. 3.2,

1. $s_1[i] = \mathrm{Blocked}(\sigma_1)$,

[8]The coq implementation is presented in appendix A.5.2

2. $\mathrm{at\_external}\,\sigma_1 = \mathrm{Some}(\mathrm{Acquire}(a_1))$,

3. $m_1|_{\pi_1^2[i]}(a_1) = 1$

4. $m_1[a_1 \mapsto 0] = m_1'$

5. $\pi_1[i] \oplus L_1(a_1) = \pi_1'$ where $\delta_1/\pi_1[i] = \pi_1'$

6. $\vec{s}_1[i \mapsto \mathrm{Resume}(0, \sigma_1)] = \vec{s}_1'$, $\vec{\pi}_1[i \mapsto \pi_1'] = \vec{\pi}_1'$ and $L_1[a_1 \mapsto \mathrm{Some}\ \emptyset] = L_1'$

From the match relation $\Sigma_1 \gtrsim_j \Sigma_2$, defined in definition 5.2.4, we can infer:

7. $s_2[i] = \mathrm{Blocked}(\sigma_2)$,

8. $\mathrm{at\_external}\,\sigma_2 = \mathrm{Some}(\mathrm{Acquire}(a_2))$ such that $a_1 \overset{j}{\hookrightarrow\!\!\!\!\!\rightarrow} a_2$

9. $m_2|_{\pi_2^2[i]}(a_2) = 1$

10. There exists π_2', such that $\pi_2[i] \oplus L_2(a_2) = \pi_2'$.

So we can construct

11. δ_2, such that $\delta_2/\pi_2[i] = \pi_2'$ and that $\delta_1 \overset{j}{\hookrightarrow\!\!\!\!\!\rightarrow} \delta_2$

12. $m_2' = m_2[a_2 \mapsto 0]$

13. $\vec{s}_2' = \vec{s}_2[i \mapsto \mathrm{Resume}(0, \sigma_2)]$,

14. $\vec{\pi}_2' = \vec{\pi}_2[i \mapsto \pi_2']$

15. $L_2' = L_2[a_2 \mapsto \mathrm{Some}\ \emptyset]$

do not Notice that by (11), $\delta_1 \overset{j}{\hookrightarrow\!\!\!\!\!\rightarrow} \delta_2$ and by the match relation $\Sigma_1 \gtrsim_j \Sigma_2$, we also know that the content of memory protected by the lock (i.e., those locations where δ_i has at least Readable permission) are injected. From all the above we can prove:

$$\Psi, \Phi \vdash_{\mathrm{HM}} \langle i \cdot \mho, (\vec{s}_2, \vec{\pi}_2, L_2), m_2 \rangle \overset{\mathrm{Acq}_i\, a_2\, \delta}{\mapsto} \big\langle \mho, (\vec{s}_2', \vec{\pi}_2', L_2), m_2 \big\rangle$$

The match relation: We consider all the requirements of the match relation for CPMs, defined in definition 5.2.4:

(a), (b) are trivial,

(c) The initial states are coherent. In the new states we have the same permissions, albeit in different places, so the new states are also coherent.

(d), (e) The new permission maps (π'_1 and π'_2), have the same permission as before plus the permissions that were in the lock. Both were injected by j, so their addition is also injected by j. The permissions of other threads do not change so, by the old match relation, they are injected as well. The memory has only changed in the lock locations (a_1 and a_2) which are injectable and, by the old match relation, the rest of memory was already in an injection relation so the new memories are as well.

(f) The only locks that change are the acquired ones. The new content of the blocks is empty, which is trivially injected.

(g),(h) The threads that are not compiling are not changing so their match relations still hold.

(i) We expand on the match relation for the compiling thread below.

We have that $\text{Blocked}(\sigma_1) \overset{j}{\sim} \text{Blocked}(\sigma_2)$ and we need to prove $\text{Resume}(0, \sigma_1) \overset{j}{\sim} \text{Resume}(0, \sigma_2)$ using definition 5.4.1. The former relation tells us that, for some previous memories m_1^0 and m_2^0, the states matched (according to the MOIST simulation of CompCert). It also means that the other threads produced interference intf_1^0 and intf_2^0 (which means changing locations in memory not visible by the thread) to produce the current memory (as depicted in fig. 5.4(a)) such that

$$\text{intf}_1^0 \overset{j}{\hookrightarrow\!\!\!\!\rightarrow} \text{intf}_2^0. \tag{5.1}$$

$$\begin{tikzcd}
(\sigma_1, m_1^0) \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "\mathrm{intf}_1^0"'] & (\sigma_2, m_2^0) \arrow[d, dotted, "\mathrm{intf}_2^{0\prime}"] \\
(\sigma_1, m_1) & (\sigma_2, \overline{m_2})
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
(\sigma_1, m_1^0) \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "\mathrm{intf}_1^0"'] & (\sigma_2, m_2^0) \arrow[d, dotted, "\mathrm{intf}_2^{0\prime}"] \\
{} \arrow[d, "\delta_1"'] & {} \arrow[d, dotted, "\delta_2'"] \\
{} \arrow[d, "\mathrm{intf}_1"'] & {} \arrow[d, dotted, "\mathrm{intf}_2'"] \\
(\sigma_1, m_1) & (\sigma_2, \overline{m_2})
\end{tikzcd}$$

(b)

Figure 5.4: Match relation for Blocked

we now know that the acquire step produces injectable events

$$\delta_1 \overset{j}{\hookrightarrow\!\!\!\!\rightarrow} \delta_2 \tag{5.2}$$

and that, in the future, other threads will produce more interference intf_1 and intf_2 (as depicted in fig. 5.4(b)), such that

$$\mathrm{intf}_1 \overset{j}{\hookrightarrow\!\!\!\!\rightarrow} \mathrm{intf}_2. \tag{5.3}$$

Can we establish the match relation of the resulting future states, $(\sigma_1, m_1'') \overset{j}{\sim} (\sigma_2, m_2'')$?

First, the entire execution from (σ_1, m_1^0) to (σ_1, m_1'') constitutes a single step in the Clight MOIST semantics, executing the acquire step

$$\Psi \vdash_{\mathrm{Clight}} (\sigma_1, m_1^0) \xrightarrow{\mathrm{Acq}\ \mathrm{intf}_1^0\ \ \delta_1\ \ \mathrm{intf}_1} (\sigma_1, m_1'') \tag{5.4}$$

and similarly

$$\Phi \vdash_{\mathrm{Asm}} (\sigma_2, m_2^0) \xrightarrow{\mathrm{Acq}\ \mathrm{intf}_2^0\ \ \delta_2\ \ \mathrm{intf}_2} (\sigma_2, m_2''). \tag{5.5}$$

$$\begin{tikzcd}
(\sigma_1, m_1^0) \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "\text{Acq intf}_1^0\ \ \delta_1\ \ \text{intf}_1"'] & (\sigma_2, m_2^0) \arrow[d, dotted, "\text{Acq}\ \overline{\text{intf}_2^0}\ \ \overline{\delta_2}\ \ \overline{\text{intf}_2}'"] \\
(\sigma_1, m_1'') \arrow[r, phantom, "\overset{\bar{j}}{\sim}"] & (\sigma_2, \overline{m_2''})
\end{tikzcd}$$

$$(\text{Acq intf}_1^0\ \ \delta_1\ \ \text{intf}_1) \xhookrightarrow{\bar{j}} (\text{Acq}\ \overline{\text{intf}_2^0}\ \ \overline{\delta_2}\ \ \overline{\text{intf}_2})$$

Figure 5.5: MOIST simulation of acquire steps

By the MOIST simulation of CompCert, we know that there are $\overline{m_2''}$, $\overline{\text{intf}_2^0}$, $\overline{\delta_2}$ and $\overline{\text{intf}_2}$, that complete the step diagram as in fig. 5.5. Since we have $(\sigma_1, m_1'') \overset{\bar{j}}{\sim} (\sigma_2, \overline{m_2''})$, we only need to prove that $\overline{m_2''} = m_2''$.

We know that $(\text{Acq intf}_1^0\ \ \delta_1\ \ \text{intf}_1) \xhookrightarrow{\bar{j}} (\text{Acq}\ \overline{\text{intf}_2^0}\ \ \overline{\delta_2}\ \ \overline{\text{intf}_2})$, which means that

$$\text{intf}_1^0 \xhookrightarrow{\bar{j}} \overline{\text{intf}_2^0} \qquad \delta_1 \xhookrightarrow{\bar{j}} \overline{\delta_2} \quad \text{and} \quad \text{intf}_1 \xhookrightarrow{\bar{j}} \overline{\text{intf}_2}.$$

Because $\hookrightarrow$ is deterministic and because of equations 5.1, 5.2, and 5.3, then

$$(\text{Acq}\ \overline{\text{intf}_2^0}\ \ \overline{\delta_2}\ \ \overline{\text{intf}_2}) = (\text{Acq intf}_2^0\ \ \delta_2\ \ \text{intf}_2)$$

Because external functions in CompCert are deterministic, up to the event, then the step in eq. (5.5) and the right step in fig. 5.5 are the same and thus $\overline{m_2''} = m_2''$. $\square$

Chapter 6

CSL soundness proof

This section describes how the Concurrent Permission Machine supports a rich logic like Concurrent Separation Logic. A key idea in this process is to define the Juicy Concurrent Machine, which mirrors the CPM but with "juicy" memories and resources instead of CompCert memories and permission maps. We start with an overview of CSL, then "juicy" semantics for sequential code and concurrent code (Juicy Concurrent Machine). Finally we show that the CPM simulates the Juicy permission Machine, by erasing the juicy resources into regular permission.

6.1 Concurrent Separation Logic

Our CSL's root judgment has the form, $\Gamma \vdash_{\text{CSL}} \Psi : \Gamma'$, where Γ is a list of function specs and Ψ is a list of function bodies. It means, assuming all the functions satisfy their specifications in Γ, then each function-body in Ψ does satisfy its specification in Γ'. Then, one proves $\Gamma \vdash_{\text{CSL}} \Psi : \Gamma$ to prove that a set of mutually recursive functions satisfy all their specs. (This is not circular reasoning! [4, Equation 81].)

