Concurrent Permission Machine for modular proofs of optimizing compilers with shared memory concurrency.

Santiago Cuellar

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

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Andrew Appel

subm. Date

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 high-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, I show how to reuse the (sequential) CompCert proof, without major changes, to guarantee a stronger concurrent-permission-aware notion of correctness.

Acknowledgements

Acknowledgements...

Dedication.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main ultimate goal of this thesis is to create a modular way to prove compilers safe, even in the presence of concurrency. There are already several compilers proven safe, with machine checked proofs, that give guarantees for sequential programs. Instead of reinventing the wheel, I intend to leverage those proofs and use it to give guarantees for concurrent programs.

On top of "just a compiler proof", our system is intended to be compatible with a realistic separation logic and usable on realistic machines like X86 TSO.

The main result:

Contributions: In this thesis I have....

Chapter 2

Related Work

Here I talk about related work:

2.1 Past work 1

Topic 1.

Chapter 3

Top to Bottom structure

Note.

3.1 Main theorem

Explaining the main theorem:

Assumptions:

```
Theorem top2bottom_correct:
  CSL_correct C_program →
  CompCert_compiler C_program = Some Asm_program →

  ∀(src_m:Memory.mem) (src_cpm:Clight.state),
    CSL_init_setup C_program src_m src_cpm →
    Clight.entry_point (Clight.globalenv C_program) src_m src_cpm (main_ptr C_program) nil →
    ∀(limited_builtins:Asm_core.safe_genv x86_context.X86Context.the_ge),
    asm_prog_well_formed Asm_program limited_builtins →

    ∀(U:schedule), ∃(tgt_m0 tgt_m: mem) (tgt_cpm:ThreadPool.t),
        permissionless_init_machine
          Asm_program limited_builtins
          tgt_m0 tgt_cpm tgt_m (main_ptr C_program) nil ∧
        spinlock_safe U tgt_cpm tgt_m.
```

- CSL_correct C_program: States that the program has been proven correct in CSL, as described in [section CSL ref]
- CompCert_compiler: States that the CompCert compiler translates C_program into the assembly program Asm_program.
- CSL_init_setup: States that src_m and src_cpm are the initial memory and state as defined by the program C_program... According to CSL
- Clight.entry_point : Same as above... According to Clight.
- limited_builtins: Statically checkable property stating that Asm_program doesn't have unsupported builtins.
- asm_prog_well_formed: Another statically checkable property stating that the initial memory and global environment created by Asm_program are well formed.

Conclusions:

- permissionless_init_machine: There exists initial memories and state for Asm_program
- spinlock_safe: The initial state is safe and "spinlock well synchronized" as defined in [reference to spinlock]

3.2 The concurrent permission machine

Describe CPM.

3.3 Limitations of the CPM model

What are the limitations? - That coin example from Ernie Cohen. - Angel transferring too many permissions proof.

