Towards Certified Compositional Compilation for Concurrent Programs

ANONYMOUS AUTHOR(S)

Certified compositional compilation is important for establishing end-to-end guarantees for certified systems consisting of separately compiled modules. In this paper, we propose a language-independent framework consisting of the key semantics components and verification steps that bridge the gap between the compilers for sequential programs and for (race-free) concurrent ones, so that the efforts on certified sequential compilation can be reused for concurrent programs. One of the key contributions of the framework is a novel footprint-preserving compositional simulation as the compilation correctness criterion. With our framework, we have verified the correctness of the CompCert x86 backend (including all the translation phases from Cminor to x86, and two optimization phases) for compositional compilation of race-free concurrent programs.

1 INTRODUCTION

Compositional compilation is important for real-world systems, which usually consist of multiple program modules that need to be compiled independently. Correct compilation then needs to guarantee that the target modules can work together and preserve the semantics of the source program as a whole. It requires not only that individual modules be compiled correctly, but also that the expected interaction between modules be preserved at the target.

CompCert [Leroy 2009a], the most well-known certified realistic compiler, establishes the semantics preservation property for compilation of *sequential* Clight programs, but with no explicit support of separate compilation. To support general separate compilation, Stewart et al. [2015] develop Compositional CompCert, which allows the modules to call each other's external functions. Like CompCert, Compositional CompCert only supports sequential programs too. There each module interacts with others only at certain program points specified in the module code (i.e., at external calls only). Also there needs to be one-to-one correspondence between the interaction points in the target program and those in the source. However, in interleaving concurrency, a thread (viewed as a module) can be preempted by others at any non-deterministically chosen program points, and the target program is usually more fine-grained and have more interaction points than the source. It is unclear if their approach can be applied to compilation of concurrent programs.

Stewart et al. [2015] do argue that Compositional CompCert may be extended for certified compositional compilation of data-race-free (DRF) concurrent programs. They argue that, for DRF concurrent programs, the behaviors of the threads under the standard interleaving semantics should be equivalent to those in some non-preemptive semantics where the control of the CPU switches between threads at certain designated program points. Since a thread cannot be interrupted between two consecutive switch points in the non-preemptive semantics, the code segment between the two switch points can be compiled as sequential code. The sequential compilation is sound as long as the switch points are viewed as external function calls so that optimizations do not go beyond them. Although the argument is plausible, there are still significant challenges in building fully certified compositional compilers for DRF concurrent programs:

- We need a proper formulation of the non-preemptive semantics and the notion of DRF. On the one hand, the formulation relies on the synchronization constructs in the language and the notion of footprints (i.e. the memory locations accessed in each step), On the other hand, like the interaction semantics in Compositional CompCert [Stewart et al. 2015], the formulation

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

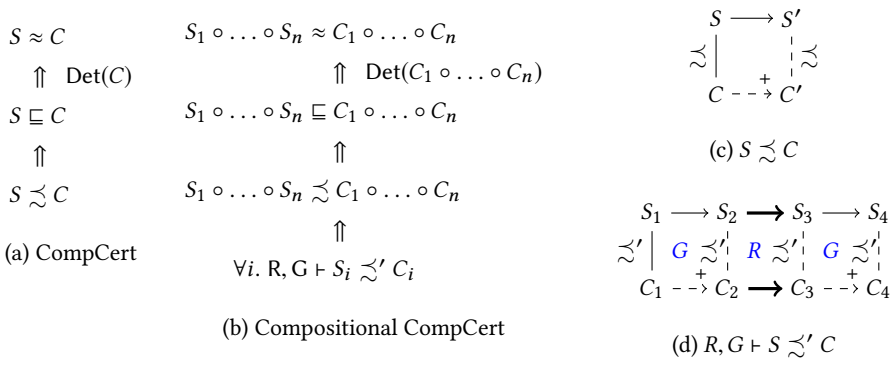
should be *language-independent* to support general compositional compilation, where the modules can be implemented in different languages.

- We need to prove that DRF programs behave the same in the standard preemptive semantics and in our non-preemptive semantics, preferably in a language-independent setting. Also the equivalence should be strong enough to preserve the termination of programs, a natural correctness requirement for certified compilation. Although such semantics equivalence has been known as a folklore theorem, we have not seen any mechanized proofs of such *termination-preserving* semantics equivalence in a language-independent setting. As we will explain in Sec. 2.2, the proofs can actually be quite challenging, and the complexity is affected by many factors of language semantics design, such as semantics for memory allocation.
- We need to prove the compilation preserves DRF, which ensures that the target compiled from a DRF source is DRF too. Then the behaviors of the target program in the preemptive semantics are the same as those in the non-preemptive semantics. However, since a data race involves the behaviors of at least two threads, DRF is not a thread-local property. Then, how do we prove DRF-preservation of a compositional compilation, which compiles one module/thread at a time without knowledge of other modules/threads? That is, we need a compositional proof of the non-local DRF-preservation property.
- Concurrency introduces non-determinism, which makes it difficult to directly reuse the downward simulation proofs in CompCert and Compositional CompCert. Instead of proving the source program simulates the target (i.e., upward simulation), CompCert proves the reverse direction (downward simulation), and then derive the semantics equivalence based on the determinism of the language semantics. Since the source is usually more coarse-grained than the target, the downward simulation is simpler to prove than the upward one. However, it is unclear if such a proof strategy can still be applied for concurrent languages, whose semantics is non-deterministic.

We will explain the challenges in detail in Sec. 2, and discuss more related work on certified compilation in Sec. 9.

In this paper, we propose a language-independent framework consisting of the key semantics components and verification steps that bridge the gap between compilation for sequential programs and for DRF concurrent programs. We also apply our framework to verify the correctness of CompCert x86 backend for compositional compilation of DRF programs. Our work is based on previous work on certified compilation, but makes the following new contributions:

- We design a compositional *footprint-preserving simulation* as the correctness formulation of separate compilation for sequential modules. As an extension of the simulation in Compositional CompCert, our simulation considers module interactions at both external function calls and synchronization points, thus is compositional with respect to both module linking and non-preemptive concurrency. It also requires that the footprints of the steps made by the source and target modules be related, where footprints refer to the set of memory locations accessed at each step of execution. This way we reduce the proof of DRF-preservation for whole programs into proofs of *local* footprint preservation.
- We work with an *abstract* programming language, which is not tied to any specific synchronization constructs such as locks but uses abstract labels to model how such constructs interact with other modules. It also abstracts away the concrete primitives that accesses memory. We introduce the notion of *well-defined languages* to enforce a set of constraints over the state transitions and the related footprints, which actually give an extensional interpretation of footprints. These constraints are satisfied by various real languages such as Cminor and x86 assembly. With the abstract language, we study the equivalence between preemptive



(a) CompCert (b) Compositional CompCert (c) $S \precsim C$ (d) $R, G \vdash S \precsim' C$

Fig. 1. Proof structures of certified compilation.

and non-preemptive semantics, the equivalence between DRF and NPDRF (the notion of race-freedom defined in the non-preemptive setting [Xiao et al. 2018]), and the properties of our new simulation. As a result, the lemmas in our proof framework are re-usable when instantiating to real languages.

- Putting all these together, our framework (see Fig. 2) is the first to build certified compositional compilation for concurrent programs *from sequential compilation*. It highlights the importance of DRF preservation for correct compilation. It also shows a possible way to adapt the existing work of CompCert and Compositional CompCert to interleaving concurrency.
- As an instantiation of our language-independent compilation verification framework, we instantiate the source and target languages as Cminor and x86 assembly with lock prefix. We have successfully proved that the CompCert-3.0.1 [2017] x86 backend (including all the translation phases and one optimization phase) satisfy our compilation correctness criterion. In the verification of the CompCert compilation phases, we reuse a considerable amount of the original CompCert proofs, with minor adjustment for footprint-preservation. The proofs for each phase take less than one person week on average.

In the rest of this paper, we first analyze the challenge and give an overview of our approach in Sec. 2. We give the basic technical setting in Sec. 3, including the preemptive semantics and the refinement definition. Sec. 4 presents the non-preemptive semantics, which is the basis for both our new simulation (Sec. 5) and the NPDRF definition (Sec. 6). We show the final theorem in Sec. 7, and discuss the implementation details in Sec. 8. We conclude and discuss related work in Sec. 9.

2 INFORMAL DEVELOPMENT

Below we first give an overview of the main ideas in CompCert [Leroy 2009a] and Compositional CompCert [Stewart et al. 2015] as starting points for our work. Then we explain the challenges and our ideas in reusing them to build certified compositional compilation for concurrent programs.

2.1 Background

2.1.1 CompCert. The pioneer work on CompCert [Leroy 2009a,b] builds certified compilation for sequential programs. Fig. 1(a) shows its key proof structure.

The compilation *Comp* is correct, if for every source program S, the compiled code C preserves the semantics of S. That is, $\mathrm{Correct}(Comp) \stackrel{\text{def}}{=} \forall S, C.\ Comp(S) = C \implies S \approx C$. Here the semantics preservation $S \approx C$ requires S and C have the same sets of observable event traces:

$$S \approx C \ \text{ iff } \ \forall \mathcal{B}.\ Etr(S, \mathcal{B}) \iff Etr(C, \mathcal{B})\ .$$

Here we write $Etr(S, \mathcal{B})$ to mean that an execution of S produces the observable event trace $\mathcal{B}$, and likewise for C. The observable events include control effects (e.g., termination and exceptions) and input-output events, which in CompCert correspond to invocations of external functions.

To verify $S \approx C$, CompCert relies on the determinism of the target language (written as Det(C) in Fig. 1(a)) and proves only the downward direction $S \sqsubseteq C$, i.e., S is a refinement of C.

$$S \sqsubseteq C \text{ iff } \forall \mathcal{B}.\ Etr(S, \mathcal{B}) \implies Etr(C, \mathcal{B})\ .$$

The determinism Det(C) ensures that C admits only one observable behavior, so one can derive the upward refinement $S \sqsupseteq C$ from $S \sqsubseteq C$. The latter is then proved by constructing a (downward) simulation relation $S \precsim C$. Depicted in Fig. 1(c), the simulation $S \precsim C$ establishes a *consistency relation* between S and C, which is always preserved under some correspondence between the executions of S and C. Every step of S must correspond to zero-or-more steps of C. Figure 1(c) shows the case when the source step corresponds to multiple target steps.

The simulation $\precsim$ serves as a proof technique for verifying whole-program compilation, but it is not compositional and *cannot* be used for verifying separate compilation. This is because $\precsim$ does not take into account the interactions with other modules which may update the shared resource.

2.1.2 Compositional CompCert. Compositional CompCert [Stewart et al. 2015] supports separate compilation by re-defining the simulation relation for modules. Figure 1(b) shows the proof structure of Compositional CompCert. Suppose we have separate compilation $Comp_1, \ldots, Comp_n$. Each $Comp_i$ transforms a source module S_i to a target module C_i. The overall compilation is correct if, when linked together, the target modules $C_1 \circ \ldots \circ C_n$ preserve the semantics of the source modules $S_1 \circ \ldots \circ S_n$ (here we write $\circ$ as the module-linking operator). For example, the following program consists of two modules. The function `f` in module `S1` calls the external function `g`. The external module `S2` may access the variable `b` in `S1`.

```
// Module S1
extern void g(int *x);
int f(){
  int a = 0, b = 0;
  g(&b);
  return a + b; }
```

```
// Module S2
int g(int *x){
  *x = 3;
}
```

(2.1)

Suppose the two modules `S1` and `S2` are independently compiled to the target modules `C1` and `C2`. Aligned with the external call in `S1`, `C1` also calls the external function `g` in `C2`. The correctness of the overall compilation requires (`S1` $\circ$ `S2`) $\approx$ (`C1` $\circ$ `C2`).

With the determinism of the target modules, we only need to prove the downward refinement (`S1`$\circ$`S2`) $\sqsubseteq$ (`C1`$\circ$`C2`), which is reduced to proving (`S1` $\circ$ `S2`) $\precsim$ (`C1` $\circ$ `C2`), just as in CompCert. Ideally we hope to know (`S1`$\circ$`S2`) $\precsim$ (`C1`$\circ$`C2`) from `S1` $\precsim$ `C1` and `S2` $\precsim$ `C2`, and ensure the latter two by correctness of $Comp_1, \ldots, Comp_n$. However, the CompCert simulation $\precsim$ is not compositional.