For each function body c one must prove $\Delta \vdash_{\text{CSL}} \{P\}c\{Q\}$, where Δ incorporates both the global Γ and this function's local-variable declarations, P is a precondition, and Q is a set of postconditions [3, Ch. 24,25].

$$\frac{\text{writable\_share } sh}{\{e \Downarrow v \wedge v \overset{sh}{\mapsto}_{\text{lock}} \_\}\ \ \textbf{makelock}\ e\ \ \{v \overset{sh}{\boxdot\!\!\rightarrow} R\}}$$

$$\frac{\text{writable\_share } sh \qquad \text{precise } R \qquad \text{positive } R}{\{e \Downarrow v \wedge v \overset{sh}{\boxdot\!\!\rightarrow} R * R\}\textbf{freelock}\ e\,\{v \overset{sh}{\mapsto}_{\text{lock}} \_ * R\}}$$

$$\frac{\text{readable\_share } sh}{\{e \Downarrow v \wedge v \overset{sh}{\boxdot\!\!\rightarrow} R\}\ \ \textbf{acquire}\ e\ \{R * v \overset{sh}{\boxdot\!\!\rightarrow} R\}}$$

$$\frac{\text{readable\_share } sh \qquad \text{precise } R \qquad \text{positive } R}{\{e \Downarrow v \wedge\ R * v \overset{sh}{\boxdot\!\!\rightarrow} R\}\ \ \textbf{release}\ e\ \ \{v \overset{sh}{\boxdot\!\!\rightarrow} R\}}$$

$$\frac{\text{EVAL} = e_1 \Downarrow f \wedge e_2 \Downarrow v}{\{EVAL \wedge f : \{P\}\{\text{F}\} \wedge P(v)\}\textbf{spawn}\, e_1(e_2)\{\text{emp}\}}$$

Figure 6.1: Concurrent Separation Logic. $e \Downarrow v$ means that e evaluates to v. $v \overset{sh}{\mapsto}_{\tau} w$ means that at address v, the value w is laid out in memory as a struct of type τ. Share sh is in a lattice between $\circ$ (empty) and $\bullet$ (full); some shares give enough permission for writing (writable_share sh), and those can be split into readable_shares that give only read permission (or permission to acquire a lock). $f : \{P\}\{Q\}$ says that address f is a function with precondition P and postcondition Q. To spawn f, the precondition must be satisfied, and the footprint of $P(v)$ in the parent's precondition disappears into the child. We require Q = False; threads must explicitly call thread-exit (but in this proof we omit thread-exit).

The CSL rules for the concurrency operations are shown in fig. 6.1.

Makelock associates with its lock a *resource invariant*, a separation-logic predicate that characterizes both the *footprint* (set of addresses controlled by the lock) and a predicate the data at these addresses must satisfy whenever the lock is unlocked. The footprint need not be static; for example, $l \boxdot\!\!\rightarrow \exists q.p \mapsto q * q \rightsquigarrow 0$ says, "address l is a lock controlling access to the linked list whose head-pointer q is stored at address p." If q is 0, the footprint is just $\{p\}$, but if $q \mapsto (1, q') * q' \mapsto (2, 0)$ then the footprint is $\{p, q.\text{hd}, q.\text{tl}, q'.\text{hd}, q'.\text{tl}\}$.

Suppose thread t_1 creates a new linked-list data structure p and a new lock l_p to control it, so $l_p \boxdot\!\!\rightarrow \exists q.p \mapsto q * q \rightsquigarrow 0$. Now thread t_1 wants to tell another thread t_2 about this lock. Either t_1 spawns function f as thread t_2, passing l_p as an argument; or t_1 stores l_p into a shared data structure s controlled by lock l_s and then releases l_s

so that t_2 can acquire l_s. In either case, one separation-logic predicate describes the binding of a lock-address to its resource invariant, another separation logic predicate. Having one predicate operate on another can lead to paradox if not treated carefully, so we use a step-indexed model of a modal separation logic [17].

6.1.1 Impredicativity and the `spawn` Rule

All of the CSL rules are impredicative—that is, they quantify over a CSL predicate, the resource invariant associated with the lock. Impredicativity gives the power to specify higher-order functions such as `spawn`: the `spawn` rule contains not just a predicate but an assertion that a certain pre- and postcondition are associated with its argument, the function to be spawned.

6.1.2 Ghost State

When proving correctness of complicated concurrent programs, we may want to keep track of more information than just shares of memory locations. Concurrent separation logic becomes much more powerful with the addition of *ghost state*, structured auxiliary state that can be introduced and manipulated as part of the CSL proof. Ghost state is particularly useful for establishing *protocols* on the use of shared resources, so that we can reason more precisely about concurrent interactions. For instance, the program in fig. 6.2 (a C program proved in Coq, but here shown in pseudocode rather than C) increments a lock-protected shared variable twice in parallel. In basic CSL, we can prove that the program executes safely according to a lock invariant such as $\mathtt{x} \geq 0$, but not that $\mathtt{x} = 2$ at the end of the program. Using ghost state, we can record the contributions made by each thread to `x` in ghost variables shared between the threads and the lock invariant, allowing us to deduce the precise value of `x` at the end of the program.

```
               x = 0;
acquire(l); || acquire(l);
x++;        || x++;
release(l); || release(l);
```

Figure 6.2: The increment example

The CSL of VST includes ghost state in the style of Iris [19], in which any *partial commutative monoid* (PCM) can be used as ghost state, including posets, histories, and state machines. VST's logic is like Iris's but builds on separation algebras instead of resource algebras; in either case, these are PCMs with a few additional properties.

Definition 6.1.1. *A* ghost algebra *is a separation algebra with a* valid *predicate, such that*
$\mathsf{valid}(a \cdot b) \Rightarrow \mathsf{valid}(a)$.

The CSL uses a separation logic assertion $\mathsf{own}(g, a)$ to assert that the current thread owns ghost state a at name g. (Note that $\mathsf{valid}(a)$ is a mathematical proposition, while $\mathsf{own}(g, a)$ is a spatial assertion.) This assertion is introduced and manipulated with the rules shown in fig. 6.3, using a *view shift* operator $\Rrightarrow$ in the style of Iris. This operator subsumes logical implication, and allows us to make frame-preserving updates to ghost state at any point in a proof.

$$\frac{P \Rrightarrow P' \qquad \{P'\}c\{Q'\} \qquad Q' \Rrightarrow Q}{\{P\}c\{Q\}} \qquad \frac{\mathsf{valid}(a)}{\mathrm{emp} \Rrightarrow \exists g, \mathsf{own}(g, a)} \qquad \frac{}{\mathsf{own}(g, a) \Rrightarrow \mathrm{emp}}$$

$$\frac{}{\mathsf{own}(g, a \cdot b) = \mathsf{own}(g, a) * \mathsf{own}(g, b)} \qquad \frac{}{\mathsf{own}(g, a) \Rightarrow \mathsf{valid}(a)} \qquad \frac{a \rightsquigarrow b}{\mathsf{own}(g, a) \Rrightarrow \mathsf{own}(g, b)}$$

Figure 6.3: CSL rules for ghost state.

With locks and ghost state, we can prove correctness of a wide range of concurrent programs. The power of the CSL has been demonstrated in an application to a messaging system for shared-memory communication in autonomous vehicles [27].

This messaging system provides an API through which a writer can make values available to a collection of readers efficiently, while protecting against interference from potentially malicious readers and writers. The proof makes use of ghost variables and histories to track the readers' knowledge of the writer's state and vice versa, using this ghost state to define the protocol by which shares of data buffers are passed between components. The program itself is 170 lines of C code, and the proof in VST is 3600 lines of Coq, showing that readers that use the API correctly always receive the most recently written value.

6.1.3 Mechanization

VST includes a proof automation system for interactively verifying C programs [8], consisting of 1) derived Hoare rules restructured for easier automation and 2) Coq tactics for symbolically executing program instructions and automatically proving separation logic entailments. For the most part, this system can be used as is to verify concurrent programs in our CSL: release, acquire, etc. are treated in the same way as normal function calls, and can be proved with the existing forward_call tactic. We have added new tactics ghost_alloc and viewshift_SEP for introducing and manipulating ghost state, and a forward_spawn tactic for applying the spawn rule, bridging the gap between the precondition of spawn and that of the spawned function. We used these features to verify the C programs described above.

In the next two sections we prove that programs with a correctness proof in Concurrent Separation Logic are safe in the Concurrent Permission Machine.

6.2 Juicy Sequential semantics

The application programmer says, "I am the center of the world. My thread may call functions, even external functions that acquire or release locks, but they return to me just where I left off. Thus I can reason sequentially using the Hoare logic $\{P\}c\{Q\}$, where c may call external functions."

The thread-library author says, "I am the center of the world. I permit a thread to resume, and after a time it comes back to me with a synchronization request. Perhaps it has pushed or popped its function-call stack in the meantime; that is not my concern; I have many other threads to manage too."

We reconcile these two views by presenting two operational semantics. In section 6.2 we define the Juicy Sequential semantics, and prove a theorem about it; in section 6.3 we define the Juicy Concurrent semantics, and prove a theorem; the composition of these theorems demonstrates that the CSL is sound with respect to CPM executions.

6.2.1 Review of VST semantic model

Appel *et al.* [3] developed a semantic model of sequential separation logic with higher-order features intended to support concurrent separation logic. In this section, we review that logic and model; then we describe the new work that implements and proves CSL.

To reason about concurrent-read/exclusive-write, the separation logic predicates (fig. 6.1) use a *share* lattice: a write-share can be split into read-shares, which can be further split into more read-shares [3, Chapters 11,41]. Also, our predicates permit the description of how a lock address is associated with its resource invariant (which

$$\frac{\Psi \vdash_{\text{CompCert}} \langle \sigma, m \rangle \overset{\epsilon}{\mapsto} \langle \sigma', m' \rangle \qquad \phi \overset{m,m'}{\rightsquigarrow} \phi'}{\mathbf{E}, \Psi \vdash_{\text{JuicySeq}} \langle \sigma, \langle \phi, m, \mu \rangle \rangle \mapsto \langle \sigma', \langle \phi', m', \mu' \rangle \rangle} \quad (\text{CORE})$$

$$\frac{\begin{array}{c}\text{atExternal } \sigma = \text{Some}(f, x) \qquad \mathbf{E}(f) = \{P\}\{Q\} \\ Pyx(jm) \qquad Qyr(jm') \qquad \text{afterExternal}(\sigma, r) = \sigma'\end{array}}{\mathbf{E}, \Psi \vdash_{\text{JuicySeq}} \langle \sigma, jm \rangle \mapsto \langle \sigma', jm' \rangle} \quad (\text{EXTERNAL})$$

$$\frac{\begin{array}{c}\text{atExternal } \sigma = \text{Some}(f, x) \qquad \mathbf{E}(f) = \{P\}\{Q\} \\ Pyx(jm) \qquad \neg\exists r, jm'.\ Qyr(jm')\end{array}}{\mathbf{E}, \Psi \vdash_{\text{JuicySeq}} \langle \sigma, jm \rangle \mapsto \langle \sigma, jm \rangle} \quad (\text{NO-RETURN})$$

Figure 6.4: Juicy Sequential machine, a thread-modular view of a concurrent execution. External specification $\mathbf{E}$ is instantiated with $\mathbf{E}_{\text{CSL}}$, the Hoare rules of fig. 6.1.

is another predicate); this permits reasoning about programs that dynamically create new locks [3, Chapter 30].