Chapter 4

Memory Observable Injectable Startable Trace simulations of CompCert

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

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

Table 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).
```

Table 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).

First, CompCert can't give any guarantees about compiling the code in 4.1 because it is not a complete program. It is reasonable to expect that the function 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. In fact, the only possible initial states, are characterized by a predicate initial_state: state → ℙ that takes no additional arguments. You can see an instantiation of the predicate for Clight in 4.2. So, even though CompCert correctly compiles the code, it's 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 4.1 describes a system call and 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 in buff (which for system calls will not be in the program's accessible memory), so

[1] Leroy [5] claims that "inputs given to the programs are uniquely determined by their previous outputs", but this is not exactly correct. A more accurate representation would be to say "inputs given to the programs are uniquely determined by their last outputs". As we will see in 4.4, 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.

Table 4.3: The events in CompCert

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 4.3, can only contain integers, floats or pointers to global variables.

Moreover, in the CompCert semantics, the entire behavior of external functions 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, but it is not exposed in its specification.

Finally, the correctness of CompCert is stated as semantic preservation theorem, where the traces are the preserved behavior and the proof uses forward simulations[2]. At least two other works ([7], [4]) have proposed alternative simulations and made the simulations an exposed feature of the compiler's specification. In this papers, the authors view CompCert correctness modularly as a thread-local or module-local

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

simulation and 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.

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

Definition 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 external 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* injections*, 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.

	Percent change
Arguments in main	0
Injectable Traces	0
Semantics	0
Simulations	0
Total	x

Table 4.4: Percentage change to CompCert: changes are calculated from the number of lines added as given by running git diff between our version of CompCert and the master branch. For each feature, an estimated percent is provided.

- *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 they can construct a memory from their state with get_mem. In practice all CompCert languages use the same memory model, described in **??**, which we will refer as mem from now on. Nevertheless, a future language could use *juicy memory* as in [1] or abstract state in [3] 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 and at_external function and, finally, we show how to put everything together in MOIST simulations for CompCert.

The changes described here represent only a x% 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 4.4.

4.1 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 Table 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 need to generalize this to any function, not just main(). In this section we define a new predicate entry_point, generalizing initial_state (Table 4.2), that characterizes starting states which includes calls to main with arguments and calls to any other function.

Passing arguments to main is of particularly important for our work with concurrency because spawning new threads behaves very similar to starting a program by calling main: The library function that spawns a new thread (e.g. pthread_create) must create a new stack, push the arguments to stack and then call foo just like a program initialization would do. The predicate entry_point is general enough to capture these two types of preprocessing. In fact, the new predicate can describe executions starting with any kind of preprocessing 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 (Table 4.5(b)) in the same way that CompCert's simulation preserves initial_state (Table 4.5(a)). The proof largely follows the simulation of internal function calls which is already proven in CompCert, so we omit

$$\begin{tikzcd} {} \arrow[d, "\mathrm{initial\_state}"'] & & {} \arrow[d, dotted, "\mathrm{initial\_state}"] \\ s_1 \arrow[rr, phantom, "\sim"] & & s_2 \end{tikzcd}$$

(a)

$$\begin{tikzcd} {} \arrow[d, "\mathrm{entry\_points}\ m_0\ f\ \mathrm{args}"'] & & {} \arrow[d, dotted, "\mathrm{entry\_points}\ m_0\ f\ \mathrm{args}"] \\ s_1 \arrow[rr, phantom, "\sim"] & & s_2 \end{tikzcd}$$

(b)

Table 4.5: 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 $\sim$.

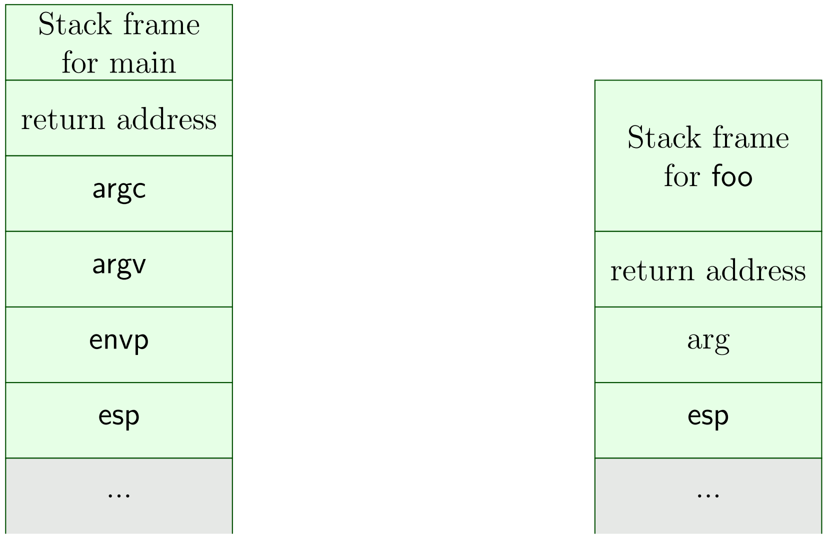
the details here. However, some interesting relevant details are presented later in subsection 4.1.2.

4.1.1 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 in 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 describe bellow.

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 (that are not passed in registers). We call *prestack* that tip of the stack, depicted in Table 4.6(a), which is relevant to the execution of main . The rest of the memory, at that point, also contains the NULL-terminated argument vector, all the global variables, and possibly stacks of other initialization functions such as start. We call the entire memory at this point the *initial memory*.

Our new predicate entry_point, instead of an empty stack, describes how arguments are set up in the prestack and, instead of an almost empty initial memory, allows memories to have arbitrary things. This extra contents of memory can be the argument vector, stacks of other functions, stacks of other threads, or anything else.



(a) Stack at the start of main .

(b) Stack created pthread_create() to start a thread running foo.

Table 4.6: Prestacks example for X86 in 32bit mode: examples of stacks created before the entry function executes. (a) Stack shape right before main executes. (b) Stack right before a function foo is executed in a new thread. Notice that in 64bit mode, the arguments will be passed in registers.

As mentioned before, spawning a thread behaves like executing main in many ways. For instance, the stack of a thread before the first function executes looks just like a prestack before calling main, as shown in Table 4.6(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 only we could pass all arguments on registers, we wouldn't need to reason about the prestack at all. Unfortunately, in architectures such as $x86$ in 32-bit mode, all arguments are passed on the stack. As the comments in the CompCert code put it "Snif!"[6]. 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 describe argument passing for entry functions in general, we must describe the prestack.

The characterization of the prestack is language dependent, and it will be described more carefully in the next subsection.

4.1.2 The entry_point: a more permisive starting state

The predicate entry_point: mem $\rightarrow$ state $\rightarrow$ val $\rightarrow$ list val $\rightarrow \mathbb{P}$ 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, but its divided in 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 don't create dangling pointers.[3]

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 Table 4.5(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 Table 4.7. Lines 4-6 ensure that the environment

[3] 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.