To achieve compositionality, Compositional CompCert defines the simulation relation $\precsim'$ shown in Fig 1(d). It is parameterized with the interactions between modules, formulated as the rely/guarantee conditions R and G [Jones 1983]. The rely condition R of a module specifies the permitted state transitions of its environment (i.e., other modules that may be linked together) at both the source and target levels, and the guarantee G specifies the possible transitions made by the module itself. The simulation diagram in Fig 1(d) requires that the steps of the current module (the thin arrows) be allowed in G, and the simulation $\precsim'$ be preserved by the environment steps R (the thick arrows). Note that the R steps happen only at the *external function calls* of the current module. It models the general callee behaviors. Such a simulation is compositional as long as the rely/guarantee conditions of linked modules are compatible.

Compositional CompCert proves that the CompCert compilation phases satisfy the new simulation $\precsim'$ (which is stronger than the CompCert simulation $\precsim$). The intuition is that the compiler optimizations do not go beyond external calls (unless only local variables get involved). That is, for the above example (2.1), the compiler cannot do optimizations based on the assumption that `b` is 0 at the last line of the function `f`.

Since the *R* steps happen only at the *external function calls* in Compositional CompCert, it cannot be applied to concurrent programs, where module/thread interactions may occur at any program point. However, if we consider race-free concurrent programs only, where threads are properly synchronized, we may consider the interleaving at certain synchronization points only. It is a well-known folklore theorem that DRF programs in interleaving semantics behave the same as in non-preemptive semantics. For instance, the following program (2.2) uses a lock to synchronize the accesses of the shared variables, and it is race-free. Its behaviors are the same as those when the threads yield controls at `lock()` and `unlock()` only. That is, we can view lines `1-2` and lines `4-5` in either thread as sequential code that will not be interfered by the other. The interactions occur only at the boundaries of the critical regions.

```
1  r1 = 1;          ||  r2 = 2;
2  r1 = r1 + 1;     ||  r2 = r2 + 1;
3  lock();          ||  lock();
4    x = 1;         ||    x = 2;
5    y = x + 1;     ||    y = x + 1;
6  unlock();        ||  unlock();
```
(2.2)

Intuitively, we can use Compositional CompCert to compile the program, where the code segment between two consecutive switch points is compiled as sequential code. By viewing the switch points as special external function calls, the simulation $\precsim'$ can be applied to relate the non-preemptive executions of the source and target modules.

Although the idea is plausible, we have to address several key challenges to really apply it to build certified compositional compilers, as we explain below.

2.2 Challenges and Our Approaches in Verifying Concurrent Program Compilation

2.2.1 How to give language-independent formulation of DRF and non-preemptive semantics? The interaction semantics in Compositional CompCert describes module interaction without referring to the concrete languages used to implement the modules. This allows composition of modules implemented in different languages. We would like to follow the semantics framework, but how do we define DRF and non-preemptive semantics if we do not even know the concrete synchronization constructs and the commands that access memory?

Our solution. We extend the module-local semantics in Compositional CompCert so that each local step of a module reports its footprints, i.e. the memory locations it accesses. Instead of relying on the concrete memory-access commands to define what valid footprints are, we introduce the notion of *well-defined languages* (in Sec. 3) to specify the requirements over the state transitions and the related footprints. For instance, we require the behavior of each step is affected by the read set only, and each step does *not* touch the memory content outside of the write set. When we instantiate the framework with real languages, we prove they satisfy these requirements.

Besides, we also allow module-local steps to generate messages EntAtom and ExtAtom to indicate the boundary of the atomic operations inside the module. The concrete commands that generate these messages are unspecified, which can be different in different modules. In Sec. 8.1 we show how the lock-prefixed x86 instructions as synchronization constructs generate these messages.

2.2.2 What memory model to use in the proofs? The choice of memory models could greatly affect the complexity of proofs. For instance, using the same memory model as CompCert allows us to reuse CompCert proofs, but it also causes many problems. The CompCert memory model records the next available block number `nextblock` for memory allocation. Using the model under a concurrent setting may allow all threads to share one `nextblock`. Then allocation in one thread would affect the subsequent allocations in other threads. This breaks the property that re-ordering non-conflicting operations from different threads would *not* affect the final states, which is a key lemma we rely on to prove the equivalence between preemptive and non-preemptive semantics for DRF programs. In addition, sharing the `nextblock` by all threads also means we have to keep track of the ownership of each allocated block when we reason about footprints.

Our solution. We decide to use a different memory model from the one of CompCert. We reserve separate address spaces F for memory allocation in different threads (see Sec. 3). Therefore allocation of one thread would not affect behaviors of others. This greatly simplifies the semantics and the proofs, but also makes it almost impossible to reuse CompCert proofs, as we explain in Sec. 8.2. We address this problem by establishing some semantics equivalence between our memory model and the CompCert memory model (shown in Sec. 8.2.1).

2.2.3 How to compositionally prove DRF-preservation? The simulation $\precsim'$ in Compositional CompCert is not sufficient to ensure DRF-preservation. As we have explained, DRF is a whole-program property, and so is DRF-preservation. To support separate compilation, we need to reduce the requirement of DRF-preservation on whole programs to some requirements on single threads. In particular, we hope to encode the requirements in the thread-local and module-local simulation.

Our solution. We propose a new compositional simulation $\preccurlyeq$ which extends $\precsim'$ with the requirements of footprint-preservation on single threads. In detail, based on the simulation diagram in Fig. 1(d), we additionally require *footprint consistency* saying that the target C should have the same or smaller footprints than the source S during related transitions. For instance, when compiling lines `4-5` of the left thread in (2.2), the target is only allowed to read `x` and write to `x` and `y`.

Note that we check footprint consistency at switch points only. This way we allow compiler optimizations as long as they do not go beyond the switch points. For the example in (2.2), the target of lines `4-5` of the left thread could be `y=2;x=1` where the writes to `x` and `y` are swapped.

2.2.4 How to flip refinement/simulation with non-deterministic behaviors? As we explained, the last steps of CompCert and Compositional CompCert in Fig. 1 derive semantics equivalence $\approx$ (or the upward refinement $\sqsupseteq$) from the downward refinement $\sqsubseteq$ using determinism of target programs. Actually the simulations $\precsim$ and $\precsim'$ can also be flipped if the target programs are deterministic. It is unclear if the refinement or simulation can still be flipped in the concurrent settings where programs have non-deterministic behaviors. The problem is that the target program can be more fine-grained and have more non-deterministic interleavings than the source.

Our solution. Data-race-freedom and the non-preemptive semantics come to the rescue. For DRF programs, the switch points in the target are aligned with those in the source under the non-preemptive semantics. Also the target and source programs can always make the same non-deterministic choices of thread switching. Thus we are able to flip the downward simulation to derive upward simulation, assuming determinism of each target module.

2.3 The framework and key semantics components

Figure 2 shows the semantics components and proof steps in our framework. The ultimate goal of our compilation correctness proof is to show the semantics preservation of the source and target

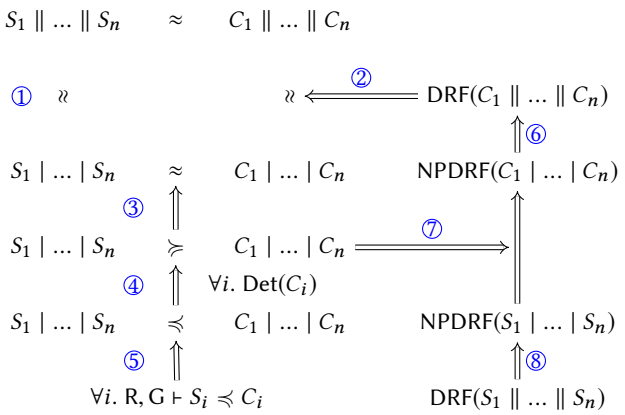


Fig. 2. Our framework

programs (i.e., $S_1 \parallel \ldots \parallel S_n \approx C_1 \parallel \ldots \parallel C_n$ at the top of the figure). This follows the correctness of the separate compilation of each module, formulated as $\mathsf{R}, \mathsf{G} \vdash S_i \preccurlyeq C_i$ (the bottom left), which is our new footprint-preserving module-local simulation explained above. We do the proofs in the following steps. Note that the double arrows in the figure represent logical implication.

First, we restrict the compilation to source preemptive programs that are data-race-free (i.e., $\mathsf{DRF}(S_1 \parallel \ldots \parallel S_n)$ at the right bottom of the figure). Then from the equivalence between the preemptive and non-preemptive semantics, we derive ①, the equivalence between $S_1 \parallel \ldots \parallel S_n$ and $S_1 \mid \ldots \mid S_n$, the latter representing non-preemptive execution of the threads. Similarly, *if we have* $\mathsf{DRF}(C_1 \parallel \ldots \parallel C_n)$ (at the top right), we can derive ②. With ① and ②, we can derive the semantics preservation $\approx$ between preemptive programs from the semantics preservation between their non-preemptive counterparts.

Second, DRF of the target programs is obtained through the right path ⑥,⑦ and ⑧. We define a notion of DRF for non-preemptive programs (called NPDRF in the figure), making it equivalent to DRF, from which we can derive ⑥ and ⑧. To know $\mathsf{NPDRF}(C_1 \mid \ldots \mid C_n)$ from $\mathsf{NPDRF}(S_1 \mid \ldots \mid S_n)$, we need a DRF-preserving simulation $\preccurlyeq$ between non-preemptive programs (see ⑦).

Third, the DRF-preserving simulation $\preccurlyeq$ for non-preemptive whole programs can be derived by composing our footprint-preservation local simulation (step ⑤). Given the the downward whole-program simulation, we flip it to get an *upward* one (step ④), with the extra premise that the local execution in each target module is deterministic. Using the simulation in both directions we derive the equivalence (step ③).

Note that the notations used here are simplified ones to give a semi-formal overview of the key ideas. We may use different notations in the formal development in the following sections.

3 THE LANGUAGE AND THE PREEMPTIVE SEMANTICS

3.1 The Abstract Language

Figure 3 shows the syntax and the state model of an abstract language for preemptive concurrent programming. A program P consists of n threads running in parallel. Each thread starts execution from an entry f, which points to the code segment defined in the code π of a module declared in Π. Each module declaration is a triple consisting of the language declaration tl, the global environment ge containing the addresses of static global variables declared in the module, and the code π of the module. Here we use $\mathcal{P}(Addr)$ to represent the powerset of memory addresses $Addr$.

$$
\begin{array}{llll}
(\mathit{Prog}) & P, \mathbb{P} & ::= & \textbf{let } \Pi \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \\
(\mathit{MdSet}) & \Pi, \Gamma & ::= & \{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\} \\
(\mathit{Lang}) & tl, sl & ::= & (\mathit{Module}, \mathit{Core}, \mathsf{InitCore}, \longmapsto) \\
 & ge & \in & \mathcal{P}(\mathit{Addr}) \\
 & \longmapsto & \in & \mathit{FList} \times (\mathit{Core} \times \mathit{State}) \rightarrow \{\mathcal{P}((\mathit{Msg} \times \mathit{FtPrt}) \times (\mathit{Core} \times \mathit{State})), \textbf{abort}\}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{Entry}) & \mathtt{f} & \in & \mathit{String} \\
(\mathit{Module}) & \pi, \gamma & ::= & \ldots \\
(\mathit{Core}) & \kappa, \Bbbk & ::= & \ldots \\
 & \mathsf{InitCore} & \in & \mathit{Module} \rightarrow \mathit{Entry} \rightharpoonup \mathit{Core}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{ThrdID}) & \mathtt{t} & \in & \mathbb{N} \\
(\mathit{State}) & \sigma, \Sigma & \in & \mathit{Addr} \rightharpoonup_{\mathsf{fin}} \mathit{Val} \\
(\mathit{FtPrt}) & \delta, \Delta & ::= & (rs, ws) \quad \text{where } rs, ws \in \mathcal{P}(\mathit{Addr}) \\
(\mathit{Msg}) & \iota & ::= & \tau \mid e \mid \mathtt{ret} \mid \mathsf{EntAtom} \mid \mathsf{ExtAtom} \\
(\mathit{Config}) & \phi, \Phi & ::= & (\kappa, \sigma) \mid \textbf{abort}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{Addr}) & l & ::= & \ldots \\
(\mathit{Val}) & v & ::= & l \mid \ldots \\
(\mathit{FList}) & F, \mathbb{F} & \in & \mathcal{P}^{\omega}(\mathit{Addr}) \\
(\mathit{Event}) & e & ::= & \ldots
\end{array}
$$

Fig. 3. The Abstract Concurrent Language

Since different modules may be written in different languages, we define the abstract module language *tl* as a tuple (*Module*, *Core*, InitCore, $\longmapsto$). *Module* describes the syntax of programs. Following Compositional CompCert [Stewart et al. 2015], *Core* is the set of internal "core" states, which can be instantiated to control continuations, instruction streams, register files, etc.The function InitCore is called whenever a thread is created or a cross-module external function call is made. Given a module π and an entry name f, InitCore returns the initial "core" state κ (which is undefined if the entry is not contained in the module). In this paper we mainly focus on compositional compilation of concurrent programs, where threads can be from different modules. Since cross-module external calls are mostly orthogonal, we omit them to simplify the presentation. *We do support external calls in our Coq implementation in the same way as in Compositional CompCert.* The labelled transition "$\longmapsto$" models the local execution of a module, which we explain below. To instantiate a language *tl*, one needs to provide concrete definitions for each component described above.