The semantic model of our predicates (and our Hoare triple) is with respect to a C semantics with enriched states that keep track of shares, predicate bindings, and ghost state; we call these *juicy memories*. The CompCert C semantics has a simpler permission-lattice than our shares (see section 4.1), and has no notion at all of "predicates in the heap." At a lower level of our proof, an erasure theorem relates our juicy C semantics to a CompCert C semantics.

We put shares, predicate-bindings, and ghost state into a state component ϕ called a *resource map*, a mapping from address to *resource*. A resource specifies a share; the resource type (ordinary value, callable function, lock); for value-type resources, the CompCert value stored at that location; for functions, the specification (pre/postcondition); for semaphores, the resource invariant. Each resource map also contains a finite map of ghost elements: each ghost name is mapped to a dependent pair of a ghost algebra and an element of that algebra. We can add new types of ghost state (i.e., new ghost algebras) on the fly during execution by extending the map. Our semantic model is step-indexed [17] to allow higher-order impredicative quantification. Each resource map ϕ has an *approximation level*, level(ϕ), applied to both invariants and ghost state. Higher levels carry more accuracy; as the program small-steps, the levels of all the ϕ decrease in lockstep.

Resource maps form a separation algebra with a join operator $\phi_1 \oplus \phi_2 = \phi$, a partial function corresponding to the separating conjunction: $\phi_1 \models P_1, \;\; \phi_2 \models P_2,$ $\phi \models P_1 * P_2$.

A CompCert memory is a map from addresses to values and permissions; we couple a resource-map ϕ and a memory m as a *juicy memory* $jm = \langle \phi, m, \mu \rangle$ where μ is a proof that the permission-shares and values claimed by ϕ at each address correspond to the CompCert *current* permission and contents of m. The predicates ($\{P\}\{Q\}$ for functions and R for locks) have no analog in the "dry" memory m.

Our "juicy sequential small-step semantics" (fig. 6.4) steps $\Psi \vdash_{\text{JuicySeq}} \langle \sigma, jm \rangle \mapsto \langle \sigma', jm' \rangle$. The CORE rule is a restriction of the CompCert step (a juicy state steps *only if* its internal CompCert state also steps). Thus, proving *erasure* (from a CORE small-step in Juicy Sequential to a small-step in CompCert) is trivial. But we need to reconstruct the new jm' after a core step; hence this lemma is useful:

Lemma 6.2.1 (from Chapter 42 of [3], extended with ghost state). *When μ proves that ϕ corresponds with m ($jm = \langle \phi, m, \mu \rangle$) and then m evolves to m' by allocating some blocks, storing at some addresses, and deallocating some blocks, we can can find ϕ' and μ' such that $jm' = \langle \phi', m', \mu' \rangle$, and also $\phi \overset{m,m'}{\leadsto} \phi'$ (that is, ϕ, ϕ' agree on all other addresses).*

External calls. Stewart [34, page 119] shows how an interaction semantics small-steps across an *external* function call $f(x)$. There is no constructive semantics for such steps; instead, we have a *specification* for external functions, in the form of a Hoare triple: $\mathbf{E}(f) = \{P\}\{Q\}$. P is a predicate on "ghost value" y, function parameter x, and juicy-memory jm. The idea is that if (for some y) $Pyx(jm)$ holds as $f(x)$ is called in state jm, then (for that y) $Qyr(jm')$ will hold on the return value r in the after-call state jm'. The y allows communication between precondition and postcondition.

We instantiate the external specification $\mathbf{E}$ with our $\mathbf{E}_{\text{CSL}}$ that specifies Hoare triples for the synchronization operations (acquire, release, makelock, etc.).

When the Juicy Sequential reaches an *external* call, it evaluates the external-function address f and arguments $\vec{x}$. Then it looks up the *specification* for f, and (non constructively) finds y such that $Pyx(jm)$ holds. Then it (angelically) finds r, jm' such that $Qyr(jm')$ holds. Finally, the afterExternal operation of our interaction-semantics model folds the return value r back into the local state [6].

But perhaps the external function's postcondition is false. In Hoare logic, that means (essentially) that the function never returns; it is *quite* safe to call such a function (provided you satisfy its precondition). Here we present this as a NO-RETURN rule; our Coq proofs (and [34, pages 119–120]) present the definition of safety more directly.

It may seem backwards to derive the small-step relation from the Hoare specification! Indeed, it is the job of the *linking theorem* to show that the appropriate y, r, m' will exist— that the external system $\mathbf{E}$ satisfies its Hoare-logic specification. Stewart [34, Thm. 9] showed such a linking theorem for modular verification of separately compiled modules. theorem 6.3.4 will be a different linking theorem, for adding synchronization operators to our core-language *Clight* semantics. By this axiomatic treatment of acquire and release, we avoid the need for Hobor's oracular semantics [16].

Step-indexes. A juicy memory $jm = \langle \phi, m, \mu \rangle$ contains a resource-map ϕ labeled with a step-index *level*. The step-indexed predicates (resource invariants, function preconditions) bound to addresses in ϕ are accurate only to that level. We define $\text{level}(jm) = \text{level}(\phi)$.

Definition 6.2.2 (Sequential safety)**.** *A Juicy Sequential state* $\langle\sigma, jm\rangle$ *is* safe, *written* $\mathbf{E}, \Psi \vdash_{\text{JuicySeq}} \text{safe}\,\langle\sigma, jm\rangle$, *if it cannot reach a stuck state within* $k = \text{level}(jm)$ *steps in the* $\vdash_{\text{JuicySeq}}$ *relation. [3, page 393]*

To prove that a program is safe for $k = 10^{99}$ steps, choose the program's initial ϕ to have level k. Quantification over the desired execution length k is done outside any other quantification, so we can reason about any finite prefix of the execution of program Ψ.

Our CSL judgments $\Gamma \vdash_{\text{CSL}} \Psi : \Gamma'$ and $\Delta \vdash_{\text{CSL}} \{P\}c\{Q\}$ are *not* (deeply embedded / syntactic) inductive predicates! They are (shallowly embedded) semantic definitions. $\{P\}c\{Q\}$ means *by definition* [3, Chapter 43] that in any state satisfying P, it is safe to run c, and if c finishes, it will do so in a state satisfying Q.

fig. 6.1 does *not* show the separation logic rules for the sequential C language. Appel *et al.* present those rules [3, Chapter 24] and prove them w.r.t. the semantic model of the the Juicy Sequential semantics [3, Part VI]. That proof is entirely parametric over $\mathbf{E}$. So nothing changes when we now introduce concurrency into the language and program logic: our CSL includes all the rules of sequential SL.

6.2.2 New result: Concurrent Separation Logic is sound

Subsection 6.2.1 described previous work, with the exception of the enhancement of resource maps (and CSL predicates) to handle ghost state, which is new. This subsection describes a new result.

We prove that if a C program is proved correct (to any specification) in concurrent separation logic, and its initial thread starts in a state satisfying its precondition, then its (pseudo)sequential execution ($\vdash_{\text{JuicySeq}}$) will be safe. Safety in $\vdash_{\text{JuicySeq}}$ implies partial correctness—not shown in fig. 6.4 is the rule [34, pages 119–120] that checks the program's postcondition when the program finishes.

Theorem 6.2.3 (Separation Logic Soundness)**.** *Suppose all the functions in program* Ψ *satisfy their specifications in* Γ*, by a derivation of* $\Gamma \vdash_{\mathrm{CSL}} \Psi : \Gamma$*. Suppose* $\Gamma(f) = \{P\}\{Q\}$*, that is, calling* f *with argument* x *and ghost value* y *we have the Hoare triple* $\{Pyx\}\, r := f(x)\, \{Qyr\}$*. Let* σ *be a local state in which the current command is the function-call* $f(v)$*. Let* jm *be a juicy memory such that* (σ, jm) *satisfies* Pyv*. Then* $\mathbf{E}_{\mathrm{CSL}}, \Psi \vdash_{\mathrm{JuicySeq}} \mathrm{safe}\,\langle \sigma, jm\rangle$*.*

Proof. By unfolding the definition of our $\vdash_{\mathrm{CSL}}$ judgment. □

6.3 The juicy concurrent machine

By theorem 6.2.3, the CSL is sound with respect to a predicate-annotated, permission-annotated, (pseudo)sequential operational semantics, the *Juicy Sequential Machine.* The predicate annotations—*resource invariants* of CSL—specify how permissions are transferred by synchronizations. Safety (correctness) in the (pseudo)sequential semantics implies safety (correctness) in a predicate-annotated, permission-annotated, operational semantics for cooperative concurrency, the *Juicy Concurrent Machine.* At the end of this section, we show how to erase the predicates: the *Juicy Concurrent Machine* is simulated by the CPM.

fig. 6.5 shows the Juicy Concurrent machine, which may be seen as a variant of a CPM where CompCert permissions are replaced by the the more expressive notion of *resources* and no angelic guessing is necessary. States take the form $\left\langle \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle$, where $\mho$, $\vec{s}$, and m are as before, and

- $\vec{\phi}$ is a list of resource maps. Each ϕ_i is a thread's own view of memory and ghost state.