```
1  Inductive entry_point (ge:genv): mem → state → val → list val → ℙ:=
2  | initi_core: ∀ f fb m0 args targs,
3        let sg:= signature_of_type targs type_int32s cc_default in
4        type_of_fundef (Internal f) = Tfunction targs type_int32s cc_default →
5        Genv.find_funct_ptr ge fb = Some (Internal f) →
6        globals_not_fresh ge m0 →
7        Mem.mem_wd m0 →
8        Val.has_type_list args (typlist_of_typelist targs) →
9        vars_have_type (fn_vars f) targs →
10       vals_have_type args targs →
11       Mem.arg_well_formed args m0 →
12       bounded_args sg →
13       entry_point ge m0 (Callstate (Internal f) args (Kstop targs) m0).
```

Table 4.7: The entry_point predicate in Clight

is allocated in memory and 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.

The prestack, is implicitly determined by Kstop targs since, from the types of the arguments, we can determine the shape of the prestack. In Clight, the continuation describes the program's call stack with Kcall describing a stackframe and Kstop describing the end of the stack. In our version, Kstop targs describes the prestack instead, which is the last frame. Through compilation, Kcall gets translated to a predicate Stackframe that describes a stack frame. Similarly, Kstop will be translated to Prestack, a special kind of Stackframe, as depicted in 4.1.

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

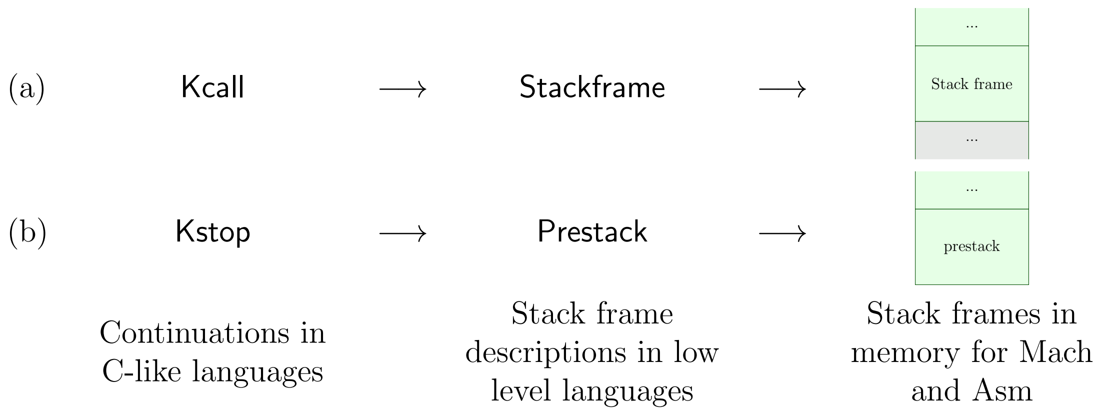


Figure 4.1: Abstractions of stack frames become more concrete through compilation. (a) Kcall is a high level abstractions of stackframes for C-like languages. It gets translated to Stackframes, which are low level descriptions of stack frames. Finally the stack is laid in memory. (b) Kstop, is a concise, high level, description of the the prestack. It gets compiled to Prestack, which is a special type of Stackframe, and finally laid in memory

none of their variables has 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 what arguments will be passed in memory and which ones in registers. Even for architectures that pass all arguments in memory, this is not done until the Stacking pass. So for languages 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 Table 4.7.

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, after the Stacking pass, it is translated to an actual prestack as shown in .