Module-local semantics. The local execution steps inside a module is modeled as a labeled transition in the form of $F \vdash_{tl} (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}} (\kappa', \sigma')$ (or $F \vdash_{tl} (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}}$ **abort** if the step goes wrong). In addition to its core state κ, each module can also access the memory state σ, which is a *finite* partial mapping from memory addresses to values. We leave the meta-type *Addr* undefined, which can be instantiated for specific languages. For instance, each address in the Cminor memory model can be instantiated into a pair of a block number and an offset. Each step may change the core state and the memory state into κ' and σ' respectively. It is also labeled with a message ι and a footprint δ. Note that the step relation can be *non-deterministic*. That is, given a pair (κ, σ), there can be more than one resulting states (κ', σ'), and the corresponding messages ι and footprints δ can be different. *To avoid clutter, we usually omit the parameter tl in the judgment.*

Each module also has a *free list* F. It is the pool of memory addresses from which fresh memory cells are allocated. It can be viewed as the preserved space for allocating local stack frames, where "$F - \mathrm{dom}(\sigma)$" is the set of free addresses, shown in Fig. 4 as the part outside of the boundary of σ. Initially we require $F \cap \mathrm{dom}(\sigma) = \emptyset$, and the only memory accessible by the module is the static global variables declared in the *ge* of all modules, represented as the *shared* part S in Fig. 4. The local execution of a module may allocate memory from its F, which enlarges the state σ. The memory allocated from

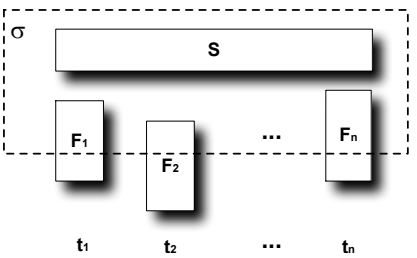


Fig. 4. The state model

F is exclusively owned by this module. We allow the set F to be *infinite*. F for different modules must be *disjoint*.

The messages ι contain information about the module-local steps. To simplify the presentation, we only consider externally observable events e (such as outputs), termination of threads (`ret`), and the beginning and the end of *atomic blocks* (EntAtom and ExtAtom). Any other steps are silent, labeled with τ. The label τ is often omitted for cleaner presentation. Atomic blocks are the language constructs to ensure sequential execution of code blocks inside them. They can be implemented differently in different module languages. In the example given in Sec. 8.1 we instantiate atomic blocks with `lock` prefixed instructions in x86. The messages define the protocols of communications with the global whole-program semantics (presented below). All the module languages use the same message formats. They allow us to abstract away details of the module languages, and focus on the interactions with other modules and the external observer (the latter observes e only).

The footprint δ is defined as a pair (rs, ws), which records the memory locations *read* and *written* in this step. Recording the footprint allows us to discuss races between threads in the following sections. We write `emp` for the special footprint where both the read and write sets are empty. Below we may directly use δ as a set, which is a shorthand for $\delta.rs \cup \delta.ws$.

Conventions. We use two sets of symbols to distinguish the source and the target level notations. The blackboard bold or capital letters (e.g., $\mathbb{P}$, $\mathbb{F}$, $\Bbbk$ and Σ) are used for the source, while their counterparts (e.g., P, F, κ and σ) are for the target. The set of modules at source is written as Γ, to distinguish from the target Π. Similarly, γ is a source module while π is a target one.

Well-defined languages. Although the abstract module language *tl* can be instantiated with different real languages, the instantiation needs to satisfy certain basic requirements. We define these requirements as *well-defined languages* below in Def. 1. It gives us an extensional interpretation of footprints. It is also used to prove properties of DRF programs in the sections below.

Definition 1 (Well-Defined Languages). $\mathrm{wd}(tl)$ iff , for any execution step $F \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\mapsto}} (\kappa', \sigma')$ in this language, all of the following hold (some auxiliary definitions are in Fig. 5):

(1) $\mathrm{forward}(\sigma, \sigma')$;
(2) $\mathtt{LEffect}(\sigma, \sigma', \delta, F)$
(3) for any σ_1, if $\mathtt{LEqPre}(\sigma, \sigma_1, \delta, F)$, then there exists σ_1' such that $F \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\mapsto}} (\kappa', \sigma_1')$ and $\mathtt{LEqPost}(\sigma', \sigma_1', \delta, F)$.
(4) for any δ_0 and σ_1, if $\delta_0 = \bigcup\{\delta'' \mid \exists \kappa'', \sigma''. F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta''}{\mapsto}} (\kappa'', \sigma'')\}$ and $\mathtt{LEqPre}(\sigma, \sigma_1, \delta_0, F)$, then $\forall \kappa_1'', \sigma_1'', \iota_1'', \delta_1''. F \vdash (\kappa, \sigma_1) \overset{\iota_1''}{\underset{\delta_1''}{\mapsto}} (\kappa_1'', \sigma_1'') \implies \exists \sigma''. F \vdash (\kappa, \sigma) \overset{\iota_1''}{\underset{\delta_1''}{\mapsto}} (\kappa_1'', \sigma'')$.

It requires that a step may enlarge the memory domain but cannot reduce it (see Item (1), where $\mathrm{forward}(\sigma, \sigma')$ is defined in Fig. 5), and the additional memory should be allocated from F and included in the write set (as required in Item (2)). This requirement follows the CompCert memory model where memory disposal does not really remove the locations from the memory (they just become invalid). Item (2) also requires that the memory out of the write set should keep unchanged, as described by $\sigma \overset{\mathrm{dom}(\sigma) - \delta.ws}{=\!=\!=\!=\!=\!=\!=\!=} \sigma'$ (which is defined at the top of Fig. 5). Item (3) says the memory updates and allocation only depend on the memory content in the read set, and the set of memory locations already allocated from F. Item (4) requires that the non-determinism of each step is not affected by memory contents outside of all the possible read sets. Here δ_0 is the union of footprints in all possible steps. Then for any state σ_1 with the same contents in the read set of δ_0 and the same set of allocated addresses as in σ, it does not generate any new behavior that cannot be generated

$$\sigma \overset{rs}{=\!=} \sigma' \quad \text{iff} \quad \forall l \in rs.\ l \notin (\mathsf{dom}(\sigma) \cup \mathsf{dom}(\sigma')) \ \vee\ l \in (\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma')) \wedge \sigma(l) = \sigma'(l)$$

$$\delta \subseteq \delta' \quad \text{iff} \quad (\delta.rs \subseteq \delta'.rs) \wedge (\delta.ws \subseteq \delta'.ws) \qquad \delta \cup \delta' \overset{\text{def}}{=} (\delta.rs \cup \delta'.rs,\ \delta.ws \cup \delta'.ws)$$

$$\mathsf{forward}(\sigma, \sigma') \ \text{iff} \ (\mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma'))$$

$$\mathtt{LEqPre}(\sigma_1, \sigma_2, \delta, F) \quad \text{iff} \quad \sigma_1 \overset{\delta.rs}{=\!=} \sigma_2 \wedge (\mathsf{dom}(\sigma_1) \cap \delta.ws) = (\mathsf{dom}(\sigma_2) \cap \delta.ws) \wedge (\mathsf{dom}(\sigma_1) \cap F) = (\mathsf{dom}(\sigma_2) \cap F)$$

$$\mathtt{LEqPost}(\sigma_1', \sigma_2', \delta, F) \quad \text{iff} \quad \sigma_1' \overset{\delta.ws}{=\!=} \sigma_2' \ \wedge\ (\mathsf{dom}(\sigma_1') \cap F) = (\mathsf{dom}(\sigma_2') \cap F)$$

$$\mathtt{LEffect}(\sigma_1, \sigma_2, \delta, F) \quad \text{iff} \quad \sigma_1 \overset{\mathsf{dom}(\sigma_1) - \delta.ws}{=\!=\!=\!=} \sigma_2 \wedge (\mathsf{dom}(\sigma_2) - \mathsf{dom}(\sigma_1)) \subseteq (\delta.ws \cap F)$$

Fig. 5. Notations and auxiliary definitions about states and footprints

by σ. This property ensures that, for data-race-free programs, the possible execution steps of a thread cannot be affected by the interleaving with other threads.

3.2 Global preemptive semantics

$$(\textit{World})\ \ W, \mathbb{W} ::= (T, \mathtt{t}, d, \sigma) \qquad (\textit{AtomBit})\ \ d ::= 0 \mid 1$$

$$(\textit{ThrdPool})\ \ T, \mathbb{T} ::= \{\mathtt{t}_1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, \mathtt{t}_n \rightsquigarrow (tl_1, F_n, \kappa_n)\}$$

$$(\textit{GMsg})\ \ o ::= \tau \mid e \mid \mathtt{sw} \qquad (\textit{ASet})\ \ S, \mathbb{S} \in \mathcal{P}(\textit{Addr})$$

Fig. 6 defines a set of global semantics rules to manipulate the preemption among threads. As shown above, the global world W consists of the thread pool T, the ID $\mathtt{t}$ of the thread currently being executed, a bit d indicating whether the current thread is in an atomic block or not, and the memory state σ. The thread pool T maps a thread ID to a triple recording the module language tl, the free list F, and the current core state κ.

The Load rule in Fig. 6 shows the initialization of the world from the program and an initial program state σ. We assume the code π of modules in Π has disjoint entries (i.e. code labels). Therefore, for each entry $\mathtt{f}_i$ of the thread i, we can find at most one module declaration in Π containing it. The core state κ_i is created through the InitCore function of the corresponding language. For each thread, we also assign a local address space F for allocation of stack frames. The local address spaces for the threads must be disjoint, and initially they are disjoint with the domain of σ. We non-deterministically pick a thread $\mathtt{t}$ as the current thread. The bit d is set to 0, indicating that the current thread is *not* in the atomic block.

The transition rules of the whole world is given in Fig. 6. Like local transitions, global transitions are also labeled with footprints and messages o. Here o marks a silent τ step, a step with external event e, or a switch (sw) step, as defined above. Different from the messages ι that record both external events and inter-module communications, the global messages o only record the externally observable events, i.e., events that are observable to the human being sitting in front of the computer or printer. The switch message sw is an exception, which is for verification purpose only.

Each global step executes the module locally and processes the message of the local transition. The τ-step rule and the Print rule show the local τ-step and the step generating an external event, respectively. Note that the Print rule requires that the flag d must be 0, i.e., external events can be generated only outside of atomic blocks. Also the step generating an external event does *not* access memory, so its footprint is empty (emp).

The EntAt and ExtAt rules correspond to the entry and exit of atomic blocks, respectively. The flag d is flipped in the two steps. Since context-switch can be done only when d is 0, as required by the Switch rule below, we know a thread in its atomic block cannot be preempted.

When the current thread terminates, we either remove it from the thread pool and then switch to another thread if there is one (see the Term rule), or terminate the whole program if not, as shown in the Done rule. The Term rule generates the sw message to record the context switch.

$$\frac{\begin{array}{l}\text{for all } i \text{ and } j \text{ in } \{1,\ldots,n\}, \text{ and } i \neq j\text{:}\\ F_i \cap F_j = \emptyset \quad \mathsf{dom}(\sigma) \cap F_i = \emptyset \qquad tl_i.\mathsf{InitCore}(\pi_i, \mathtt{f}_i) = \kappa_i, \ \text{ where } (tl_i, ge_i, \pi_i) \in \Pi\\ T = \{1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, n \rightsquigarrow (tl_n, F_n, \kappa_n)\} \qquad \mathtt{t} \in \{1,\ldots,n\}\end{array}}{(\textbf{let } \Pi \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \sigma) \overset{load}{\Longrightarrow} (T, \mathtt{t}, 0, \sigma)}\ \textsf{Load}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}} (\kappa', \sigma') \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, d, \sigma) \overset{\tau}{\underset{\delta}{\Rightarrow}} (T', \mathtt{t}, d, \sigma')}\ \tau\textsf{-step} \qquad \frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \overset{e}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, 0, \sigma) \overset{e}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}, 0, \sigma)}\ \textsf{Print}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, 0, \sigma) \overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}, 1, \sigma)}\ \textsf{EntAt} \qquad \frac{T(\mathtt{t}) = (tl, F, \kappa) \quad \mathtt{t}' \in \mathsf{dom}(T \backslash \mathtt{t}) \quad F \vdash (\kappa, \sigma) \overset{\mathtt{ret}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \overset{\mathsf{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T \backslash \mathtt{t}, \mathtt{t}', 0, \sigma)}\ \textsf{Term}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \overset{\mathsf{ExtAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, 1, \sigma) \overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}, 0, \sigma)}\ \textsf{ExtAt} \qquad \frac{T(\mathtt{t}) = (tl, F, \kappa) \quad \mathsf{dom}(T) = \{\mathtt{t}\} \quad F \vdash (\kappa, \sigma) \overset{\mathtt{ret}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} \textbf{done}}\ \textsf{Done}$$

$$\frac{\mathtt{t}' \in \mathsf{dom}(T)}{(T, \mathtt{t}, 0, \sigma) \overset{\mathsf{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T, \mathtt{t}', 0, \sigma)}\ \textsf{Switch} \qquad \frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{}{\underset{\delta}{\longmapsto}} \textbf{abort}}{(T, \mathtt{t}, 0, \sigma) \overset{\tau}{\underset{\delta}{\Rightarrow}} \textbf{abort}}\ \textsf{Abort}$$

Fig. 6. The Preemptive Global Semantics

The Switch rule shows that context switch can occur at any program point outside of atomic blocks ($d = 0$). This also indicates that the semantics is preemptive. The step is also marked with the sw message. The Abort rule says the whole program aborts if a local module aborts.

