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 4

MOIST simulations of CompCert

The simple code in Figure 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 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 Figure 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

```
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).

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 Figure 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 Figure 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 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

[1]Leroy [7] 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 :=                        Inductive eventval: Type :=
  | Event_syscall:                                | EVint: int → eventval
        string → list eventval →                  | EVlong: int64 → eventval
        eventval → event                          | EVfloat: float → eventval
  | Event_vload:                                  | EVsingle: float32 → eventval
        memory_chunk → ident →                    | EVptr_global: ident → ptrofs → eventval.
        ptrofs → eventval → event
  | Event_vstore:
        memory_chunk → ident →
        ptrofs → eventval → event
  | Event_annot:
        string → list eventval → event
```

Figure 4.3: The events in CompCert

pointer stored in buff as part of its trace but CompCert events, shown in Figure 4.3, can only contain integers, floats or pointers to global variables.

Moreover, 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, this guarantee is not exposed in its specification.

Finally, 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]. At least two other works ([10], [6]) have proposed alternative CompCert specifications that make the simulations an exposed feature of the compiler's specification. In

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

these papers, the authors view CompCert correctness modularly as a thread-local or module-local 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.

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 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 $\to$ 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 $\to$ **option** (f_ext, args) *returns the external function being called and its arguments. The simulation preserves external calls.*

	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.

- *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 [1] or abstract state in [5] 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

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

Figure 4.4: CompCert memory model.

and at_external functions and, finally, we show how to put everything together in MOIST simulations for CompCert.

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 [9]. 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

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

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.

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

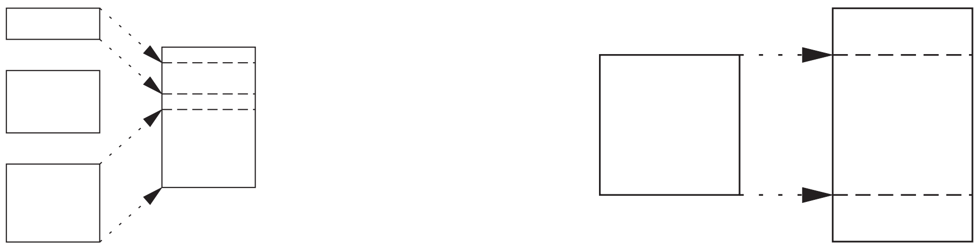

(a) Memory injection (b) Memory extension

Figure 4.5: Memory transformations of CompCert. [1]

reordering them, deleting them, changing their internal offsets or even merging two of them. Leroy and Blazy [9] 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 bellow.

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 won't reorder the blocks; we call these *extension passes* Figure 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* Figure 4.5(a). All passes can be shown to be injections [10], but it is much easier to treat equality and extension phases as special cases.

An injection is described by a mapping $j : \mathsf{block} \to \mathbf{option}\ (\mathsf{block} * \mathsf{Z})$, 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 oth-

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

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

(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

ers as shown in Figure 4.5(a). Beyond a permutation of memory, an injection induces 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 Figure 4.6(a). In the stronger injection, constants are preserved and pointers are renamed according to the injection. In the simpl 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 Figure 4.6(b).

Definition 2 (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:*

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

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

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

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

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

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

- *mi_representable and mi_perm_inv [1]*

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 Figure 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 (Figure 4.2), that characterizes starting states which includes calls to main with arguments and calls to any other function.

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 argu-

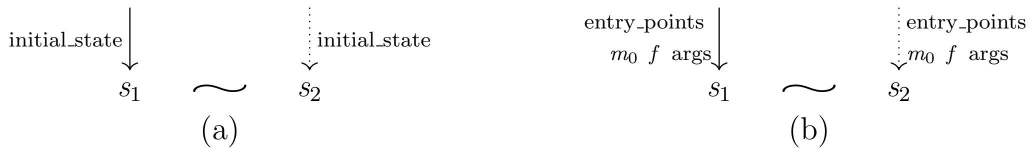


Figure 4.7: Entry simulation diagrams: (a) if s_1 is an initial state for the source program, then there exists some state s_2 that is related to s_1 and is an initial state for the compiled program. (b) Just like the diagram for initial states, but it generalizes and exposes the initial memory m_0,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

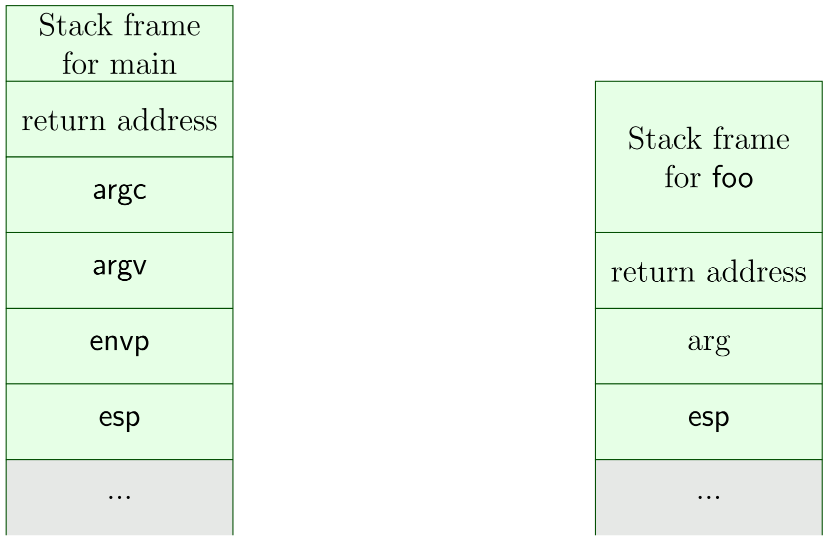
ments, threads wouldn't be able to share pointers (or would have to do it clumsily through global variables) and thus they would all execute in disjoint pieces of memory with no communication. That would be a much easier and less interesting result.