```
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    ⋮
```

Table 4.8: Part of the entry_point predicate in Mach

Machine languages

In the machine languages (Mach and Asm), a function expects certain shape from the stack frame of its caller. We replace the empty stack frame allocation above, with the construction of the prestack as shown in Table 4.8. 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 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.

4.2 Memory events.

The visible behavior of the CompCert semantics (for all languages) is a trace of events, as described in 4.3. It records interactions with the outside world; for example, the results of a read system call will record a Event_syscall together with the name of the system call, its parameters, and its result. The only pointers that can appear in the trace are locations of global variable. As we explained above, these events are insufficient to capture the behavior of incr in 4.1. The events are also not sufficient to express any kind of behavior that depends on the memory. For example, a call

```
Inductive event: Type :=
  ...
  | Event_acq_rel:
        list mem_effect →
        delta_perm_map →
        list mem_effect → event
  | Event_spawn:
        block →
        delta_perm_map →
        delta_perm_map → event.
```

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

Table 4.9: The new events in CompCert: mem_effect reflects changes to memory and delta_perm_map represents transfer of Cur permissions.

to pthread_mutex_ulock(&l) not only changes the state of the lock l, conceptually it gives away control to the data in memory protected by l. Such behavior that reveals locations in memory cannot be expressed in CompCert events that can't have pointers on them (except the location of global variables). More generally, these events are poorly suited to express any sort of shared memory interactions, such as concurrency or separated compilation. We propose to include 2 new types of events which we call *memory events* as described in table 4.9. As their name suggests, they contain references to locations in memory, beyond the global variables. The first one, Event_acq_rel, represents a generic memory interactions 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. We find it convenient to split the effects on memory as those that happen before the changes in permissions and those that happen after. 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_map, one representing the permissions given and the other the starting permissions of a new thread/module.

The correctness of the CompCert compiler is formulated as a preservation of traces. However, if our traces contain memory events and the memory can be reordered by compilation, the trace must be reordered accordingly. Thus our new correctness will be formulated up to memory reordering as shown in **??**. In fact, if a compiler pass (or passes) don't change the order of memory (such as equality and extension passes), then the trace is preserved and nothing changes in the simulation. If the pass changes the order of memory according to some j, then we need show that the source and target traces,t and t', are related by inject_trace_strong j t t' (denoted $t \overset{j'}{\hookrightarrow} t'$). That means that events in t and t' are identical except all memory locations are reordered according to j. To add this property, we need simulations to expose how they reorder memory. The full description of our new conjunctive simulation will be explained in section 4.4.

Compilers not only reorder memory, sometimes undefined values in the source program are mapped to concrete values in the target. Such mappings are useful in compiler passes such as register coalescing, where registers that were previously uninitialized, can now map to an initialized register with concrete values. The predicate inject_trace_strong always maps undefined values to undefined values. This is a reasonable restriction since inspecting undefined values is not an allowed behavior, so they shouldn't appear in the trace. However, if an application required traces with undefined values, we can still support that. It turns out that it is enough to prove that the execution with the *strongly injected* trace is safe, and from it we can derive save executions for all traces that have some undefined values defined.

4.3 Conjunctive 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 *conjunctive semantics* and it extends CompCert semantics with the following