Below we write $F \vdash \phi \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{+} \phi'$ for multiple silent-step transitions, where δ is the accumulation of the footprints generated. $F \vdash \phi \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{*} \phi'$ is for zero or multiple silent-step transitions, where δ is emp for the case of zero step. Similarly, for global steps, we write $W \overset{\tau}{\underset{\delta}{\Rightarrow}}{}^{+} W'$ for multiple silent-step transitions. Besides, we also write $W \Rightarrow^{+} W'$ for multiple steps that either are silent or produce sw events. It must contain at least one silent step. The meanings of $W \Rightarrow^{+} \textbf{abort}$ and $W \Rightarrow^{+} \textbf{done}$ are similar. $W \overset{e}{\Rightarrow}{}^{+} W'$ represents multiple steps with exactly one e event produced (while other steps either are silent or produce sw events).

3.3 Event-Trace Refinement and Equivalence

The correctness of compilation for concurrent programs is defined as the event-trace refinement (or equivalence) between source and target programs. An externally observable event trace $\mathcal{B}$ is a finite or infinite sequence of external events e, and may end with a termination marker **done** or an abortion marker **abort**. It is co-inductively defined as follows.

$$(\mathit{EvtTrace}) \quad \mathcal{B} \ ::= \ \textbf{done} \mid \textbf{abort} \mid \epsilon \mid e :: \mathcal{B} \quad \text{(co-inductive)}$$

$$\mathit{ProgEtr}((P, \sigma), \mathcal{B}) \ \text{ iff } \ \exists W.\, ((P, \sigma) \overset{load}{\Longrightarrow} W) \wedge \mathit{Etr}(W, \mathcal{B})$$

$$\frac{W \Rightarrow^{+} \textbf{abort}}{\mathit{Etr}(W, \textbf{abort})} \qquad \frac{W \Rightarrow^{+} \textbf{done}}{\mathit{Etr}(W, \textbf{done})} \qquad \frac{W \overset{e}{\Rightarrow}{}^{+} W' \qquad \mathit{Etr}(W', \mathcal{B})}{\mathit{Etr}(W, e :: \mathcal{B})} \qquad \frac{W \Rightarrow^{+} W' \qquad \mathit{Etr}(W', \epsilon)}{\mathit{Etr}(W, \epsilon)}$$

We use $ProgEtr((P,\sigma),\mathcal{B})$ to say that the trace $\mathcal{B}$ can be produced by executing P with the initial state σ. The co-inductive definition of $Etr(W,\mathcal{B})$ says that $\mathcal{B}$ can be produced by executing W. Note we distinguish the traces of non-terminating (diverging) executions from those of terminating ones. If the execution of W diverges, its observable event trace $\mathcal{B}$ is either of infinitely length, or finite but does not end with **done** or **abort** (called *silent divergence*, see the right-most rule above).

Then we define the refinement $(\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)$ and the equivalence $(\mathbb{P},\Sigma) \approx (P,\sigma)$ below. They ensure that if (P,σ) has a diverging execution, so does $(\mathbb{P},\Sigma)$. Thus the refinement and the equivalence relations preserve the termination of $(\mathbb{P},\Sigma)$.

Definition 2 (Event-Trace Refinement and Equivalence)**.**
$(\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)$ iff $\forall \mathcal{B}.\ ProgEtr((P,\sigma),\mathcal{B}) \implies ProgEtr((\mathbb{P},\Sigma),\mathcal{B})$.
$(\mathbb{P},\Sigma) \approx (P,\sigma)$ iff $\forall \mathcal{B}.\ ProgEtr((P,\sigma),\mathcal{B}) \iff ProgEtr((\mathbb{P},\Sigma),\mathcal{B})$.

Following CompCert, the compilation correctness assumes safety of the source programs. Below we use the event traces to define Safe($\mathbb{W}$) and Safe($\mathbb{P},\Sigma$).

Definition 3 (Safety)**.** Safe($\mathbb{W}$) iff $\neg\exists tr.\ Etr(\mathbb{W}, tr::\mathbf{abort})$.
Safe($\mathbb{P},\Sigma$) iff $(\exists \mathbb{W}.\ (\mathbb{P},\Sigma) \overset{load}{\Longrightarrow} \mathbb{W})$ and $\forall \mathbb{W}.\ ((\mathbb{P},\Sigma) \overset{load}{\Longrightarrow} \mathbb{W}) \implies$ Safe($\mathbb{W}$).

4 THE NON-PREEMPTIVE SEMANTICS

A key step in our framework is to reduce the semantics preservation under the preemptive semantics to the semantics preservation in non-preemptive semantics. In this section we define the global non-preemptive semantics, where a thread interacts with other threads at only synchronization points (i.e., when it enters and exits atomic blocks, and outputs). The non-preemptive semantics is the basis for both our new simulation (see Sec. 5) and our NPDRF definition (see Sec. 6).

$$(NPProg)\quad \hat{P} ::= \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \qquad (NPWorld)\quad \widehat{W}, \widehat{\mathbb{W}} ::= (T, \mathtt{t}, \mathbb{d}, \sigma)$$
$$(AtomBitMap)\quad \mathbb{d} ::= \{\mathtt{t}_1 \rightsquigarrow d_1, \ldots, \mathtt{t}_n \rightsquigarrow d_n\}$$

To distinguish from the preemptive concurrency, we write **let** Π **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ for the program with non-preemptive semantics, denoted by $\hat{P}$. As shown above, the non-preemptive global world $\widehat{W}$ is defined similarly as the preemptive world W, except that $\widehat{W}$ keeps an atomic bit map $\mathbb{d}$ recording whether each thread's next step is inside an atomic block. We need to record the atomic bits of all threads because the context switch may occur when a thread just enters an atomic block.

Fig. 7 defines the non-preemptive global steps $\widehat{W} \overset{o}{\underset{\delta}{:\Rightarrow}} \widehat{W}'$. There is no rule like Switch of the preemptive semantics, since context switch occurs only at synchronization points in the non-preemptive setting. The rules $\mathsf{Print}_{\mathsf{np}}$, $\mathsf{EntAt}_{\mathsf{np}}$, $\mathsf{ExtAt}_{\mathsf{np}}$ and $\mathsf{Term}_{\mathsf{np}}$ execute one step of the current thread $\mathtt{t}$, and then non-deterministically switch to a thread $\mathtt{t}'$ (which could just be $\mathtt{t}$). The corresponding global steps produce the $\mathtt{sw}$ events (or the external event e in the ($\mathsf{Print}_{\mathsf{np}}$) rule). Note that in the $\mathsf{EntAt}_{\mathsf{np}}$ rule, a thread may switch before it executes the body of the atomic block. Thus the global step needs to set the corresponding atomic bit in $\mathbb{d}$, indicating that the thread must be inside an atomic block when it regains the control later. Other rules are very similar to their counterparts in the preemptive semantics in Fig. 6, and are not presented here.

Similar to Def. 2, we define $(\hat{\mathbb{P}},\Sigma) \sqsupseteq (\hat{P},\sigma)$ and $(\hat{\mathbb{P}},\Sigma) \sqsupseteq (\hat{P},\sigma)$ for the refinement and equivalence between programs under non-preemptive semantics.

$$\frac{\begin{array}{c}\text{for all } i \text{ and } j \text{ in } \{1,\ldots,n\}, \text{ and } i \neq j\text{:}\\ F_i \cap F_j = \emptyset \quad \mathrm{dom}(\sigma) \cap F_i = \emptyset \quad tl_i.\mathsf{InitCore}(\pi_i, \mathtt{f}_i) = \kappa_i, \quad \text{where } (tl_i, ge_i, \pi_i) \in \Pi\\ T = \{1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, n \rightsquigarrow (tl_n, F_n, \kappa_n)\} \quad \mathtt{t} \in \{1,\ldots,n\} \quad \mathbb{d} = \{\mathtt{t}_1 \rightsquigarrow 0, \ldots, \mathtt{t}_n \rightsquigarrow 0\}\end{array}}{(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \sigma) :\overset{load}{\Longrightarrow} (T, \mathtt{t}, \mathbb{d}, \sigma)}\ \mathsf{Load}_{\mathsf{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0\\ F \vdash (\kappa, \sigma) \overset{e}{\underset{\mathsf{emp}}{\mapsto}} (\kappa', \sigma) \qquad \mathtt{t}' \in \mathrm{dom}(T)\\ T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{e}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}, \sigma)}\ \mathsf{Print}_{\mathsf{np}} \qquad \frac{\begin{array}{c}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0\\ F \vdash (\kappa, \sigma) \overset{\mathsf{ret}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \qquad \mathtt{t}' \in \mathrm{dom}(T \backslash \mathtt{t})\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{\mathsf{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T \backslash \mathtt{t}, \mathtt{t}', \mathbb{d} \backslash \mathtt{t}, \sigma)}\ \mathsf{Term}_{\mathsf{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0\\ F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)\\ T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\} \quad \mathtt{t}' \in \mathrm{dom}(T)\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{\mathsf{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 1\}, \sigma)}\ \mathsf{EntAt}_{\mathsf{np}} \qquad \frac{\begin{array}{c}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 1\\ F \vdash (\kappa, \sigma) \overset{\mathsf{ExtAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)\\ T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\} \quad \mathtt{t}' \in \mathrm{dom}(T)\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{\mathsf{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 0\}, \sigma)}\ \mathsf{ExtAt}_{\mathsf{np}}$$

Fig. 7. The Non-Preemptive Global Semantics

5 THE FOOTPRINT-PRESERVING COMPOSITIONAL SIMULATION

In this section, we define a module-local simulation as the correctness obligation of each module's compilation, which is compositional and preserves footprints, allowing us to derive a whole-program simulation that preserves data-race-freedom. We will discuss compositionality in Sec. 5.2 and postpone the discussions of DRF and NPDRF preservation to Sec. 6.

5.1 The Module-Local Simulation

As informally explained in Sec. 2, the simulation establishes a *consistency relation* between executions of the source module γ and the target one π. To achieve compositionality, our simulation uses rely/guarantee conditions to specify the interactions between the current module and its environment at switch points. The consistency relation should be preserved under the environment transitions allowed in the rely condition. So the key to define the local simulation is to define a proper consistency relation and proper rely/guarantee conditions.

Footprint consistency. As in CompCert, the consistency relation require that the source and the target generate the same external events. In addition, we also require that the target has the same or smaller footprints than the source, which is important to ensure DRF-preservation. Recall that the memory accessible by a thread $\mathtt{t}_i$ consists of two parts, the shared memory $\mathbb{S}$ and the local memory allocated from $\mathbb{F}_i$, as shown in Fig. 4. DRF informally requires that the threads never access the shared memory in $\mathbb{S}$ at the same time where at least one such access is a write.