- L is a function from address to option(option(resource-map)), indicating the state of each lock: $L(a) =$ None means that a is not a lock, Some(None) means that

$$\frac{\begin{array}{c} s_i = \langle \mathrm{Run}, \sigma \rangle \qquad \phi_i \overset{m|_{\phi_i}, m'}{\rightsquigarrow} \phi' \qquad \Psi \vdash_{\mathrm{CompCert}} \left\langle \sigma, m|_{\phi_i} \right\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m' \rangle \\ \vec{s}\,' = \vec{s}[i \mapsto \langle \mathrm{Run}, \sigma' \rangle] \qquad \vec{\phi}\,' = \vec{\phi}[i \mapsto \phi'] \end{array}}{\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle i \cdot \mho, (\vec{s}\,', \vec{\phi}\,', L), m' \right\rangle}$$

(CORE)

$$\frac{\begin{array}{c} s_i = \mathrm{Blocked}(\sigma) \qquad \mathrm{atExternal}\ \sigma = \mathrm{Some}(\mathrm{Acquire}, a) \qquad \phi_i(a) = \mathrm{Lock}\ R \\ m|_{\hat{L}}(a) = 1 \qquad m|_{\hat{L}}[a \mapsto 0] = m' \qquad L(a) = \mathrm{Some}(\mathrm{Some}\,\delta) \qquad \phi_i \oplus \delta = \phi' \\ \vec{\phi}[i \mapsto \phi'] = \vec{\phi}\,' \qquad \vec{s}[i \mapsto \mathrm{Resume}(0, \sigma)] = \vec{s}\,' \qquad L[a \mapsto \mathrm{Some}(\mathrm{None})] = L' \end{array}}{\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho, (\vec{s}\,', \vec{\phi}\,', L'), m' \right\rangle}$$

(ACQUIRE)

$$\frac{s_i = \mathrm{Blocked}(\sigma) \quad \mathrm{atExternal}\ \sigma = \mathrm{Some}(\mathrm{Acquire}, a) \quad \phi_i(a) = \mathrm{Lock}\ R \quad m(a) = 0}{\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle}$$

(ACQFAIL)

$$\frac{\begin{array}{c} s_i = \mathrm{Blocked}(\sigma) \qquad \mathrm{atExternal}\ \sigma = \mathrm{Some}(\mathrm{Release}, a) \qquad \phi_i(a) = \mathrm{Lock}\ R \\ m|_{\hat{L}}(a) = 0 \qquad m|_{\hat{L}}[a \mapsto 1] = m' \qquad L(a) = \mathrm{Some}(\mathrm{None}) \qquad \delta \models R \qquad \delta \oplus \phi' = \phi_i \\ \vec{s}[i \mapsto \mathrm{Resume}(0, \sigma)] = \vec{s}\,' \quad \vec{\phi}[i \mapsto \phi'] = \vec{\phi}\,' \quad L[a \mapsto \mathrm{Some}(\mathrm{Some}\,\delta)] = L' \end{array}}{\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho, (\vec{s}\,', \vec{\phi}\,', L'), m' \right\rangle}$$

(RELEASE)

$$\frac{\begin{array}{c} s_i = \mathrm{Blocked}(\sigma) \qquad \mathrm{atExternal}\ \sigma = \mathrm{Some}(\mathrm{Spawn}(f, a)) \qquad \phi_i(f) = \mathrm{Func}\{P\}\{Q\} \\ \delta \models P(a) \qquad \delta \oplus \phi' = \phi_i \\ \vec{s}[i \mapsto \mathrm{Resume}(0, \sigma)] \cdot \langle \mathrm{Start}(f, a) \rangle = \vec{s}\,' \qquad \vec{\phi}[i \mapsto \phi'] \cdot \delta = \vec{\phi}\,' \end{array}}{\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho, (\vec{s}\,', \vec{\phi}\,', L'), m \right\rangle}$$

(SPAWN)

Figure 6.5: Juicy Concurrent machine. Rules for START, RESUME, SUSPEND are not shown here; they look identical to those of the CPM (Fig. 3.3) except that they use ϕ (resource map) instead of π (permission). Rules for MAKELOCK, FREELOCK are omitted. Event traces ϵ are omitted; see the supplement (part A).

a is locked. Some(Some ϕ_a) means a is unlocked and ϕ_a is the resource that a thread would obtain by acquiring a.

m is the global memory, shared by all threads.

We say coherent$(\vec{\phi}, L, m)$ when the resource maps in $\vec{\phi}$ and L join together to a ϕ_{tot} that agrees with m on max permissions and the contents of memory cells. By the nature of our join relation, that also means that the ϕ_i and the ϕ_a all have the same step-index level and are all mutually noninterferent; at no address does ϕ_i grant write permission and ϕ_j $(j \neq i)$ grant read or write permission. (Not shown in fig. 6.5: the ACQUIRE and RELEASE rules must "age" $\vec{\phi}'$ and L' by one step-index.)

The operator $m|_{\phi_i}$ constructs a memory like m but whose current permissions are restricted to ϕ_i, so that $\langle \phi_i, m|_{\phi_i}, \mu \rangle$ is a juicy memory (provided that ϕ_i and m agree about the value at each address). The CORE rule of the JuicyConc semantics says that the machine small-steps a thread ensuring that the core step (in Clight or assembly language) does not interfere with other threads' data (and with resources stashed in unlocked locks). We write $m|_{\hat{L}}$ to set write-permission in m at those addresses that L identifies as locks.

The CORE rule implements cooperative concurrency: it learns from the schedule that the ith thread is to be stepped, and leaves i at the head of the schedule. When the thread reaches atExternal (a SUSPEND step), i will be consumed. This enforces that at most one thread is marked Run.

Definition 6.3.1 (State Invariant). *Given program specification Γ and JuicyConc state*

$S = \left\langle \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle$, *we say* StateInvar$_n(\Gamma, S)$ *when,*

- *All the resource maps in $\vec{\phi}$ and L join together to make a total resource map ϕ_{tot}, whose step-index level is n; note that this implies* coherent$(\vec{\phi}, L, m)$.

- *At most one s_i is marked as* Run, *and (if so) i is at the head of $\mho$.*

- *The function specifications in* ϕ_{tot} *are exactly those of* Γ.
- *Whenever* $L(a) = \mathrm{Some}(\mathrm{Some}\ \phi_a)$*, then* $\exists R.\ \phi_{\mathrm{T}}(a) = \mathrm{Lock}\ R$ *with resource invariant* R *and* $\phi_a \models R$*. Otherwise, if* $L(a) = \mathrm{Some}(\mathrm{None})$*, then* $\exists R.\ \phi_{\mathrm{T}}(a) = \mathrm{Lock}\ R$*, and if* $L(a) = \mathrm{None}$ *then* $\phi_{\mathrm{T}}(a)$ *is not of the form* $\mathrm{Lock}\ R$.
- *Each thread is safe: the state* $\langle \sigma_i, \langle \phi_i, m|_{\phi_i}, \mu \rangle \rangle$ *cannot get stuck in* $\vdash_{\mathrm{JuicySeq}}$ *within* n *steps.*

Lemma 6.3.2 (Safety induction)**.** *If* $\Gamma \vdash_{\mathrm{CSL}} \Psi : \Gamma$ *and* $\mathrm{StateInvar}_n(\Gamma, S)$ *and* $n > 0$*, then there is a state* S' *and* $n - 1 \leq n' \leq n$ *such that* $\Psi \vdash_{\mathrm{JuicyConc}} S \mapsto S'$ *and* $\mathrm{StateInvar}_{n'}(\Gamma, S')$.

Proof. See the supplement. □

Definition 6.3.3 (Safety)**.** *A JuicyConc state is* safe_k*, written*
$\Psi \vdash_{\mathrm{JuicyConc}} \mathrm{safe}_k \left\langle \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle$*, when either*

- $\mho = \mathrm{nil}$ *or* $k = 0$,
- $\Psi \vdash_{\mathrm{JuicyConc}} \left\langle \mho, (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho, (\vec{s}', \vec{\phi}', L'), m' \right\rangle$
  and $\Psi \vdash_{\mathrm{JuicyConc}} \mathrm{safe}_{k-1} \left\langle \mho, (\vec{s}', \vec{\phi}', L'), m' \right\rangle$*, or*
- $\mho = i \cdot \mho'$ *and* $\Psi \vdash_{\mathrm{JuicyConc}} \left\langle i \cdot \mho', (\vec{s}, \vec{\phi}, L), m \right\rangle \mapsto \left\langle \mho', (\vec{s}', \vec{\phi}', L'), m' \right\rangle$
  and $\forall \mho''.\ \Psi \vdash_{\mathrm{JuicyConc}} \mathrm{safe}_{k-1} \left\langle \mho'', (\vec{s}', \vec{\phi}', L'), m' \right\rangle$.

This is *almost* a conventional inductive definition of safety, except that after consuming one step of schedule $\mho$, the remaining state must be safe for *all possible* schedules.

Theorem 6.3.4 (Safety of the Juicy Concurrent machine)**.** *We wish to run* Ψ *for* k *steps. Suppose* Ψ *satisfies its specification* Γ*, that is,* $\Gamma \vdash_{\mathrm{CSL}} \Psi : \Gamma$*. Let* m_0 *be the* *initial memory for* Ψ*. Let* σ_0 *be the initial state, calling* **main()***. Let* $\langle \phi_0, m_0, \mu_0 \rangle$ *be a juicy memory with* $\text{level}(\phi) = k$ *(one can always construct this). Let* $L_0 = \lambda a.\text{None}$ *be the (empty) initial lock pool. Let* $\mho$ *be any schedule. Then the Juicy Concurrent state* $\langle \mho, ([\langle \text{Run}, \sigma_0 \rangle], [\phi_0], L_0), m_0 \rangle$ *is* safe_k*.*

Proof. StateInvar holds initially, since L is empty and $\vec{\phi}$ has only one thread. Then we apply lemma 6.3.2. However, because of our stronger notion of safety, we also use a lemma stating that StateInvar is preserved when changing the "remainder" of the schedule. □

6.4 CPM simulates the Juicy Concurrent Machine

Definition 6.4.1 (Erasure from JuicyConc to CPM)**.** *Let* S *be a JuicyConc state. We define* $\hat{S}$*, the erasure of* S*, as the CPM state obtained by converting all resource maps* ϕ *to permission maps* π*.*

Theorem 6.4.2 (Erasure)**.** *For any juicy step* $\Psi \vdash_{\text{JuicyConc}} S \mapsto S'$*, there exists a Concurrent Permission Machine step* $\Psi \vdash_{\text{CPM}} \hat{S} \overset{\epsilon}{\mapsto} \hat{S}'$*.*

Proof. Converting ϕ to π is easy. But we must demonstrate guesses δ such that CPM execution is not stuck. These guesses come from the resource maps δ in the ACQUIRE, RELEASE, and SPAWN rules of $\vdash_{\text{JuicyConc}}$. Traces ϵ are derived from the other constraints of a CPM execution. □

Corollary 6.4.3 (CSL implies the CPM is safe)**.** *If a Clight program is proved correct in Concurrent Separation Logic, then its execution in the Concurrent Permission Machine is safe.*

Proof. By theorems 6.3.4 and 6.4.2. □

Chapter 7

Instruction interleaving and well synchronization

This section describes how the Concurrent Permission Machine relates to a realistic weakly consistent memory model. There are three important results, first the cooperative CPM can be simulated by a preemptive CPM (the interleaving proof, section 7.1); second, safe CPM executions are well-synchronized and execute safely on X86-32 (section 7.2); and third, the permission in the CPM don't change the behavior of a safe programs, so they can be erased (section 7.3).