```
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.
```

Table 4.10: at_external definition for Clight. The function checks that (1) the current state is about to make a function call, (2) that the function is an External function and, (3) that the external function cannot be inlined (The compiler is allowed to inline specific functions such as memcpy and certain builtins).

- get_mem and set_mem: The state of every language in CompCert can be interpreted as a pair of a *core* and a memory [7]. get_mem is the projection that returns the memory inside the state and set_mem changes the memory.

- entry_points: This is a generalization of initial_state, as described in section 4.1.

- 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 table 4.10.

Our conjunctive semantics is very closely related to *interaction semantics* [7] with two main differences. First, we don't need to define after_external for every language. Second, states that are at_external can 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 abstracted into oracles that execute in one step.[4].

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

In fact, given a conjunctive 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.

4.4 More revealing simulations

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

Theorem 4.4.1 (CompCert semantic preservation) *Let S be a source program and C its compiled version. For all behaviors B that don't go wrong, if S has behavior B, then C also 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 specification, such as the higher order, separation logic specifications that can be proven on Clight programs by tools like [2] or [?]. Moreover, the high level specification in Equation 4.1 is not well suited for modular reasoning to support shared memory concurrency or compositional compilation [7].

We consider that the simulations that CompCert uses to prove Equation 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 Equation 4.1. These simulations, encoded in the record fsim_properties,

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

(a)

$$
\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'\ *"] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}
\qquad
\begin{array}{c}
t \overset{j'}{\hookrightarrow} t' \\[1em]
j \sqsubseteq j'
\end{array}
$$

(b)

Table 4.11: Step simulation step diagrams. (a) if s_1 takes a step to s_2 with trace t and s_1 is related to some s_2, then there s_2 can take a number of steps with trace t to a new state $2_2'$ related to s_1'. (b) The new diagram exposes the memory reordering injections j and j' and the traces t and t' are equivalent up to injection, by inject_trace_strong j' t t'.

state that (1) public global variables and functions are preserved, (2) initial_states are preserved (Table 4.5), (3) final_states are preserved and (4) execution is preserved (Table 4.11(a)). The simulations is parametric on a *match relation*, noted as $\sim$, as an invariant of related states in source and target; 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 doesn't 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-allocate 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 (Table 4.11(b)).

We propose a more expressive simulation inject_sim that improves the CompCert simulations in the following ways:

- 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 preserve the memory or just extend it, we also define the simpler simulations eq_sim and extend_sim respectively. These simulation follow immediately from the ones already proven in CompCert. All of the simulations we define compose horizontally to inject_sim as shown by the composition lemmas Theorem 4.4.2, Theorem 4.4.3 and Theorem 4.4.4.

Lemma 4.4.2 *For all semantics L_1 and L_2 if eq_sim L_1 L_2 then extend_sim L_1 L_2*

Lemma 4.4.3 *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 4.4.4 *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*

- Preserves external function calls: The original CompCert simulation only preserves traces so, 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 builtins. 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 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).

- Preserves the number of steps taken by external functions: This fact was already proven in the CompCert but was hidden in the less expressive simulation. We

$$
\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"'] \\
s_1' \arrow[r, phantom, "\overset{j'}{\sim}"] & s_2'
\end{tikzcd}
\qquad
\begin{array}{c}
t \overset{j'}{\hookrightarrow} t' \\
j \sqsubseteq j' \\
\textsf{at\_external}\ s_1 = \textsf{Some (f,args)}
\end{array}
$$

Table 4.12: At external step diagram (simulation_atx). Exclusive for external function calls, this diagram follows the simulation diagram in Table 4.11, but enforces that the compiled execution takes only one step.

include a new diagram (Table 4.12), 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 Table 4.11), and the two resulting states match.

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

- 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 Table 4.5.

It might be surprising that we don't further change the diagram for entry_points. In CompComp [7], the initial core simulation must accept almost arbitrary (but injected) memories. Our techniques allow us to assume that the context is not changing while the program compiles. Similarly, we can expect that the execution of external functions changes, based how the compiler reorders the memory, but the external function will not change the order in which it allocates memory.

As mentioned before, our definition of simulation_atx might be too strong for external functions that have traces with undefined values. If those were read from memory allocated by the compiling program, it is reasonable that the values become concrete as the program compiles. Fortunately, it is enough (and easier) to prove the stricter version described in Table 4.12. We do provide the more permissive version