We introduce the triple μ below to record the key information about the shared memory at the source and the target.

$$\mu \overset{\text{def}}{=} (\mathbb{S}, S, f) \quad \text{where } \mathbb{S}, S \in \mathcal{P}(Addr) \text{ and } f \in Addr \rightharpoonup Addr$$

Here $\mathbb{S}$ and S specify the shared memory locations at the source and the target respectively. The partial mapping f maps locations at the source level to those at the target. We require μ to be well-formed, defined as $\mathsf{wf}(\mu)$ in Fig. 8. It says that the domain of f is $\mathbb{S}$, f is injective, and maps shared locations (in $\mathbb{S}$) to shared locations (in S). Here $f\{\!|\mathbb{S}|\!\}$ returns the set of target locations that are mapped from locations in $\mathbb{S}$.

$$f\{\mathbb{S}\} \stackrel{\text{def}}{=} \{l' \mid \exists l.\ (l \in \mathbb{S}) \wedge f(l) = l'\} \qquad f|_{\mathbb{S}} \stackrel{\text{def}}{=} \{(l, f(l)) \mid l \in (\mathbb{S} \cap \text{dom}(f))\}$$

$$\text{wf}(\mu) \ \text{ iff } \ \text{injective}(f) \wedge \text{dom}(f) = \mathbb{S} \wedge f\{\mathbb{S}\} \subseteq S \qquad \text{where } \mu = (\mathbb{S}, S, f)$$

$$\text{FPmatch}(\mu, \Delta, \delta) \ \text{ iff } \ (\delta.rs \cap S \subseteq f\{\Delta.rs\}) \wedge (\delta.ws \cap S \subseteq f\{\Delta.ws\}) \qquad \text{where } \mu = (\mathbb{S}, S, f)$$

$$\text{closed}(\mathbb{S}, \Sigma) \ \text{ iff } \ \text{cl}(\mathbb{S}, \Sigma) \subseteq \mathbb{S}$$

$$\text{cl}(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \bigcup_k \text{cl}_k(\mathbb{S}, \Sigma)\,, \ \text{where } \text{cl}_k(\mathbb{S}, \Sigma) \text{ is inductively defined:}$$

$$\text{cl}_0(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \mathbb{S} \qquad \text{cl}_{k+1}(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \{l' \mid \exists l.\ (l \in \text{cl}_k(\mathbb{S}, \Sigma)) \wedge \Sigma(l) = l'\}$$

$$\text{initM}(\varphi, ge, \Sigma, \sigma) \ \text{ iff } \ ge \subseteq \text{dom}(\Sigma) \wedge \text{closed}(\text{dom}(\Sigma), \Sigma) \wedge \text{dom}(\sigma) = \varphi\{\text{dom}(\Sigma)\} \wedge \text{Inv}(\varphi, \Sigma, \sigma)$$

$$\text{HG}(\Delta, \Sigma', \mathbb{F}, \mathbb{S}) \ \text{ iff } \ \Delta \subseteq (\mathbb{F} \cup \mathbb{S}) \wedge \text{closed}(\mathbb{S}, \Sigma')$$

$$\text{LG}(\mu, (\delta, \sigma', F), (\Delta, \Sigma')) \ \text{ iff } \ \delta \subseteq (F \cup \mu.S) \wedge \text{closed}(\mu.S, \sigma') \wedge \text{FPmatch}(\mu, \Delta, \delta) \wedge \text{Inv}(\mu.f, \Sigma', \sigma')$$

$$\text{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S}) \ \text{ iff } \ (\Sigma \stackrel{\mathbb{F}}{=\!=} \Sigma') \wedge \text{closed}(\mathbb{S}, \Sigma') \wedge \text{forward}(\Sigma, \Sigma')$$

$$\text{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F)) \ \text{ iff } \ \text{R}(\Sigma, \Sigma', \mathbb{F}, \mu.\mathbb{S}) \wedge \text{R}(\sigma, \sigma', F, \mu.S) \wedge \text{Inv}(\mu.f, \Sigma', \sigma')$$

$$\text{Inv}(f, \Sigma, \sigma) \ \text{ iff } \ \forall l, l'.\ (l \in \text{dom}(\Sigma) \wedge f(l) = l') \Longrightarrow (l' \in \text{dom}(\sigma) \wedge \Sigma(l) \stackrel{f}{\hookrightarrow} \sigma(l'))$$

$$v_1 \stackrel{f}{\hookrightarrow} v_2 \ \text{ iff } \ (v_1 \notin \mathit{Addr}) \wedge (v_1 = v_2) \ \vee \ (v_1, v_2 \in \mathit{Addr} \wedge f(v_1) = v_2)$$

Fig. 8. Footprint Matching and Rely/Guarantee Conditions in Our Simulation

Then, given footprints Δ and δ, we define their consistency with respect to μ as $\text{FPmatch}(\mu, \Delta, \delta)$ in Fig. 8. It says the *shared* locations in δ must be contained in Δ, modulo the mapping $\mu.f$. We only consider the shared locations in $\mu.S$ because accesses of local memory would *not* cause races.

Rely/guarantee conditions. We use rely and guarantee conditions to specify the module interaction protocols. One of the most important protocol is to enforce the view of accessibility of shared and local memory in Fig. 4. More specifically, when the execution of a module switches to external modules, it expects them to keep its local memory (in $\mathbb{F}$) intact. In addition, although the external modules may update the shared memory $\mathbb{S}$, they must preserve certain properties of $\mathbb{S}$. One important property is that $\mathbb{S}$ cannot contain memory pointers pointing to local memory cells in any $\mathbb{F}_i$[1]. Otherwise a thread $\mathtt{t}_j$ can update the local memory in $\mathbb{F}_i$ by tracing the pointers in S. This requirement is formalized as $\text{closed}(\mathbb{S}, \Sigma)$ in Fig. 8, which says the closure of addresses reachable from $\mathbb{S}$ must be no bigger than $\mathbb{S}$. We encode these requirements in the rely condition, and guarantee conditions are defined correspondingly to ensure the rely is satisfied.

We define the module-local downward simulation $(sl, ge, \gamma) \preccurlyeq_\varphi (tl, ge', \pi)$ below, which relates the executions of the source module (sl, ge, γ) and the target module (tl, ge', π). The injective function φ maps source addresses to the target ones.

Definition 4 (Module-Local Downward Simulation)**.**
$(sl, ge, \gamma) \preccurlyeq_\varphi (tl, ge', \pi)$ iff for all $\mathtt{f}, \Bbbk, \Sigma, \sigma, \mathbb{F}, F$, and $\mu = (\text{dom}(\Sigma), \text{dom}(\sigma), \varphi|_{\text{dom}(\Sigma)})$,
if $sl.\text{InitCore}(\gamma, \mathtt{f}) = \Bbbk$, $\varphi\{ge\} = ge'$, $\text{initM}(\varphi, ge, \Sigma, \sigma)$, and $\mathbb{F} \cap \text{dom}(\Sigma) = F \cap \text{dom}(\sigma) = \emptyset$,
then there exist κ and $i \in \mathtt{index}$ such that $tl.\text{InitCore}(\pi, \mathtt{f}) = \kappa$, and
$(\mathbb{F}, (\Bbbk, \Sigma), \mathtt{emp}) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \mathtt{emp})$,
where $(\mathbb{F}, (\Bbbk, \Sigma), \Delta) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta)$, is defined in Def. 5 below.

It says that if we take any function entry $\mathtt{f}$ and the corresponding initial core states $\Bbbk$ and κ at the source and the target respectively, then with any states (Σ and σ) and free lists ($\mathbb{F}$ and F) satisfying

[1] This means we do not allow escape of pointers pointing to stack variables.

some initial constraints, we can establish the simulation $(\mathbb{F}, (\Bbbk, \Sigma), \mathsf{emp}) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \mathsf{emp})$, relating the local module configurations, which is defined in Def. 5 and is explained below. Here we require the initial states Σ and σ be related through initM defined in Fig. 8. The last condition in initM is the invariant Inv defined at the bottom of Fig. 8. It says that the location $f(l)$ must be in $\mathsf{dom}(\sigma)$ if l is in $\mathsf{dom}(\Sigma)$, and the memory contents $\Sigma(l)$ and $\sigma(f(l))$ must be equal modulo $\mu.f$.

Definition 5. We define $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$ as the largest relation such that, whenever $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$, then the following are true:

(1) for all $\Bbbk'$, Σ' and Δ, if $\mathbb{F} \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\mapsto}} (\Bbbk', \Sigma')$ and $(\Delta_0 \cup \Delta) \subseteq (\mathbb{F} \cup \mu.\mathbb{S})$,
  then one of the following holds:
  (a) $\exists j < i.\ (\mathbb{F}, (\Bbbk', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (F, (\kappa, \sigma), \delta_0)$, or
  (b) there exist κ', σ', δ and j such that the following are true:
    (i) $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^{+} (\kappa', \sigma')$;
    (ii) $(\delta_0 \cup \delta) \subseteq (F \cup \mu.S)$ and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0 \cup \delta)$; and
    (iii) $(\mathbb{F}, (\Bbbk', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (F, (\kappa', \sigma'), \delta_0 \cup \delta)$.

(2) for all $\Bbbk'$ and ι, if $\mathbb{F} \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\mathsf{emp}}{\mapsto}} (\Bbbk', \Sigma)$, $\iota \neq \tau$, and $\mathsf{HG}(\Delta_0, \Sigma, \mathbb{F}, \mu.\mathbb{S})$,
  then there exist κ', δ, σ' and κ'' such that the following are true:
  (a) $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^{*} (\kappa', \sigma')$, and $F \vdash (\kappa', \sigma') \overset{\iota}{\underset{\mathsf{emp}}{\mapsto}} (\kappa'', \sigma')$, and
  (b) $\mathsf{LG}(\mu, (\delta_0 \cup \delta, \sigma', F), (\Delta_0, \Sigma))$, and
  (c) for all σ'' and Σ', if $\mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma', \sigma'', F))$, then there exists j such that
    $(\mathbb{F}, (\Bbbk', \Sigma'), \mathsf{emp}) \preccurlyeq^j_\mu (F, (\kappa'', \sigma''), \mathsf{emp})$.

The simulation $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$ carries Δ_0 and δ_0, the footprints accumulated until now at the source and the target, respectively. The definition follows the diagram in Fig. 1(d). For every τ-step in the source (case 1), if the newly generated footprints and the accumulated Δ_0 are *in scope* (i.e. every location must either be from the freelist space $\mathbb{F}$ of current thread, or from the shared memory $\mu.\mathbb{S}$), then the step should correspond to zero or multiple τ-steps in the target, and the simulation holds over the resulting states with the accumulated footprints and a new index j. Here we use Δ as a shorthand for $(\Delta.rs \cup \Delta.ws)$. If the source step corresponds to zero target step (case 1-a), the index j must be strictly smaller than i. Here the indices i and j belong to a well-founded set index that has no infinite decreasing sequences. The use of a smaller index j in this case ensures non-terminating source module can only be simulated by non-terminating target.

If the source step corresponds to at least one target steps (case 1-b), the index j can be arbitrary. In this case we require the footprints at the target are also in scope, and they must be consistent with the source level footprints (see our explanation of FPmatch before). The accumulation of footprints allows us to establish FPmatch for compiler optimizations that reorder the instructions.

At the switch points when the source generates a non-silent message ι (case 2), if the footprints and states satisfy HG, the target must be able to generate the same ι (after zero or multiple silent steps), and the accumulated footprints and the state satisfy LG. As defined in Fig. 8, both HG and LG require the footprints are in scope and the shared memory is closed. LG additionally requires the footprints at the target and the source satisfy FPmatch, and the states satisfy Inv.

One may wonder if it is possible to check FPmatch at the switch points only. However, executions of non-terminating modules would never reach a switch point. That is why we have to also require FPmatch during internal τ-steps (case 1).

At the switch points we also need to consider the interaction with the environment (i.e. other modules or threads). For any environment steps at the source and the target, if they satisfy the rely

condition, then the simulation holds over the new states, with some index j and *empty* footprints — Since the effects of the current thread have been made visible to the environments at the switch point, we can clear the accumulated footprints. The rely condition is defined in Fig. 8. As explained before, it requires the local memory is untouched, the shared memory is closed, and the domain of states is not reduced (see the definition of forward in Fig. 5). Also it requires that the invariant Inv be preserved over the new states at the source and the target.

Note that each case in Def. 5 has prerequisite about the source level footprints (e.g. the footprints are in scope or satisfy HG). On the one hand, this makes the simulation weak and easy to prove. We do not need to prove these requirements indeed hold for each compilation phase since they are premises. On the other hand, after we apply transitivity of the simulation and prove that the target generated after the multi-phase compilation simulates the source, we need to additionally prove that these requirements indeed hold at the source level, to make the simulation meaningful instead of being vacuously true. Following the approach by Stewart et al. [2015], we formalize these requirements separately as ReachClose (Def. 7) in the next subsection.

Our simulation is transitive. One can decompose the whole compiler correctness proofs into proofs for individual compilation phases.