7.1 Instruction interleaving

From source code down to assembly language we reason about CPMs with cooperative concurrency: each thread executes undisturbed until it explicitly calls a synchronization function such as acquire or release.

Because the threads have noncompeting memory permissions, (preemptive) instruction-level interleaving of the threads should execute with equivalent behavior. We formalize this as a safety-preservation proof.

Definition 7.1.1. *We define* $\vdash_{\text{FineConc}}$*, the fine-grained concurrent machine, just like the Concurrent Permission Machine except that all rules "consume" the head of the schedule. We illustrate for the* CORE *rule:*

$$\frac{\begin{array}{c} s_i = \langle \text{Run}, \sigma \rangle \qquad \Psi \ \vdash_{\text{CompCert}} \ \left\langle \sigma, m|_{\pi_i^1} \right\rangle \overset{\epsilon}{\mapsto} \langle \sigma', m' \rangle \qquad \text{upmax}(m', m'') \\ \vec{s}' = \vec{s}[i \mapsto \langle \text{Run}, \sigma' \rangle] \qquad \vec{\pi}' = \vec{\pi}[i \mapsto (\text{Cur}(m'), \pi_i^2)] \end{array}}{\Psi \vdash_{\text{FineConc}} \langle i \cdot \mho, (\vec{s}, \vec{\pi}, L), m \rangle \overset{\epsilon_i}{\mapsto} \langle \mho, (\vec{s}', \vec{\pi}', L), m'' \rangle}$$

*(*CORE*)*

Compared to the CPM CORE rule: the schedule after the step is $\mho$ here, versus $i \cdot \mho$ in CPM; and upmax increases maximum memory permissions. When the CPM chooses thread i, it keeps choosing i until the thread reaches atExternal; but FineConc consumes i from $\mho$, and the next step may run another thread. After each step, m's max-permissions increase to Freeable. Max-permissions were used to prove CompCert's optimizations correct, but are no longer useful after compilation; upmax simplifies the proofs in the next stage. Any nonstuck execution without upmax will produce the same sequence of memory operations with the increased permissions. This *upmax* has no effect on Cur permissions: in the FineConc machine, the Cur permissions of threads are still noncompeting, and threads are still stuck if they try to load/store without Read/Write Cur permission.

Because threads' memory permissions don't compete, any two interleavings that only change the order of execution of core steps should exhibit the same observable behaviors and result in equivalent memories (and thread states).

What if a thread never reaches an external call (because it infinite-loops, or the FineConc machine's schedule is not long enough)? Removing this thread from the schedule will not change the observable behavior of the execution—no other thread can observe its writes, because it never releases a lock. Obviously, the thread states of the two executions will no longer be related. This does not matter because only external steps affect the observable behaviors of the machine—unless the thread gets stuck while executing core steps, *e.g.,* divides by zero. Hence it is only sound to remove a thread from the schedule if it does not get stuck and does not reach an external call.

How do we relate a FineConc execution to a CPM execution? When all the FineConc threads are atExternal, it is simple to find a CPM schedule resulting in equivalent thread/memory states. When some are not atExternal, our simulation relation keeps track of how many (internal) steps each thread has made since the last atExternal state.

Definition 7.1.2. $\langle(\vec{s}, \vec{\pi}, L), m\rangle \overset{xs}{\sim} \left\langle(\vec{s}', \vec{\pi}', L'), m'\right\rangle$ *is a* simulation relation *between two states, annotated with a list of thread identifiers xs (representing extra steps taken by the FineConc machine), iff: For every thread i, there exists a state* $\langle(\vec{s}_2, \vec{\pi}_2, L), m_2\rangle$ *s.t.:*

1. $\langle xs|_i, (\vec{s}, \vec{\pi}, L), m\rangle \mapsto^* \langle nil, (\vec{s}_2, \vec{\pi}_2, L_2), m_2\rangle$
2. $\vec{s}'(i) = \vec{s}_2(i)$, $\vec{\pi}'(i) = \vec{\pi}_2(i)$, *and at any address a where* $\vec{\pi}'(i)$ *gives read permission,* $m'(a) = m_2(a)$.

(We write $xs|_i$ *to mean the subsequence of xs elements that are equal to i.)*

By establishing such a relation between the two machines we can prove that safety of the CPM implies safety of the internal and administrative steps steps (START, RESUME, CORE, SUSPEND, STUTTER) of the FineConc machine.

Lemma 7.1.3. *Suppose* $\langle(\vec{s}_c, \vec{\pi}_c, L_c), m_c\rangle \overset{xs}{\sim} \langle(\vec{s}_f, \vec{\pi}_f, L_f), m_f\rangle$*, that is, CPM state* S_c *simulates FineConc state* S_f *modulo some extra fine steps xs. Suppose* S_c *is safe for all schedules:*
$\forall \mho_c,\ \Psi \vdash_{\mathrm{CPM}} \mathrm{safe}\ \langle \mho_c, (\vec{s}_c, \vec{\pi}_c, L_c), m_c\rangle$*. Then for any thread i and schedule* $\mho_f$ *such that* $\vec{s}_f(i)$ *is at an internal step there exists a state* $\left\langle(\vec{s}_f', \vec{\pi}_f', L_f'), m_f'\right\rangle$ *such that:*

1. $\langle i \cdot \mho_f, (\vec{s}_f, \vec{\pi}_f, L_f), m_f\rangle \mapsto \left\langle \mho_f, (\vec{s}_f', \vec{\pi}_f', L_f'), m_f'\right\rangle$
2. $\langle(\vec{s}_c, \vec{\pi}_c, L_c), m_c\rangle \overset{i \cdot xs}{\sim} \left\langle(\vec{s}_f', \vec{\pi}_f', L_f'), m_f'\right\rangle$

Proof. Instantiate the CPM schedule as $\mho_{\mathrm{c}}=[i]$; the conclusion follows from definition 7.1.2 with determinism of internal steps. □

lemma 7.1.3 demonstrates that in order to prove safety of internal steps for the FineConc machine it suffices to only relate the state of individual threads with the ones of an equivalent execution in the CPM machine. However, that is not enough to prove safety of synchronization steps. Synchronization steps may consult the lock pool; will lock pools be equivalent between CPM and FineConc executions? Synchronization steps require guesses δ of what permissions to transfer; will the CPM δ's work in the FineConc machine?

Lock pools are equivalent between the two machines, since the extra FineConc internal steps do not change the lock pool. The CPM δ's do indeed work for FineConc, but since internal steps may free blocks, we must prove that these δ's still avoid incoherence between thread permissions.

Definition 7.1.4. *Let* $\langle(\vec{s}_{\mathrm{c}},\vec{\pi}_{\mathrm{c}},L_{\mathrm{c}}),m_{\mathrm{c}}\rangle$ *and* $\langle(\vec{s}_{\mathrm{f}},\vec{\pi}_{\mathrm{f}},L_{\mathrm{f}}),m_{\mathrm{f}}\rangle$ *be two states such that* $\langle(\vec{s}_{\mathrm{c}},\vec{\pi}_{\mathrm{c}},L_{\mathrm{c}}),m_{\mathrm{c}}\rangle \overset{xs}{\sim} \langle(\vec{s}_{\mathrm{f}},\vec{\pi}_{\mathrm{f}},L_{\mathrm{f}}),m_{\mathrm{f}}\rangle$ *for some list of thread identifiers* xs*. The states are* weakly equivalent *iff for every thread* i*,* $\vec{\pi}_{\mathrm{c}}(i) \geq \vec{\pi}_{\mathrm{f}}(i)$*.*

The $\vec{\pi}_{\mathrm{c}} \geq \vec{\pi}_{\mathrm{f}}$ lets the extra FineConc steps free some blocks.

Theorem 7.1.5 (Interleaved safety, special case)**.**
Let Ψ *be an assembly-language program* ***with no internal function calls****, with initial state* $\langle\sigma_0,m_0\rangle$*. Let* π_0 *be the max permissions of* m_0*. Assume* $\forall n,\mho_{\mathrm{c}}.\ \Psi \vdash_{\mathrm{CPM}} \mathrm{safe}_n\ \langle\mho_{\mathrm{c}},([\langle\mathrm{Run},\sigma_0\rangle],[\pi_0],\{\}),m_0\rangle$.
Then $\forall\mho_{\mathrm{f}}.\ \Psi \vdash_{\mathrm{FineConc}} \mathrm{safe}_{|\mho_{\mathrm{f}}|}\ \langle\mho_{\mathrm{f}},([\langle\mathrm{Run},\sigma_0\rangle],[\pi_0],\{\}),m_0\rangle$.

Proof. We must prove that for any possible preemptive schedule, the FineConc machine is safe. How do we know that those extra internal steps are safe (e.g., don't access memory outside the thread's current π footprint)?

$$\begin{tikzcd}
\langle s_c, m_c\rangle \arrow[d, dash, "xs"'] \arrow[dr, dashed, dash, "i \cdot xs"] & \\
\langle s_f, m_f\rangle \arrow[r, dashed, "i"] & \langle s'_f, m'_f\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
\langle s_c, m_c\rangle \arrow[r, dashed, "i"] \arrow[d, dash, "xs"'] \arrow[dr, phantom, "xs - i"] & \langle s'_c, m'_c\rangle \arrow[d, dashed, dash] \\
\langle s_f, m_f\rangle \arrow[r, dashed, "i"] & \langle s'_f, m'_f\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
\langle s_c, m_c\rangle \arrow[r, dashed, "i"] \arrow[d, dash] \arrow[dr, phantom, "i \notin xs"] & \langle s'_c, m'_c\rangle \arrow[d, dashed, dash, "xs"] \\
\langle s_f, m_f\rangle \arrow[r, dashed, "i"] & \langle s'_f, m'_f\rangle
\end{tikzcd}$$

Internal Step   Suspend Step   External Step

Each step on schedule $\mho_\mathrm{f}$ is an internal, suspend, or external step. If internal, use lemma 7.1.3 to show that this step is safe and simulation is retained. If SUSPEND, then by the diagram both machines step and simulation is retained. If external, then this thread is suspended by a previous SUSPEND step and hence its thread id cannot be in the list of extra steps, and the third diagram applies. □

Renaming stack frames. Now we remove the "no internal function calls" restriction.