Once we define a new starting point for executions, we must prove that compilation preserves the predicate entry_point (Figure 4.7(b)) in the same way that CompCert's simulation preserves initial_state (Figure 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 subsection 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 *prestack* that tip of



(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.

the stack, depicted in Figure 4.8(a), which 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, 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.

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 Figure 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 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!" [8]. 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 in the next subsection.

4.2.2 The entry_point: a more permissive 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, and is 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.[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.

[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.

```
 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).
```

Figure 4.9: The entry_point predicate in Clight

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 Figure 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 Figure 4.9. Lines 4-6 ensure that the environment 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.

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 frame containing arguments to main. Consequently, during later phases of compila-

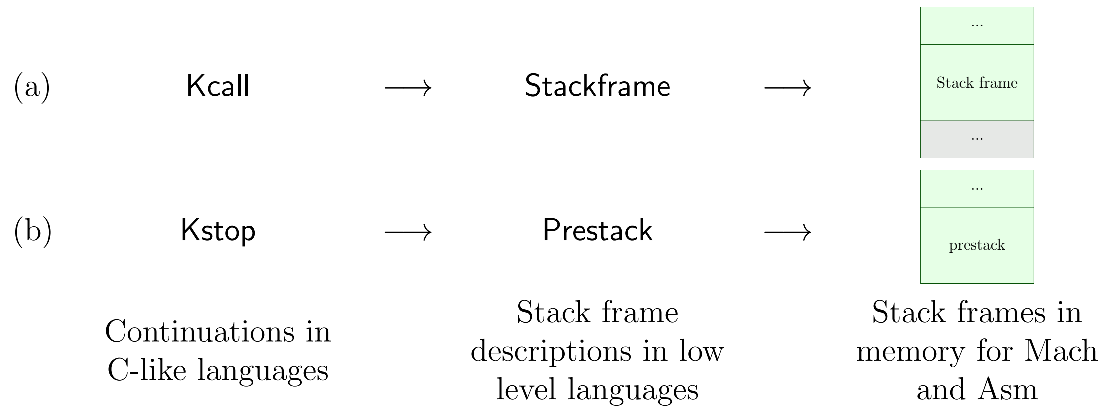

(a)
Kcall
Stackframe
...
Stack frame
...
(b)
Kstop
Prestack
...
prestack
Continuations in C-like languages
Stack frame descriptions in low level languages
Stack frames in memory for Mach and Asm

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

tion, 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, a special kind of Stackframe, as depicted in Figure 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 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. Hence, for languages

```
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

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 Figure 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, after the Stacking pass, it is translated to an actual prestack as shown in Figure 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 Figure 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 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 Figure 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 Figure 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 Figure 4.12(b), "because these are not preserved literally during compilation"[7]. In fact, for this reason, CompCert doesn't have any event that exposes memory state or pointer values. Unfortunately, this limitation rules out several reasonable programs such as one calling incr (Figure 4.1) and the one in Figure 4.12(b), moreover, these events are poorly suited to express any sort of shared memory interactions, such as concurrency or separated compilation. For example, a call 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. The location and the content of the data might expose memory state, so it cannot be expressed in CompCert semantics.

```
x = 3;
*a = x;
putbuf(a);
```

(a)

```
y = 3;
x = &y;
*a = x;
putbuf(a);
```

(b)

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

(c)

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

(d)

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

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 Figure 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 must not communicate uninitialized variables to the external world; it can be a security risk. But output is not always going to the external world. If putbuf was a verified library, it would be acceptable to reveal undefined values to it. For another example, consider the code in Figure 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[1]. However, doing so would rule out some reasonable programs that we would like to verify (such as the one in Figure 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 Figure 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

```
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.
```

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

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 Figure 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 Figure 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 Figure 4.6(a), the value Vundef can be injected to any other value. Consequently, the injection relations for memories, for mem_effects, for events and for traces are not deterministic. We do, however,

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

(a) Injection of ranges.

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

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

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

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

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

(c) Strong injection of mem_effects

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

(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.

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 subsection 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 [2] and get_mem is the projection that returns the memory inside the state.
- 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 Figure 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* [10] 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

```
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) 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).

after_external; therfore, there is 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 abstracted into oracles that execute in one step.[6].

In fact, given a Startable factorable-state semantics 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:

[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.