```
1  Definition simulation_atx_inj_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, Step L1 s1 t s1' →
6                          match_states i f s1 s2 →
7                          ∃ f', Values.inject_incr f f' ∧
8                            (∃ i' s2' t',
9                                Step L2 s2 t' s2' ∧
10                               match_states i' f' s1' s2' ∧
11                               inject_trace_strong f' t t') ∧
12                           (∀ t', inject_trace f' t t' →
13                                    ∃ i', ∃ s2',
14                                    Step L2 s2 t' s2' ∧
15                                    match_states i' f' s1' s2') .
```

Table 4.13: Stronger simulation for external steps, that universally quantifies over all injected traces. Lines 8-11 describe the existentially quantified diagram as described in Table 4.12. Lines 12-15, in bold, describe all the other executions that may have undefined values determined. inject_trace is the predicate that allows undefined values to be mapped to defined ones.

of the simulation as part of the external specification of the compiler. The full Coq definition is presented in Table 4.13 with the addition highlighted.

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 don't 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 (although the implementation is beyond the scope of this thesis). We further discuss solutions for this limitation in related work chapter 2 and in our future work section 6.1 sections.

Consider the remember() and incr() functions from Table 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 and 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 it's address has escaped, but this fact 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 it gains access to, is not unmapped and unchanged. A thread can only use the locations it has permission over (which is a superset of the locations it accesses). This approach allows us to ensure that the memory doesn't change when other threads execute. Unfortunately, if part of the memory is unmapped, we can't 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 solve the problem for concurrency.

In his original paper about CompCert Leroy [5] claims that "inputs given to the programs are uniquely determined by their previous outputs". That seems to suggest that functions like 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 are not unmapped by SimplLocals or Unusedglob, since it only contains escaping pointers. These changes are beyond the scope of the thesis, so we temporarily use full_injection and we skip the two problematic passes. We discuss this solution further in the section 6.1.

Chapter 5

Compiler Correctness

This chapter describes the compiler correctness proof and its implementation.

5.1 Compiler Theorem

Here goes the compiler theorem

5.1.1 Self simulations

5.2 Compiler Theorem Implementation

Here goes the compiler theorem implementation

Chapter 6

Conclusion

Conclusion text.

6.1 Future Work

Lots to talk about here.

Appendix A

Implementation Details

Implementation details.

A.1 CPM design

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Xavier Leroy. The CompCert verified compiler, software and ann. proof, March 2019.

[7] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional compcert. In *POPL*, volume 50, pages 275–287. ACM, 2015.