CompCert's handling of "fresh" blocks makes the interleaving proof more complicated. Whenever any CompCert language (Clight, x86) needs a fresh block for a stack-allocated variable or an entire stack frame, it increments the *nextblock* component of the memory m. When showing a relation between differently interleaved executions (e.g., cooperative vs. preemptive), one must "α-convert" the block-numbers. Fortunately, all the CompCert languages have semantics that are invariant under such renaming; for example:

Lemma 7.1.6. *Execution of x86 Asm is invariant under permutation of block-numbers (between any computation steps).*

Theorem 7.1.7 (Interleaved safety, general case)**.** *Let* Ψ *be an assembly-language program, with initial state* $\langle \sigma_0, m_0\rangle$*. Let* π_0 *be the max permissions of* m_0*.*
Assume that $\forall n, \mho_\mathrm{c}.\ \Psi \vdash_{\mathrm{CPM}} \mathrm{safe}_n\ \langle \mho_\mathrm{c}, ([\langle \mathrm{Run}, \sigma_0\rangle], [\pi_0], \{\}), m_0\rangle$*.*
Then $\Psi \vdash_{\mathrm{FineConc}} \mathrm{safe}_{|\mho_\mathrm{f}|}\ \langle \mho_\mathrm{f}, ([\langle \mathrm{Run}, \sigma_0\rangle], [\pi_0], \{\}), m_0\rangle$*.*

Proof. The supplementary documents contain definitions of equivalence-modulo-renaming that lead to this proof. It is like the proof of theorem 7.1.5, but wherever "equivalence" of memories or thread-states is used, use equivalence modulo renaming of blocks. □

7.2 Well synchronized programs

Correctness of sequentially consistent executions does not *normally* guarantee correctness on actual processors due to weaker consistency guarantees. But it does so for *well-synchronized* programs that make proper use of synchronization mechanisms to ensure race-freedom between nonatomic accesses. For our instance of the CPM with x86 semantics, we adopt a slight modification of Owens's [29] notion of spinlock well-synchronized for TSO, which is the memory model actually used by x86 processors. In particular, Owens proved:

Definition 7.2.1 (memorySC). *A program is* memory sequentially consistent (memorySC) *iff for each of its possible executions on x86-TSO, there exists a memory equivalent execution on x86-SC [29, Definition 4]. Two execution traces are* memory equivalent *iff they have the same subsequence of writes to shared memory, and there exists a bijection between the read events of each trace such that corresponding read events read from the same write event. x86-TSO is the instruction set architecture supported by Intel and AMD; x86-SC is a sequentially consistent architecture obtained by removing all of the write buffers.*

Definition 7.2.2 (spinlock well-synchronized). *A program is* spinlock well-synchronized *w.r.t. a particular spinlock implementation iff for every x86-SC execution, and for every pair of competing events that are not on a spinlock, there is a spinlock that is released and then acquired between them. [29, Definition 7]*

Theorem 7.2.3 (Owens Theorem 2). *If an x86 program is spinlock well-synchronized and the locations of spinlocks are only accessed by the spinlock code, then it is memorySC. [29, Theorem 2]*

We generalize "spinlocks are only accessed by the spinlock code" to account for Makelock and Freelock:

Definition 7.2.4 (Spinlock clean)**.** *An event trace ϵ is* spinlock clean *iff:*

- *for all $i < j$ such that ϵ_i is* Makelock(a) *and there is no $i < u < j$ such that ϵ_u is* Freelock(a)*, then ϵ_j is not* Read(a) *or* Write(a)*. That is, between* Makelock(a) *and* Freelock(a) *there are no (nonatomic) Reads or Writes to a.*

- *forall i such that ϵ_i is* Acquire(a) *or* Release *there is $j < i$ such that $\epsilon_j =$* Makelock(a) *and there is no $j < u < i$ such that $\epsilon_u =$* Freelock(a).

Conjecture 7.2.5. *If an x86 program is spinlock well-synchronized and spinlock clean, then it is memorySC.*

This is a straightforward extension of Owens's theorem.

Theorem 7.2.6 (FineConc is synchronized and clean)**.**
Suppose $\Psi \vdash_{\text{FineConc}} \langle \mho_{\text{f}}, ([\langle \text{Run}, \sigma_0 \rangle], [\pi_0], \{\}, m_0 \rangle \overset{\epsilon}{\mapsto} S$, that is, an interleaved execution executes with event trace ϵ. Then ϵ is spinlock well-synchronized and spinlock clean.

Proof. Proved in Coq, and in LaTeX: see the supplement. □

Owens uses the x86 spinlock from Linux v2.6.24.7 [29, Fig. 2]. For "ticketed spinlocks" Owens proves a result (his Theorem 3) that is weaker: it requires the thread releasing a lock to be the same one that acquired it. Thus it prohibits daring concurrency; we cannot use ticketed spinlocks to implement our model.

While Owens's theorem only addresses x86-TSO, we expect that similar theorems exist for all other major architectures: indeed, "race-free programs have sequentially consistent behavior" is generally considered a necessary correctness condition for weak-memory processors [2].

7.3 Erasing the permissions, at last

Our Concurrent Erased Machine is like our fully interleaved machine ($\vdash_{\text{FineConc}}$), but without the thread permissions $\vec{\pi}$, lock permissions L, or guesses δ. In the memories m, all permissions are reset to $\top$ (Freeable) at each step. This imitates a conventional sequentially consistent machine, but it can be conveniently formulated on top of CompCert's sequential semantics.

We can define an erasure from states S of FineConc to states $\hat{S}$ of the CEM, by removing $\vec{\pi}$ and L.

Theorem 7.3.1 (Erasing the permissions).
Suppose a program executes, $\Psi \vdash_{\text{FineConc}} \langle \mho_{\text{f}}, ([\langle \text{Run}, \sigma_0\rangle], [\pi_0], \{\}, m_0\rangle \overset{\epsilon}{\mapsto} S$. *Then* Ψ *executes in the Concurrent Erased Machine* with the same trace ϵ, $\Psi \vdash_{\text{CEM}} \langle \mho_{\text{f}}, ([\langle \text{Run}, \sigma_0\rangle], \{\}, m_0\rangle \overset{\epsilon}{\mapsto} \hat{S}$.

Proof. As usual in an erasure theorem, permissions π do not affect *what* results are computed in FineConc, only whether the execution gets stuck. The trace ϵ records what memory operations have taken place. □

Claim 7.3.2 (x86-SC). *The Concurrent Erased Machine is exactly a model of what Owens calls* x86-SC.

Proof. Self-evident. To prove it, one must compare the CEM instantiated with CompCert's definition of x86 assembly language, with Owens's x86-SC model. □

Chapter 8

Conclusion

This work presents a powerful semantics for concurrent programs called the Concurrent Permission Machine. It enables a top-to-bottom proof of correctness accross the soundness of the Concurrent Separation Logic, a proven correct optimizing compiler, and an assembly program that runs correctly in a realistic model of a machine with weakly consistent cashes. All of this is done with machine checked proofs in Coq. The CPM uses the notions of permission, already existing in CompCert, and the simple notion of *coherence* that has the same guarantees of data-race freedom, while allowing for local reasoning.

We present the proof of correctness of a real optimizing compiler that supports programs with concurrency. The proof leverages an existing proof of correctness that only supports sequential programs and, with minimal changes, derives a proof that supports concurrency. We compose this compiler correctness proof with the soundness of the CSL, the interleaving proof, and the synchronization proof to produce a top-to-bottom proof of correctness. The proof ensures the correct execution of a compiled program whose source has been proven correct in a machine like x86.

8.1 Future Work

8.1.1 Short-term work

Removing statically checked properties. In section 3.1.2 we use statically checked validation for some properties of the assembly language program. We know that all the properties in static_validation can be derived from the CompCert compiler, but we leave these proofs for future work.

Relaxing full injections. In section 4.5.3, we explain why the current work assumes that simulations preserve full_injection. We propose to lift this restriction and support the compiler passes that break this assumption.

We need to enhance CompCert's specification of external functions, allowing them to modify only locations reachable from *all* escaped addresses. All those escaped locations can be kept in a history of arguments to external functions args_hist. Then we would prove that args_hist are not unmapped by SimplLocals or Unusedglob, since it only contains escaping pointers and these passes delete variables that do not escape and unused global variables.

There is one more wrinkle to supporting Unusedglob, that we can easily address in future work. The current Compile One At a Time (COAT) technique starts by compiling the first thread. However the first thread is the one that allocates the global variables, so if it deletes some of these global variables the self-simulation of other threads will not work. To solve this problem, we only have to observe that allocating more global variables is supported by self-simulation, and thus propose to extend the COAT technique to compile all threads beyond the second one and then compile the first thread last. This way, when the first thread is compiled, none of the other threads depend on the global variables that will be deleted. That should be enough to support Unusedglob.

Simpler for separate compilation. We suggested a new approach to separate compilation using our Compile One At a Time Technique, in section 5.2. We leave the Coq implementation and the connection to the present development for future work.

8.1.2 Long-term work

Support for fine-grained synchronizations. We intend to extend the work to other synchronization models: atomic operations in the relaxed, release-acquire, and SC modes of C11. We will model them as "external calls" in CompCert's model even though the compiler may actually inline them at the very end. Because the sequential compiled code treats these atomics as external, adding these operations does not change the compiler in any way. To handle SC atomics alone, the CPM could be extended with rules similar to those for locks, since an SC access to a location can be modeled by acquiring a lock on the location, changing its value, and then releasing the lock. Release-acquire and relaxed accesses will require storing additional information, such as histories of accesses and associated permission maps for each atomic location, using an approach analogous to Kaiser *et al.* [20] or Kang *et al.* [21].