Theorem 4.5.1 (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 VST [3] or *What's it's name?* [?].Moreover, the high level specification in Equation 4.1 is not well suited for modular reasoning to support shared memory concurrency or compositional compilation [10].

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, state that (1) public global variables and functions are preserved, (2) initial_states are preserved (Figure 4.7), (3) final_states are preserved and (4) execution is preserved (Figure 4.16(a)). The simulations are 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

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\sim"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t"', "*"] \\
s'_1 \arrow[r, phantom, "\sim"] & s'_2
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
s_1 \arrow[r, phantom, "\overset{j}{\sim}"] \arrow[d, "t"'] & s_2 \arrow[d, dotted, "t'"', "*"] \\
s'_1 \arrow[r, phantom, "\overset{j'}{\sim}"] & s'_2
\end{tikzcd}
\qquad
\begin{array}{c} t \overset{j'}{\hookrightarrow\!\!\!\twoheadrightarrow} t' \\[1em] j \sqsubseteq j' \end{array}$$

(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').

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 (Figure 4.16(b)).

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

- 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 preserve the memory or just 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 horizontally 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 extend_sim L_1 L_2.*

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 Figure 4.16, but enforces that the compiled execution takes only one step.

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 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). We consider the simulations of built-ins in 4.5.2.

- The simulation preserves the number of steps taken by external functions: This fact was already proven in the each CompCert phase but was hidden in the

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

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

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

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.

less expressive simulation theorems exposed by each phase. We include a new diagram (Figure 4.17), 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 Figure 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 Figure 4.7.

Our simulation diagrams for entry_points is strikingly simple compared to that of other authors. In Compositional CompCert [10] and in CASCompCert [6], 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 it unnecessarily complicates passes that don't 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

[7]Stewart et al.[10] modify all passes to use injections. In other work [2] the authors propose using nine different composition theorems to compose all possible memory relations pairs (i.e., equality, extension and injection).

to achieve concurrency in chapter 5 and we conjecture that the diagram also is enough for separate compilation, which we discuss in section 6.1.

Summary regarding inject_trace_strong.

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

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

1. CompCert correctness proofs can mostly behave as if trace-injection is deterministic, so Vundef injects to Vundef;
2. Event_acqrel can inject from Vundef to defined values, which permits programs where semaphores control shared access to uninitialized buffers.

Theorem 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 [4] 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 Figure 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 so, to establish a behavioral at_externals_1 = Some (f,args)

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

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

(a)

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

(b)

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

refinement, we also need the diagram in Figure 4.19(b). It says 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 Figure 4.20, where the bolded code corresponds to the second diagram Figure 4.19(b).

Keep in mind that we could avoid needing the second diagram in Figure 4.19, if we keep uninitialized variables out of the trace, just like CompCert does. We have defined *strong injections* (the $\hookrightarrow\!\!\!\!\twoheadrightarrow$ in Figure 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 don't reveal uninitialized values in the trace.[8] However, we want (and can!) support functions that expose uninitialized variables.

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

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

```
 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 **??**. Lines 12-15, in bold, describe the diagams for all other images of thee trace injection relation as described in Figure 4.19(b). inject_trace is the predicate that allows undefined values to be mapped to defined ones.

commute with injection; for external functions with memory events, the only way to commute with injections is to satisfy the two diagrams in Figure 4.19.

We must wonder if there are reasonable functions that don't satisfy both diagrams in Figure 4.19. The answer is yes: imagine a function show_a interacting with the code in Figure 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 can't support, show_a and other similar functions? No. We can define a behavior[9], denoted $\overline{\text{show\_a}}$, whose ouputs the 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 can't distinguish between show_a and $\overline{\text{show\_a}}$. That is because a safe program cannot compare an undefined value. Similarly, if the

[9]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 Figure 4.17

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 can 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 simulation, inject_sim and inject_sim_strong, with the external step diagrams simulation_atx (Figure 4.21) and simulation_atx_strong (Figure 4.20) respectively. We prove that they compose, according to the lemma below, so we can 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.[10]

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.*

[10]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 (subsection 5.1.1), then all compiler proofs can be specified according to the simpler inject_sim, and we still get the stronger property for the entire compiler.

4.5.2 Simulation diagrams for builtins.

Our simulations require that external function calls are preserved 1-to-1 by the compiler (Figure 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 Builtins (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 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 builtins 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 builtins 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 don't 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 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 builtins 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 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 builtins. 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 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 execute 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 Builtins 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, 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; builtins (even inline assembly) would be expressed as if they were function calls; then rewritten as Ibuiltin in the RTL. The semantics would end up identical, the generated code would 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 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 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 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 Figure 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 its 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. 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 (by a compiler phase, because the program contains no accesses to it), 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 [7] 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 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.

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] 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. Springer, 2014. To appear.

[3] 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.

[4] Robert W. Dockins and Andrew W. Appel. *Operational refinement for compiler correctness—*. PhD thesis, Princeton University, July 2012.

[5] 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.

[6] 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.

[7] Xavier Leroy. compcert-compcertrealistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

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

[9] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1), 2008.

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