Lemma 6 (Transitivity). $\forall sl, sl', tl, \gamma, \gamma', \pi.$
if $(sl, ge, \gamma) \preccurlyeq_{\varphi} (sl', ge', \gamma')$ and $(sl', ge', \gamma') \preccurlyeq_{\varphi'} (tl, ge'', \pi)$, then $(sl, ge, \gamma) \preccurlyeq_{\varphi' \circ \varphi} (tl, ge'', \pi)$.

5.2 Compositionality and the Non-Preemptive Global Simulation

The whole program *downward* simulation $\hat{\mathbb{P}} \preccurlyeq_{\varphi} \hat{P}$ relates the execution of the whole source program $\mathbb{P}$ and target program P. It serves as an intermediate result for proving event trace refinement and data-race-freedom preservation, as shown in Fig. 2.

Due to space limit, we omit the definition of the whole program simulation, which is similar to the module local simulation, except the case for environment interference (rely steps), which is unnecessary for whole program simulation. Every source step should correspond to multiple target steps. As in module local simulation, we always require the target footprints matches those of the source, defined as FPmatch. Also the footprint should always be in scope. As a special requirement for the whole program simulation, we always require the source and the target do lock-step context switch and they always switch to the same thread.

Compositionality. Following Liang et al. [2012], the module-local simulations could be composed to derive the whole-program simulation by proving the Rely condition of a module is guaranteed by other modules' guarantee conditions HG and LG. However, as explained at the end of Sec. 5.1, in our module-local simulation, LG is established if the source module satisfies HG. We require that HG always hold during the execution of the source, and formulate this requirement as ReachClose in Def. 7. It is a simplified version of the *reach-close* concept by Stewart et al. [2015] (simplified because we do not allow the leak of local stack pointers into the shared memory).

Definition 7 (Reach Closed Module). $\mathsf{ReachClose}(sl, ge, \gamma)$ iff , for all f, $\Bbbk$, Σ, $\mathbb{F}$ and $\mathbb{S}$,
if $sl.\mathsf{InitCore}(\gamma, \mathsf{f}) = \Bbbk$, $ge \subseteq \mathbb{S} = \mathsf{dom}(\Sigma)$, $\mathbb{F} \cap \mathbb{S} = \emptyset$, and $\mathsf{closed}(\mathbb{S}, \Sigma)$, then $\mathsf{RC}(\mathbb{F}, \mathbb{S}, (\Bbbk, \Sigma))$.

Here RC is defined as the largest relation such that, whenever $\mathsf{RC}(\mathbb{F}, \mathbb{S}, (\Bbbk, \Sigma))$, then for all Σ' such that $\mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S})$, and for all $\Bbbk'$, Σ', Σ'', ι and Δ such that $\mathbb{F} \vdash (\Bbbk, \Sigma') \overset{\iota}{\underset{\Delta}{\mapsto}} (\Bbbk', \Sigma'')$, we have $\mathsf{HG}(\Delta, \Sigma'', \mathbb{F}, \mathbb{S})$, and $\mathsf{RC}(\mathbb{F}, \mathbb{S}, (\Bbbk', \Sigma''))$.

The relation $\mathsf{RC}(\mathbb{F}, \mathbb{S}, (\Bbbk, \Sigma))$ essentially says during every step of the execution of $(\Bbbk, \Sigma)$, HG always holds over the resulting footprints Δ and states, even with possible interference from the environment, as long as the environment steps satisfy the rely condition R defined in Fig. 8.

Assuming all source modules are ReachClose, we can prove the Compositionality Lemma.

Lemma 8 (Compositionality, ⑤ in Fig. 2)**.** For any $\mathtt{f}_1, \ldots, \mathtt{f}_n, \varphi, \Gamma = \{(sl_1, ge_1\gamma_1), \ldots, (sl_m, ge_m\gamma_m)\}$, $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m\pi_m)\}$ if

$$\forall i \in \{1, \ldots, m\}.\, \mathsf{wd}(sl_i) \wedge \mathsf{wd}(tl_i) \wedge \mathsf{ReachClose}(sl_i, ge_i, \gamma_i) \wedge (sl_i, ge_i, \gamma_i) \preccurlyeq_\varphi (tl_i, ge'_i, \pi_i)\,,$$

then $\quad$ **let** Γ **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \preccurlyeq_\varphi$ **let** Π **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$.

5.3 Flip of the Non-Preemptive Global Simulation

As explained in Sec. 2.2, with determinism of the target modules, one is able to flip the downward simulation to derive upward simulation for safe source programs. Then the upward event trace refinement follows the upward simulation.

We define the whole program upward simulation as $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$. Due to space limit, we omit its concrete definition here. It is similar to the downward simulation $\hat{\mathbb{P}} \preccurlyeq_\varphi \hat{P}$, with the positions of source and target swapped. Note that we do *not* flip the FPmatch condition, since we always require footprint of target program being a refinement of the footprint of the source program, in order to prove DRF of the target program. Correspondingly, the address mapping φ is *not* flipped either.

Definition 9 (Deterministic Languages)**.** $\mathsf{det}(tl)$ iff, for all configuration ϕ in tl (see the definition of ϕ in Fig. 3), and for all F, $\quad F \vdash \phi \overset{\iota_1}{\underset{\delta_1}{\mapsto}} \phi_1 \wedge F \vdash \phi \overset{\iota_2}{\underset{\delta_2}{\mapsto}} \phi_2 \implies \phi_1 = \phi_2 \wedge \iota_1 = \iota_2 \wedge \delta_1 = \delta_2$.

With determinism of the target module languages, we are able to prove Lemma 10.

Lemma 10 (Flip, ④ in Fig. 2)**.** For any $\mathtt{f}_1, \ldots, \mathtt{f}_n, ge, \varphi, \Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, if $\forall i.\ \mathsf{det}(tl_i)$, and

$$\begin{array}{ll} & \textbf{let } \Gamma \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \preccurlyeq_\varphi \textbf{let } \Pi \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m\,, \\ \text{then} & \textbf{let } \Pi \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \leqslant_\varphi \textbf{let } \Gamma \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m\,. \end{array}$$

Soundness. The non-preemptive global simulation ensures the refinement. Before presenting the soundness lemma, we first lift the refinement $(P, \Sigma) \sqsubseteq (\mathbb{P}, \sigma)$ (see Def. 2) to $P \sqsubseteq_\varphi \mathbb{P}$ as follows. Similarly, $(\hat{P}, \Sigma) \sqsubseteq (\hat{\mathbb{P}}, \sigma)$ is lifted to to $\hat{P} \sqsubseteq_\varphi \hat{\mathbb{P}}$.

$$\begin{array}{l} P \sqsubseteq_\varphi \mathbb{P} \ \text{ iff } \ \forall \Sigma, \sigma.\, \mathsf{initM}(\varphi, \mathsf{GE}(\mathbb{P}.\Gamma), \Sigma, \sigma) \implies (P, \sigma) \sqsubseteq (\mathbb{P}, \Sigma) \\ \text{where } \mathsf{GE}(\{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\}) \overset{\text{def}}{=} \bigcup_{i=1}^{m} ge_i \end{array}$$

Lemma 11 (Soundness, ③ in Fig. 2)**.** If $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$, then $\hat{P} \sqsubseteq_\varphi \hat{\mathbb{P}}$.

6 DATA-RACE-FREEDOM

Informally, a data race occurs when two threads concurrently access the same memory location and at least one of the accesses is a write. A program is DRF if it never generates data races in all possible executions. Below we first define the conflict of footprints.

$$\begin{array}{rcl} \delta_1 \frown \delta_2 & \text{iff} & (\delta_1.ws \cap \delta_2 \neq \emptyset) \ \vee \ (\delta_2.ws \cap \delta \neq \emptyset) \\ (\delta_1, d_1) \frown (\delta_2, d_2) & \text{iff} & (\delta_1 \frown \delta_2) \wedge (d_1 = 0 \vee d_2 = 0) \end{array}$$

Two footprints δ_1 and δ_2 are conflicting, denoted as $\delta_1 \frown \delta_2$, if the write set ws of one of them overlaps with the read set or the write set of the other. Recall that, when used as a set, δ represents $\delta.rs \cup \delta.ws$. Since we do *not* treat accesses of the same memory location inside atomic blocks as a race, we instrument a footprint δ with the atomic bit d to record whether the footprint is generated inside an atomic block ($d = 1$) or not ($d = 0$). Two instrumented footprints (δ_1, d_1) and (δ_2, d_2) are conflicting if δ_1 and δ_2 are conflicting and at least one of d_1 and d_2 is 0.

We formulate data races in Fig. 9(a) for preemptive semantics. A program P with initial memory state σ is racy ($(P, \sigma) \Longmapsto \mathsf{Race}$) if its execution reaches a program configuration W' that steps to

$$\frac{(P,\sigma) \xrightarrow{load} W \qquad W \Rightarrow^* W' \qquad W' \Longmapsto \mathsf{Race}}{(P,\sigma) \Longmapsto \mathsf{Race}}$$

$$\frac{\mathtt{t}_1 \neq \mathtt{t}_2 \qquad (\delta_1, d_1) \frown (\delta_2, d_2) \qquad \mathtt{predict}(W, \mathtt{t}_1, (\delta_1, d_1)) \qquad \mathtt{predict}(W, \mathtt{t}_2, (\delta_2, d_2))}{W \Longmapsto \mathsf{Race}}\ \mathsf{Race}$$

$$\frac{W = (T, \_, 0, \sigma) \qquad T(\mathtt{t}) = (F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}} (\kappa', \sigma')}{\mathtt{predict}(W, \mathtt{t}, (\delta, 0))}\ \text{Predict-0}$$

$$\frac{W = (T, \_, 0, \sigma) \qquad T(\mathtt{t}) = (F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \qquad F \vdash (\kappa', \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^* (\kappa'', \sigma'')}{\mathtt{predict}(W, \mathtt{t}, (\delta, 1))}\ \text{Predict-1}$$

(a) Data Race in Preemptive Semantics

$$\frac{(\hat{P},\sigma) \overset{load}{:\Longrightarrow} \widehat{W} \qquad \widehat{W} :\Rightarrow^* \widehat{W}' \qquad \widehat{W}' :\Longmapsto \mathsf{Race}}{(\hat{P},\sigma) :\Longmapsto \mathsf{Race}}$$

$$\frac{(\hat{P},\sigma) \overset{load}{:\Longrightarrow} \widehat{W} \quad \mathtt{t}_1 \neq \mathtt{t}_2 \quad (\delta_1, d_1) \frown (\delta_2, d_2) \quad \mathtt{NPpredict}(\widehat{W}, \mathtt{t}_1, (\delta_1, d_1)) \quad \mathtt{NPpredict}(\widehat{W}, \mathtt{t}_2, (\delta_2, d_2))}{(\hat{P},\sigma) :\Longmapsto \mathsf{Race}}$$

$$\frac{\widehat{W} \overset{o}{\underset{\mathsf{emp}}{:\Longrightarrow}} \widehat{W}' \qquad o \neq \tau \qquad \mathtt{t}_1 \neq \mathtt{t}_2 \qquad (\delta_1, d_1) \frown (\delta_2, d_2) \qquad \mathtt{NPpredict}(\widehat{W}', \mathtt{t}_1, (\delta_1, d_1)) \qquad \mathtt{NPpredict}(\widehat{W}', \mathtt{t}_2, (\delta_2, d_2))}{\widehat{W} :\Longmapsto \mathsf{Race}}\ \mathsf{Race}_{\mathsf{np}}$$

$$\frac{\widehat{W} = (T, \_, \mathbb{d}, \sigma) \qquad T(\mathtt{t}) = (F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^* (\kappa', \sigma') \qquad \mathbb{d}(\mathtt{t}) = d}{\mathtt{NPpredict}(\widehat{W}, \mathtt{t}, (\delta, d))}\ \mathsf{Predict}_{\mathsf{np}}$$

(b) Data Race in Non-Preemptive Semantics

Fig. 9. Predictive Semantics for Defining Race

Race ($W' \Longmapsto \mathsf{Race}$). In the Race rule, W steps to Race if there are conflicting footprints of two threads predicted from the current configuration ($\mathtt{predict}(W, \mathtt{t}, (\delta, d))$). The predictions are only performed at the switch points, i.e., the program points outside atomic blocks where $d = 0$, since the intermediate states inside atomic blocks are not visible to other threads. Footprints of steps before reaching the next switch point are able to be predicted using the rules Predict-0 or Predict-1. Predict-0 is used for predicting footprint of the next τ-step of thread $\mathtt{t}$, where there is a switch point (in preemptive semantics) right after the step. Predict-1 applies if thread $\mathtt{t}$ enters an atomic block, the footprint generated by any number of steps inside the atomic block can be predicted. Note that we do not insist on predicting the footprint generated by the execution of the *whole* atomic block because the code inside the atomic block may be nonterminating.