Support for other builtins and external functions. Our framework can easily be extended to support silent external functions and builtins that do not communicate amongst threads. However, some system calls and external functions can be used to synchronize executions (e.g., treating a system call read/write as a lock) or to expose the order of execution (e.g., output that depends on the order of execution). To avoid races, the use of shared resources should be properly synchronized (e.g., using locks). We can extend our permissions to cover non-memory resources (such as external sensors) to avoid races. Such extension should be mirrored by a protocol in the

Concurrent Separation Logic, that proves the correctness of other external functions and is sound with respect to this resource permissions.

Support for other source level logics. Our semantics is general enough that we expect it to be compatible with other logics. We are particularly interested in exploring a connection to logics that support more relaxed atomics such as the logics of Kang *et al.* [21] or IRIS [19].

Support for other architectures. Our current development targets x86 TSO, particularly relying on Giannarakis's reduction theorem (section 7.2) for this architecture. We expect that similar reduction theorems exists for other architectures.

Appendix A

Implementation Details

This appendix contains the Coq implementation of a selected number of definitions, lemmas and proofs. It is intended as a companion for both the tech report (to fill in implementation details that might be interesting to the reader) and to the Coq code (to have more complete explanation of the code, than what fits in code comments). The pieces of code are organized by the chapters as they are defined in the tech report.

A.1 Obtaining and compiling the Coq code

To obtain the code. Clone `https://github.com/PrincetonUniversity/VST` into a local folder /VST and switch to branch *concurrency*. If you want to follow along with the code here, switch to commit ff36d5e8d7457e304b8d14e5cfad72a68fa39ee7.

To compile the code. In the /VST directory, make a file named CONFIGURE containing exactly the text:

```
COMPCERT=compcert_new
```

Then, navigate to the folder /VST, and run

```
make concurrency
```

or, if you have a multicore computer, `make concurrency -j`.

A.2 Useful definitions

Contains Thread

We say that a CPM contains a thread i, if its number of threads is lower than the thread index i. Using this predicate is particularly useful since it allows us to avoid having option types, for every function that gets permission maps (i.e., $\vec{\pi}[i]$), or states (i.e., $\vec{s}[i]$) from the CPM state. We use the notation $i \in$ tp to say containsThread tp i. We will use the notation

```
Definition containsThread (tp : cpm) (i : tid) : ℙ := i < num_threads tp.
```

Mem compatible

We carry a notion of compatibility with memory ensuring that memories restricted to some permission map ($m|_\pi$) are well formed.

```
Record mem_compatible (tp : cpm) (m : mem) : ℙ :=
```

Well formed memories must satisfy that the current permissions are at most the max permissions. We require that the permission of each thread is lower than the max permissions in the memory.

```
  { compat_th : ∀(i : nat) (cnt : i ∈ tp),
                     permMapLt (getThreadR cnt).1 (getMaxPerm m) ∧
                     permMapLt (getThreadR cnt).1 (getMaxPerm m);
```

Similarly, we require that the permission of each lock be lower than the max permissions in the memory.

```
    compat_lp : ∀(l : address) (pmaps : permission_map),
                     ThreadPool.lockRes tp l = Some pmaps →
                     permMapLt pmaps.1 (getMaxPerm m) ∧
                     permMapLt pmaps.2 (getMaxPerm m);
```

All locations noted as a lock by the lock pool L, must have been allocated in the memory (i.e., lower than nextblock)

```
lockRes_blocks : ∀ (l : address) (rmap : permission_map),
                          lockRes tp l = Some rmap → valid_block m (fst l) }
```

A.3 The concurrent permission machine

A.3.1 Formal definition of the CPM

Permission coherence

The definition of Coherence is described in section 3.2.2 and the full code can be found in *VST/concurrency/common/HybridMachine.v* in line 91. We use the notation ∈, to mean containsThread, defined above in appendix A.2. We use the predicate permMapsDisjoint2, which says that two permissions are not competing (definition 3.2.1) and lock_coherence which says the permissions are data-lock coherent (definition 3.2.2). The definition of coherence, is an invariant preserved by the execution of CPMs, so we call it invariant:

```
Record invariant (tp: thread_pool) :=
```

The permissions of any two threads (both permissions π_1 and π_2), are not competing

```
  { no_race_thr : ∀ i j (cnti: i ∈ tp) (cntj: j ∈ tp) (Hneq: i ≠j),
          permMapsDisjoint2 (getThreadR cnti) (getThreadR cntj);
```

The permissions protected by any two unlocked locks are not competing

```
    no_race_locks:
      ∀ laddr1 laddr2 rmap1 rmap2
        (Hneq: laddr1 ≠laddr2)
        (Hres1: lockRes tp laddr1 = Some rmap1)
```

```
         (Hres2: lockRes tp laddr2 = Some rmap2),
         permMapsDisjoint2 rmap1 rmap2;
```

The of permissions any thread and that of am unlocked lock are not competing

```
     no_race_locks_thr: ∀ i laddr (cnti: i ∈ tp) rmap
         (Hres: lockRes tp laddr = Some rmap),
         permMapsDisjoint2 (getThreadR cnti) rmap;
```

For a thread, the locations it sees as regular location (i.e., have nonempty permissions in π_1) are not seen as locks by it is or the permissions in locks

```
     thread_data_lock_coh: ∀ i j (cnti: i ∈ tp) (cntj: i ∈ tp),
             lock_coherence (getThreadR cntj).1 (getThreadR cnti).2) ∧
         (∀ laddr rmap, lockRes tp laddr = Some rmap →
             lock_coherence rmap.1 (getThreadR cnti).2);
```

For an unlocked lock, the locations it sees as regular location (i.e., have nonempty permissions in π_1) are not seen as locks by any other thread or any other lock

```
     locks_data_lock_coh: ∀ laddr rmap (Hres: lockRes tp laddr = Some rmap),
         (∀ j (cntj: j ∈ tp), lock_coherence (getThreadR cntj).1 rmap.2) ∧
         (∀ laddr' rmap', lockRes tp laddr' = Some rmap' →
             lock_coherence rmap'.1 rmap.2);
```

Finally, and this is a small technical detail, for every lock created we protect a number of addresses that cannot be written. This simulates some libraries, such as Pthreads that store more information in a lock than just its state. The definition is parametric on the size of the locks and for a lock of size 1, the following proposition is trivial. lr_valid protects the lock, by ensuring that all the permissions are empty for the parts of the lock that do not contain the state (and thus should only be accessed by the library).

```
        lockRes_valid: lr_valid (lockRes tp)}.
```

A.4 MOIST simulations and semantics of CompCert

A.4.1 Definitions for MOIST simulations

Here we present and explain the Coq definition for inj_sim. The definitions takes two languages L_1 and L_2 with states state L_1 and stateL_2 respectively.

```
Record fsim_properties_inj: Type :=
```

The simulation uses measure Injindex with a well founded order to allow for a finite amount of stuttering in the target. This is taken from CompCert.

```
{ Injindex: Type;
  Injorder: Injindex → Injindex → ℙ ;
  Injfsim_order_wf: well_founded Injorder;
```

The simulation also uses a match relation (denoted ∼), to relate states in the source and target. We also require that the match relation imply an injection of the memories (using get_mem). We have also defined a similar simulation for extension phases (extend_sim) that only requires an extension and supports equality phases. CompComp [33] has demonstrated that we can weaken this requirement to only hold when the states are at_external; we plan to do so as future work.

```
  Injmatch_states: Injindex → meminj → state L1 → state L2 → ℙ ;
  Injfsim_match_meminj: ∀ i f s1 s2, Injmatch_states i f s1 s2 →
                                  Mem.inject f (get_mem1 s1) (get_mem2 s2);
```

As we explain in section 4.5.3, we currently require the injections not to drop any block. We will address this too in future work.

```
Injfsim_match_full: ∀i f s1 s2, Injmatch_states i f s1 s2 → injection_full f (get_mem1 s1);
```

The simulation must hold at initial states as depicted in fig. 4.18

```
Injfsim_match_entry_points:
  ∀s1 m0 fb args, entry_point $L_1$ m0 s1 fb args →
        ∃i f s2, entry_point $L_2$ m0 s2 fb args ∧ Injmatch_states i f s1 s2;
```

The simulation must also relate final states

```
Injfsim_match_final_states:
  ∀i s1 s2 r f,
    Injmatch_states i f s1 s2 → final_state $L_1$ s1 r → (final_state $L_2$ s2 r);
```

The simulation of internal steps follows exactly like in CompCert, except with injectable traces, as depicted in fig. 4.16

```
Injfsim_simulation:
  ∀s1 t s1' f, Step $L_1$ s1 t s1' →
                  ∀i s2, Injmatch_states i f s1 s2 →
                          ∃i', ∃s2' f' t',
                              (Plus $L_2$ s2 t' s2' ∨ (Star $L_2$ s2 t' s2' ∧ Injorder i' i))
                              ∧ Injmatch_states i' f' s1' s2' ∧
                              Values.inject_incr f f' ∧
                              inject_trace_strong f' t t';
```

The at_external property is preserved by the match relation and has a *lockstep* diagram, as depicted in fig. 4.19. The code of simulation_atx_inj is presented bellow.

```
Injsim_simulation_atx:
  simulation_atx_inj Injmatch_states;
Injsim_atx:
    preserves_atx_inj Injmatch_states;
```

The public global definitions (such as public functions) are preserved by compilation.

```
  Injfsim_public_preserved:
    ∀ id, Senv.public_symbol (symbolenv $L_2$) id = Senv.public_symbol (symbolenv $L_1$) id
}.
```

External step diagram (fig. 4.19).

The following diagram follows exactly like the internal step diagram in fsim_properties_inj, except for line 4 (which ensures that the state is at_extenral) and line 8 (which that requires exactly one step from the source).

```
1      Definition simulation_atx_inj {index:Type} {$L_1$ $L_2$}
2                  (match_states: index → meminj → state $L_1$ → state $L_2$ → ℙ ) :=
3         ∀ s1 f args,
4           at_external $L_1$ s1 = Some (f,args) →
5           ∀ t s1', Step $L_1$ s1 t s1' →
6                    ∀ i j s2, match_states i j s1 s2 →
7                             ∃ i', ∃ s2' j' t',
8                                 Step $L_2$ s2 t' s2' ∧
9                                 match_states i' j' s1' s2' ∧
10                                Values.inject_incr j j' ∧
11                                inject_trace_strong j' t t'.
```

A.5 Compiler correctness

A.5.1 COAT proof of compiler correctness

Match relation for Hybrid Machine states.