A program P with an initial state σ is DRF if it never steps to Race. A program P is DRF if for any proper initial state σ, $\mathsf{DRF}(P, \sigma)$:

$$\mathsf{DRF}(P,\sigma) \ \text{ iff } \ \neg((P,\sigma) \Longmapsto \mathsf{Race})$$
$$\mathsf{DRF}(P) \ \text{ iff } \ \forall\sigma.\ (\mathsf{GE}(P.\Pi) \subseteq \mathsf{locs}(\sigma)) \wedge (\mathsf{cl}(\mathsf{dom}(\sigma), \sigma) = \mathsf{dom}(\sigma)) \implies \mathsf{DRF}(P, \sigma)$$

Data races in the non-preemptive semantics ($(\hat{P}, \sigma) :\Longmapsto \mathsf{Race}$) are defined in Fig. 9(b). Similar to the definition for the preemptive semantics, a non-preemptive program configuration $\widehat{W}$ steps to

Race ($\widehat{W} :\Longrightarrow \mathsf{Race}$) if two threads are predicted having conflicting footprints, as shown in rule $\mathsf{Race}_{\mathsf{np}}$. Predictions are made only at the switch points of non-preemptive semantics ($\widehat{W} :\xRightarrow[\mathsf{emp}]{o} \widehat{W}'$ where $o \neq \tau$), i.e., program points at atomic block boundaries, after an observable event, or after thread termination. The prediction rule $\mathsf{Predict}_{\mathsf{np}}$ is a unified version of its counterpart Predict-0 and Predict-1, where $\mathbb{d}$ indicates whether the predicted steps are inside an atomic block. Similar to the Predict-1 rule, we do *not* insist on predicting the footprint generated by the execution reaching the next switch point, because the code segments between switch points could be nonterminating. In addition to at the switch points, we also need to be able to perform a prediction at the initial state as well (the second rule in Fig. 9), because the first executing thread is non-deterministically picked at the beginning, which has similar effect as thread switching.

A program $\hat{P}$ with the initial state σ is NPDRF if it never steps to Race, and a program $\hat{P}$ is NPDRF if, for any proper initial state σ, $\mathsf{NPDRF}(\hat{P}, \sigma)$:

$$\mathsf{NPDRF}(\hat{P}, \sigma) \;\text{ iff }\; \neg((\hat{P}, \sigma) :\Longrightarrow \mathsf{Race})$$
$$\mathsf{NPDRF}(P) \;\text{ iff }\; \forall\sigma.\, (\mathsf{GE}(P.\Pi) \subseteq \mathsf{locs}(\sigma)) \wedge (\mathsf{cl}(\mathsf{dom}(\sigma), \sigma) = \mathsf{dom}(\sigma)) \implies \mathsf{NPDRF}(P, \sigma)$$

Note that defining NPDRF is not for studying the absence of data races in the non-preemptive semantics (which is probably not very interesting since the execution is non-preemptive anyway). Rather, it is intended to serve as an equivalent notion of DRF but formulated in the non-preemptive semantics. The following lemma shows the equivalence.

Lemma 12 (Equivalence between DRF and NPDRF, ⑥ and ⑧ in Fig. 2)**.**
For any $\mathsf{f}_1, \ldots, \mathsf{f}_m, \sigma, \Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$ such that $\forall i.\ \mathsf{wd}(tl_i)$,

$$\mathsf{DRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_m, \sigma) \iff \mathsf{NPDRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_m, \sigma)\ .$$

As mentioned in Sec. 2, we need the compilation to preserve DRF of the source, which should be ensured by our upward whole program simulation $\hat{P} \leqslant_{\varphi} \hat{\mathbb{P}}$. The following Lemma 13 shows the simulation preserves NPDRF of the source. Together with Lemma 12, we know it preserves DRF.

Lemma 13 (NPDRF Preservation, ⑦ in Fig. 2)**.**
For any $\hat{\mathbb{P}}$, $\hat{P}$, φ, Σ and σ, if $\hat{P} \leqslant_{\varphi} \hat{\mathbb{P}}$, $\mathsf{initM}(\varphi, \mathsf{GE}(\hat{\mathbb{P}}.\Gamma), \Sigma, \sigma)$, and $\mathsf{NPDRF}(\hat{\mathbb{P}}, \Sigma)$, then $\mathsf{NPDRF}(\hat{P}, \sigma)$.

The lemma below shows the semantics equivalence for DRF programs.

Lemma 14 (Equivalence between Preemptive and Non-Preemptive semantics, ① and ② in Fig. 2)**.**
For any $\Pi, \mathsf{f}_1, \ldots, \mathsf{f}_m, \sigma$, if $\mathsf{DRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_m, \sigma)$, then

$$(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_m, \sigma) \approx (\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_m, \sigma)\ .$$

7 THE FINAL THEOREM

Putting all the previous results together, we are able to prove our final theorem, i.e., certified sequential compositional compilers could correctly compile data-race-free concurrent programs by compiling each module separately.

Before presenting the theorem, we first model a sequential compiler SeqComp as a code transformation function CodeT with a data transformation function φ. Here φ maps the addresses in the global environments *ge*. It may not be an identity function when the compiler performs optimizations on global environments, such as eliminating unused global variables.

$$\mathsf{SeqComp} ::= (\mathsf{CodeT}, \varphi) \qquad \mathsf{CodeT} \in \mathit{Module} \rightharpoonup \mathit{Module} \qquad \varphi \in \mathit{Addr} \rightharpoonup \mathit{Addr}$$

As the key proof obligation, we need to verify that each SeqComp satisfies Correct. We define Correct as follows using our footprint-preserving module-local simulation.

Definition 15 (Sequential Compiler Correctness). Correct(SeqComp, sl, tl) iff

$$\forall \gamma, \pi, ge, ge'.\ \mathsf{SeqComp.CodeT}(\gamma) = \pi \wedge \mathsf{SeqComp}.\varphi\{ge\} = ge' \implies (sl, ge, \gamma) \preccurlyeq_{\mathsf{SeqComp}.\varphi} (tl, ge', \pi)\ .$$

The desired correctness property GCorrect of concurrent program compilation is the semantics preservation of whole programs, i.e., every target concurrent program is an event-trace refinement of the source. We formulate GCorrect in Def. 16. Here we require all the sequential compilers to agree on the transformation φ of global environments (see Def. 16(1)).

Definition 16 (Concurrent Compiler Correctness).
GCorrect($(\mathsf{SeqComp}_1, sl_1, tl_1), \ldots, (\mathsf{SeqComp}_m, sl_m, tl_m)$) iff
for any $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, $\Pi = \{(tl_1, ge'_1\pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, φ, if

(1) $\forall i \in \{1, \ldots, m\}.\ (\mathsf{SeqComp}_i.\mathsf{CodeT}(\gamma_i) = \pi_i) \wedge (\mathsf{SeqComp}_i.\varphi = \varphi) \wedge \mathsf{injective}(\varphi)$,
(2) DRF(**let** Γ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$), and Safe(**let** Γ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$),
(3) $\forall i \in \{1, \ldots, m\}.\ \mathsf{ReachClose}(sl_i, ge_i, \gamma_i)$,

then **let** Γ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_\varphi$ **let** Π **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

Our final theorem is then formulated as Thm. 17. It says if a set of sequential compilers are certified to satisfy our correctness obligation Correct, the source and target languages sl_i and tl_i are well-defined, and the target languages are deterministic, then the sequential compilers as a whole is GCorrect for compiling concurrent programs. The proof simply applies the lemmas that correspond to ①-⑧ in Fig. 2.

Theorem 17 (Final Theorem).
For any $\mathsf{SeqComp}_1, \ldots, \mathsf{SeqComp}_m$, $sl_1, \ldots, sl_m$, $tl_1, \ldots, tl_m$ such that for any $i \in \{1, \ldots, m\}$ we have wd(sl_i), wd(tl_i), det(tl_i), and Correct($\mathsf{SeqComp}_i$, sl_i, tl_i), then

$$\mathsf{GCorrect}((\mathsf{SeqComp}_1, sl_1, tl_1), \ldots, (\mathsf{SeqComp}_m, sl_m, tl_m)).$$

8 FRAMEWORK INSTANTIATION AND COMPCERT BACKEND VERIFICATION

We apply our compiler verification framework to prove the correctness of CompCert-3.0.1 x86 backend [Leroy 2009a; CompCert Developers 2017] for compositional compilation of DRF programs. To demonstrate the support of cross-language inter-module interaction, we provide synchronization modules implemented in x86 assembly and link them with the modules compiled by CompCert.

8.1 Language Instantiations

Below we instantiate our abstract languages as Cminor and x86 assembly. Cminor is a low-level imperative language, and serves as the source languge of several compiler backend of the CompCert compiler. Although it is a sequential language, we can write concurrent Cminor programs by parallel compositions of sequential Cminor threads. Inter-thread synchronization can be achieved through the external call mechanism to call functions outside Cminor modules.

We give a tiny example in Fig. 10, where the synchronization functions `sc_cas` and `sc_store` are implemented in a separate x86 module $\pi_{\mathtt{x86}}$ (see Fig. 10(b)). Their signatures have been exported to Cminor in Fig. 10(a). These functions are similar to the C-11 SC-atomic primitives [Batty et al. 2011]. The function `sc_cas` tests if the value in the destination memory location (1st argument) is the same with the expected value (2nd argument). If true it stores the new value (3rd argument) into the destination address. The `sc_store` function simply stores the second argument to the destination memory location (1st argument). Their implementation in Fig. 10(b) utilizes the `lock`-prefixed instructions. With these external x86 functions, the Cminor program implements a simple test-and-set lock and a counter `inc` that increments the shared variable `x` in the lock-protected

```
void    sc_store (int32_t *, int32_t)
int     sc_cas   (int32_t *, int32_t, int32_t)
```

(a) sc_atomic signatures

```
sc_store:
        movl   4(%esp),    %eax
        movl   8(%esp),    %ecx
   lock xchgl  %ecx,       (%eax)
        retl
sc_cas:
        movl   4(%esp),    %edx
        movl   8(%esp),    %eax
        movl   12(%esp),   %ecx
lock cmpxchgl  %ecx,       (%edx)
        sete   %al
        retl
```

(b) sc_atomic x86 implementation, module π_{x86}

```
int32_t x = 0, l = 0;

int lock(){  return sc_cas(&l, 0, 1); }

void unlock(){  sc_store(&l, 0); }

void inc(){
  int32_t tmp;
  while(!lock());
  tmp = x;  x ++;
  unlock();
  print(tmp);
}
```

(c) example Cminor module γ_{C}

Fig. 10. An example program with implementation of C11-like atomics

critical region and prints the old value of x. Here we present the Cminor program in C syntax. An example whole program $\mathbb{P}$ is **let** $\{\pi_{\text{x86}}, \gamma_{\text{C}}\}$ **in** `inc()` $\|$ `inc()`.

The Cminor Language. The selected parts of our Cminor instantiation are represented in Fig. 11(a). To instantiate our abstract language of Fig. 3, we instantiate the *Module* with the same syntax as the CompCert Cminor languge. A core state κ is a pair of a local state c and an index N of type $\mathbb{N}$ indicating the position of the next block in the freelist F to be allocated. Here F is defined as a sequence of block numbers. The local state c is instantiated the same way as the Cminor interaction semantics in Compositional CompCert [Stewart et al. 2015]. The InitCore function is also adapted from Compositional CompCert, with the index N initialized to 0. We omit their definitions here.

The instantiation of the local transition semantics instruments Compositional CompCert's interaction semantics with footprints, which is determined by the memory locations (base-offset pairs, following the CompCert block-based memory model) accessed in each step. For instance, the allocation (e.g., for allocating stack frames) in our semantics takes the next available block number ($b = F(N)$) as the base address and then increments N so that it points to the next available block number in F. The memory σ is extended with the newly allocated memory cells, resulting in $\sigma\{(b, 0) \rightsquigarrow \mathsf{undef}, \ldots, (b, n-1) \rightsquigarrow \mathsf{undef}\}$, where n is the size of allocated block. The write set of the footprint includes all the addresses with base address b, and the read set is empty.

As we mentioned at the beginning of this subsection, Cminor does not support synchronization operations within the language, therefore the operational semantics of Cminor does not generate steps with labels EntAtom or ExtAtom.

x86 Assembly with `lock`*-prefixed Instructions.* Figure 11(b) presents the selected syntax and key semantics components of x86 assembly with `lock`-prefixed instructions. In addition to the instructions that are already implemented in CompCert, we introduce the `lock`-prefixed instructions (`lockxchg`, `lockxadd`, `lockcmpxchg`) for synchronization.