We explain the Coq code for the match relation, $\gtrsim$, of Hybrid Machine states as defined in definition 5.2.4. The full code can be found in *VST/concurrency/compiler/concur_match.v*. The relation takes a hybrid bound hb, a well-founded measure ocs, a memory injection j, two memories m_1, m_2, and the rest of the states of each cstate1 and cstate2.

```
Record concur_match hb (ocd: option compiler_index)(j: meminj)
          (cstate1: HM_hb) (m1: mem)
          (cstate2: HM_hb+1) (m2: mem):=
```

The states have the same number of threads

```
    { same_length: num_threads cstate1 = num_threads cstate2
```

As explained in section 4.5.3, we currently require that injections be full. This will be addressed in future work.

```
      ; full_inj: Events.injection_full j m1 (* to be removed *)
```

Both states are compatible with the memory. The definition of mem_compatible is explained in appendix A.2

```
      ; memcompat1: mem_compatible cstate1 m1
      ; memcompat2: mem_compatible cstate2 m2
```

Each thread's lock permissions π_2 is mapped in the following sense: if the thread has some (non-empty) permissions in the target (Σ_2), then in the source (Σ_1) the same thread has the same permission in the same location (up to injection).

```
      ; lock_perm_preimage:
          ∀ i (cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2),
            perm_surj j (lock_perms cnt1) (lock_perms cnt2)
```

For each thread, the memories restricted to the thread's permissions inject: $m_1|_{\pi^1_{1,i}} \xhookrightarrow{j} m_2|_{\pi^2_{2,i}}$. The extra arguments Hlt1 Hlt2 ensure that the memories restricted to the thread's permissions are well formed. Both arguments can be derived from the memory compatibility described above

```
      ; INJ_threads:
          ∀ i (cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2) Hlt1 Hlt2,
            Mem.inject j (@restrPermMap (getThreadR cnt1).1 m1 Hlt1)
                         (@restrPermMap (getThreadR cnt2).1 m2 Hlt2)
```

Similarly, for each thread, the memories restricted to the thread's lock permissions inject, $m_1|_{\pi^2_{1,i}} \xhookrightarrow{j} m_2|_{\pi^2_{2,i}}$.

```
      ; INJ_locks:
          ∀ i (cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2) Hlt1 Hlt2,
            Mem.inject j (@restrPermMap (getThreadR cnt1).2 m1 Hlt1)
                         (@restrPermMap (getThreadR cnt2).2 m2 Hlt2)
```

Also, for each lock, the memories restricted to the lock permissions inject (notice the locks are not the same address, but are related by the injection).

```
      ; INJ_lock_permissions:
          ∀ b b' delt opt_rmap,
            j b = Some (b', delt) →
            ∀ ofs, lockRes cstate1 (b, unsigned ofs) = opt_rmap →
                   lockRes cstate2 (b', unsigned (add ofs (repr delt))) =
                   (option_map (virtueLP_inject m2 j) opt_rmap)
```

And all locks are injected.

```
    ; INJ_lock_content:
        ∀ b ofs rmap,
          lockRes cstate1 (b, ofs) = Some rmap →
          inject_lock j b ofs m1 m2
```

Both states are coherent, as defined in appendix A.3.1 (code explained above in appendix A.3.1) .

```
    ; source_invariant: invariant cstate1
    ; target_invariant: invariant cstate2
```

Finally, all threads are in a match relation from source to target. For i > hb we use the match relation for Clight given by the self simulation by lemma 5.3.2 and, similarly, for i < hb we use the match relation for Assmebly given in lemma 5.3.2. For i = hb, we use the match relation given by the MOIST simulation of CompCert

```
    ; mtch_source: ∀ (i:nat), (i > hb) →
          ∀ (cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2) Hlt1 Hlt2,
            match_thread_source j (getThreadC cnt1)
                                  (@restrPermMap (getThreadR cnt1).1 m1 Hlt1)
                                  (getThreadC cnt2)
                                  (@restrPermMap (getThreadR cnt2).1 m2 Hlt2)
    ; mtch_target:
        ∀ (i:nat), (i < hb) →
          ∀ (cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2) Hlt1 Hlt2,
            match_thread_target j (getThreadC cnt1)
                                  (@restrPermMap (getThreadR cnt1).1 m1 Hlt1)
                                  (getThreadC cnt2)
                                  (@restrPermMap (getThreadR cnt2).1 m2 Hlt2)
  ; mtch_compiled:
          ∀(i:nat), (i = hb) →
            ∀(cnt1: i ∈ cstate1) (cnt2: i ∈ cstate2) Hlt1 Hlt2,
              match_thread_compiled ocd j (getThreadC cnt1)
                                           (@restrPermMap (getThreadR cnt1).1 m1 Hlt1)
                                           (getThreadC cnt2)
                                           (@restrPermMap (getThreadR cnt2).1 m2 Hlt2) }.
```

A.5.2 Simulations for synchronizations

Diagram for synchronization steps

We present bellow the Coq specification of a diagram for synchronization steps, as defined in lemma 5.5.1. Notice that we separate the memory from the rest of the CPM state in the implementation:

```
Lemma external_step_diagram:
  ∀(U : schedule) (cd : option compiler_index)(j : meminj)
    (st1 : cpm)(m1 : mem)(st1' : cpm)(m1' : mem) (st2 : cpm) (m2 : mem)
    (ev1 : sync_event),
```

The initial states are in a match relation, $(\mathrm{st}_1, m_1) \gtrsim_j (\mathrm{st}_2, m_2)$. The condition Hcmpt is redundant with the match relation, but we use it to construct the well formed memories.

```
    ∀(Hcmpt : mem_compatible st1 m1),
    concur_match cd j st1 m1 st2 m2 →
```

The thread i is at the top of the shedule

```
    ∀(i:tid) (cnt1 : i ∈ st1),
      schedPeek U = Some i →
```

The thread i can take a synchronization step $\Psi, \Phi \vdash_{\mathrm{HM}} \langle i \cdot \mho, \mathrm{st}_1, m_1\rangle \overset{\mathrm{ev}_1}{\mapsto} \langle \mho, \mathrm{st}_1', m_1'\rangle$

```
        syncStep true cnt1 Hcmpt st1' m1' ev1 →
```

Then we can construct, in the target HM, a new event ev_2, a new state st_2', measure cd and memory injection j.

```
        ∃ev2 (st2' : cpm) (m2' : mem) (cd' : option compiler_index) (j' : meminj),
```

The new states are in a match relation $(\mathrm{st}_1', m_1') \gtrsim_j (\mathrm{st}_2', m_2')$.

```
          concur_match cd' j' st1' m1' st2' m2' ∧
```

the produced events inject, $\mathrm{ev}_1 \overset{j}{\hookrightarrow\!\!\!\!\twoheadrightarrow} \mathrm{ev}_2$,

```
          inject_mevent j' (external i ev1) (external i ev2) ∧
```

Finally, the target machine can take a synchronization step $\Psi, \Phi \vdash_{\mathrm{HM}} \langle i \cdot \mho, \mathrm{st}_2, m_2\rangle \overset{\mathrm{ev}_2}{\longmapsto} \left\langle \mho, \mathrm{st}_2', m_2' \right\rangle$

```
          external_step U st2 m2 (schedSkip U)
            (external i ev2) st2' m2'.
```

Bibliography

[1] IEEE Standard for Information Technology-Portable Operating System Interface (POSIX(R)) base specifications, issue 7, in IEEE std 1003.1-2017 (revision of IEEE std 1003.1-2008).

[2] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[3] 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.

[4] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th Annual Symposium on Principles of Programming Languages (POPL'07)*, pages 109–122, January 2007.

[5] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge, 2014. 2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS Distinguished Dissertation Competition winner.

[6] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium of Programming*, Lecture Notes in Computer Science, pages 107–127. Springer, 2014.

[7] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, 2005.

[8] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reasoning*, 61(1-4):367–422, 2018.

[9] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: a practical system for verifying concurrent C. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), LNCS 5674*, 2009.

[10] Patterson Daniel and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *International Conference on Functional Programming. no:85*, 2019.

[11] Robert W. Dockins. *Operational refinement for compiler correctness.* PhD thesis, Princeton University, July 2012.

[12] Mike Dodds, Mark Batty, and Alexey Gotsman. Compositional verification of compiler optimisations on relaxed memory. In *European Symposium on Programming*, pages 1027–1055. Springer, 2018.

[13] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, 2007.

[14] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *POPL*, pages 595–608. ACM SIGPLAN, 2015.

[15] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, pages 646–661, June 2018.

[16] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, pages 353 – 367, 2008.

[17] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *Proc. 37th Annual ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 171–185, January 2010.

[18] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. Towards certified separate compilation for concurrent programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.*, 2019.

[19] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 637–650. ACM, 2015.

[20] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. *In 31st European Conference on Object-Oriented Programming (ECOOP 2017)*, 2017.

[21] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *Proceedings of the 44th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages - POPL 17*, 29, 2017.

[22] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, 2016.

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

[24] Xavier Leroy. The CompCert verified compiler, software and annotated proof, March 2019.

[25] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In Appel [3], chapter 32.

[26] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1), 2008.

[27] William Mansky, Andrew W. Appel, and Aleksey Nogin. A verified messaging system. *Proc. ACM Program. Lang.*, 1(OOPSLA):87:1–87:28, October 2017.

[28] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1):271–307, May 2007.

[29] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP 2010: 24th European Conference on Object-Oriented Programming*, pages 478–503. Springer, 2010.

[30] Anton V. Podkopaev, Ori Lahav, and Viktor Vafeiadis. Promising compilation to ARMv8.3. *Proceedings of the Institute for System Programming of the RAS*, 29(5):149–164, 2017.

[31] Anton V. Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages 3, no. POPL*, 2019.

[32] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. In *Journal of the ACM (JACM)*, page 60(3), 2013.

[33] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. In *POPL*, volume 50, pages 275–287. ACM, 2015.

[34] James Gordon Stewart. *Verified Separate Compilation for C*. PhD thesis, Princeton University, Princeton, NJ, June 2015.