The `lock` prefix asserts a `lock#` signal (or something alike in later IA-32 implementations), which guarantees exclusive memory access while executing the accompanying instruction. The behavior of `lock`-prefixed instructions could be modeled by our abstract language with EntAtom and ExtAtom events. To implement the semantics of `lock`-prefixed instructions, we introduce a

$$(FList)\quad F ::= b_1 :: b_2 :: \ldots \qquad (Mem)\quad \sigma \in Block \rightharpoonup_{\text{fin}} (\mathbb{N} \rightharpoonup \texttt{val})$$
$$(Block)\quad b \in \mathbb{N}^+ \qquad (Core)\quad \kappa ::= (c, N) \qquad (BIndex)\ N \in \mathbb{N}$$

(a) Cminor instantiation

$$\begin{array}{rrll} (Instr) & c & ::= & \texttt{mov}\ \texttt{r}_d, \texttt{r}_s \mid \texttt{Iadd}\ \texttt{r}_d, \texttt{r}_s \mid \texttt{call f} \mid \texttt{ret} \mid \ldots \\ & & \mid & \texttt{lock xchg}\ l\ \texttt{r}_s \mid \texttt{lock xadd}\ l\ \texttt{r}_s \mid \texttt{lock cmpxchg}\ l\ \texttt{r}_s \\ (Core) & \kappa & ::= & (s, R) \qquad (Signal)\ s ::= \texttt{none} \mid \texttt{locked} \mid \texttt{done} \\ (Register) & \texttt{r} & ::= & \texttt{PC} \mid \texttt{SP} \mid \ldots \qquad (RegFile)\ R \in Register \to Val \end{array}$$

$$\frac{\texttt{find\_instr}(ge, R(\texttt{PC})) = \texttt{lock xchg}\ l\ \texttt{r}_s}{(ge, F) \vdash < (\texttt{none}, R), \sigma > \overset{\text{EntAtom}}{\underset{\text{emp}}{\longmapsto}} < (\texttt{locked}, R), \sigma >} \qquad \frac{}{(ge, F) \vdash < (\texttt{done}, R), \sigma > \overset{\text{ExtAtom}}{\underset{\text{emp}}{\longmapsto}} < (\texttt{none}, R), \sigma >}$$

$$\frac{\texttt{find\_instr}(ge, R(\texttt{PC})) = \texttt{lock xchg}\ l\ \texttt{r}_s \quad \delta = (\{l\}, \{l\}) \quad R' = R\{\texttt{PC} \rightsquigarrow (\texttt{PC} + 1), \texttt{r}_s \rightsquigarrow \sigma(l)\}}{(ge, F) \vdash < (\texttt{locked}, R), \sigma > \overset{\tau}{\underset{\delta}{\longmapsto}} < (\texttt{done}, R'), \sigma\{l \rightsquigarrow R(\texttt{r}_s)\} >}$$

(b) Pseudo-x86 Assembly

Fig. 11. Language instantiations

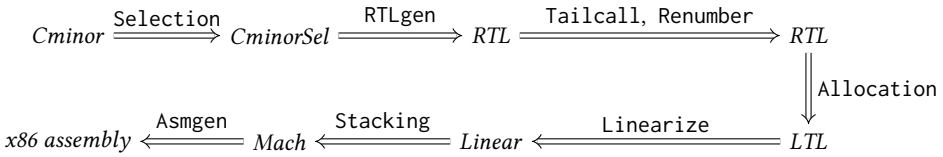


Fig. 12. Proved CompCert Compilation Passes

Signal flag in the core state κ. The flag has three possible values: none means a normal state with no lock prefix asserted; locked means a lock# signal is asserted; and done indicates that we have finished executing the accompanying instruction and is about to unset the lock# signal. We give the operational semantics rules for lockxchg as an example at the bottom of Fig 11(b).

8.2 Adapting CompCert Compiler

As mentioned at the beginning of this section, we adapt the original CompCert compiler passes for compiling Cminor modules to x86 assembly. The compilation passes (shown in Fig. 12) include all translation passes and two optimization passes (Tailcall and Renumber), which are proved to be correct with respect to our correctness judgement. Other optimization passes have not been proved yet and are left as future work.

Given the program **let** $\{\gamma_1, \ldots, \gamma_l, \pi_{l+1}, \ldots, \pi_m\}$ **in** $\texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n$ consisting of Cminor modules γ_i and x86 modules π_j (we omit the corresponding language definitions *sl* and *ge* in the modules to simplify the presentation), the compilation Comp is formulated as

$$\texttt{Comp}(\textbf{let}\ \{\gamma_1, \ldots, \gamma_l, \pi_{l+1}, \ldots, \pi_m\}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n) \overset{\text{def}}{=}$$
$$\textbf{let}\ \{\texttt{CminorTrans}(\gamma_1), \ldots, \texttt{CminorTrans}(\gamma_l), \texttt{IdTrans}(\pi_{l+1}), \ldots, \texttt{IdTrans}(\pi_m)\}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n$$

where CminorTrans is the adapted compiler consisting of the original CompCert backend passes, and IdTrans is the identity translation which returns the x86 assembly module unchanged. The state transition function φ of both compilers are instantiated as identity function.

We have proved that CminorTrans satisfies our Correct condition:

Lemma 18 (`CminorTrans` Correctness)**.** Correct(`CminorTrans`, `Cminor`, `x86asm`).

We also prove the correctness of `IdTrans`, the well-definedness of Cminor and the x86 assembly language, and the determinism of x86 assembly. Together with our framework's final theorem (Theorem 17), we proved the following result:

Theorem 19 (Compilation Correctness)**.**
GCorrect((`CminorTrans`, `Cminor`, `x86asm`), (`IdTrans`, `x86asm`, `x86asm`)) .

To prove Lemma 18, we try to reuse as much the original CompCert correctness proofs as possible, but have encountered two major challenges: (1) Many CompCert lemmas rely on the specific definition of the CompCert memory model, which is different from ours; and (2) Footprints are new in our setting and it seems we need to re-prove most existing lemmas to support footprints. Below we explain these challenges and show our efforts to reuse CompCert proofs.

8.2.1 Reusing CompCert Proofs by Converting Memory Layout. The memory layout in our semantics is different from the CompCert memory model. CompCert memory maintains a `nextblock` field indicating the next block to be allocated. Starting from 0, `nextblock` is incremented after each allocation. Therefore *the sequence of memory allocations in an execution get consecutive natural numbers as block numbers*. As a consequence, b is a valid block number if $b <$ `nextblock`. Also a block with a smaller block number must be allocated earlier than those with bigger block numbers.

But these assumptions do not hold in our model. As we explain in Sec. 8.1, each allocation takes the block number $F(N)$ and then increments the index N, but the block numbers on F are *not* consecutive. Actually we do not even assume an increasing order of the block numbers. Recall that in our model each thread has its own freelist F, which can be an infinite sequence of block numbers. Also the freelists of different threads must be disjoint. This means we *cannot* make a freelist F a infinite sequence of consecutive natural numbers to directly simulate CompCert.

Unfortunately, CompCert correctness proofs heavily rely on its allocation strategy, making it difficult for us to reuse the proofs. For instance, the check of block validity is used extensively in CompCert's fundamental libraries (e.g., `Memory.v` for memroy operations and their properties, and `Separation.v` for separation logic style predicates) and the compilation correctness proofs. Modifying the validity check would affect most of these proofs.

Lifting simulations to CompCert memory model. Although our memory model is different from that of CompCert, we can define a bijection between memories under the two models. As a result, the behaviors of a thread under our model are equivalent to its behaviors under CompCert model, and our module-local simulation can be derived from a simulation based on the CompCert model.

In detail, we first define auxiliary semantics for the source and target languages based on CompCert memory model. We denote the CompCert memory by the hat-notation $\hat{\Sigma}$ and $\hat{\sigma}$, and the corresponding core states under CompCert memory model by $\hat{\mathbb{k}}$ and $\hat{\kappa}$. The lemmas below show the correspondence between our original semantics and the auxiliary semantics.

Lemma 20 (Lifting `Cminor`)**.** For any `Cminor` module (ge, γ) and function entry `f`, for any $\mathbb{k}, \Sigma, \hat{\Sigma}$, if $\mathsf{InitCore}(\gamma, \mathtt{f}) = \mathbb{k}$ and $ge \subseteq \mathsf{dom}(\Sigma)$, then there exist $\hat{\mathbb{k}}, \hat{\Sigma}, \mu_s$ and f such that $\mu_s = (\mathsf{dom}(\Sigma), \mathsf{dom}(\hat{\Sigma}), f)$ and $(\mathbb{F}, (\mathbb{k}, \Sigma)) \preccurlyeq_{\mu_s} (\hat{\mathbb{k}}, \hat{\Sigma})$. Here f is a bijection from $\mathsf{dom}(\Sigma)$ to $\mathsf{dom}(\hat{\Sigma})$.

It says that, starting from some well-formed initial state, the execution in our semantics is simulated by the execution in the auxiliary semantics. Here $(\mathbb{F}, (\mathbb{k}, \Sigma)) \preccurlyeq_{\mu_s} (\hat{\mathbb{k}}, \hat{\Sigma})$ is defined as a simple lock-step simulation relation, where the corresponding steps in the two semantics generate footprints Δ and $\hat{\Delta}$ such that $\mathsf{FPmatch}(\mu_s, \Delta, \hat{\Delta})$ holds.

The next lemma shows simulation in the reverse direction in the target language (x86 assembly).

```
Lemma sel_expr_correct:
  forall sp e m a v fp,
  Cminor.eval_expr sge sp e m a v ->
  Cminor.eval_expr_fp sge sp e m a fp ->
  forall e' le m', env_lessdef e e' -> Mem.extends m m' ->
  exists v', exists fp',
  eval_expr tge sp e' m' le (sel_expr a) v' /\ Val.lessdef v v'
  /\ eval_expr_fp tge sp e' m' le (sel_expr a) fp' /\ FP.subset fp' fp.
```

Fig. 13. Coq code example

Library Code	Spec		Proof	
	CompCert	Ours	CompCert	Ours
Selectionproof.v	336	500	647	780
RTLgenproof.v	428	543	821	857
Tailcallproof.v	173	328	275	403
Renumberproof.v	86	245	117	356
Allocproof.v	704	785	1410	1696
Linearizeproof.v	236	371	349	732
Stackingproof.v	730	1154	1108	2015
Asmgenproof.v	208	881	571	583
Compositionality (Lemma 8)		580		2249
DRF preservation (Lemma 13)		358		1142
Semantics equivalence (Lemma 14)		1529		4742
Lifting (Theorem 22)		828		1795

Fig. 14. Lines of code (using coqwc) for selected parts of the Coq implementation

Lemma 21 (Lifting `x86asm`)**.** For any `x86asm` module (ge, π) and function entry `f`, for any $\kappa, \sigma, \hat{\sigma}$, if $\mathsf{InitCore}(\pi, \mathtt{f}) = \kappa$ and $ge \subseteq \mathsf{dom}(\sigma)$, then there exist $\hat{\kappa}, \hat{\sigma}, \mu_t$ and f such that $\mu_t = (\mathsf{dom}(\hat{\sigma}), \mathsf{dom}(\sigma), f)$ and $(\hat{\kappa}, \hat{\sigma}) \preccurlyeq_{\mu_t} (F, (\kappa, \sigma))$. Here f is a bijection from $\mathsf{dom}(\hat{\sigma})$ to $\mathsf{dom}(\sigma)$.

Theorem 22 below says we can derive the local simulation $(\mathbb{F}, (\Bbbk, \Sigma), \Delta) \preccurlyeq_{\mu} (F, (\kappa, \sigma), \delta)$ by proving the simulation $(\hat{\Bbbk}, \hat{\Sigma}, \hat{\Delta}) \preccurlyeq_{\hat{\mu}} (\hat{\kappa}, \hat{\sigma}, \hat{\delta})$ instead. Here $(\hat{\Bbbk}, \hat{\Sigma}, \hat{\Delta}) \preccurlyeq_{\hat{\mu}} (\hat{\kappa}, \hat{\sigma}, \hat{\delta})$ is the same as our local simulation, except using CompCert memory and the corresponding core states instead of our memory with freelists. As a result, most CompCert libraries and compilation proofs could be reused without modification.

Theorem 22 (Lifting)**.** If $(\mathbb{F}, (\Bbbk, \Sigma)) \preccurlyeq_{\mu_s} (\hat{\Bbbk}, \hat{\Sigma})$, $\mathsf{FPmatch}(\mu_s, \Delta, \hat{\Delta})$, $(\hat{\kappa}, \hat{\sigma}) \preccurlyeq_{\mu_t} (F, (\kappa, \sigma))$, $\mathsf{FPmatch}(\mu_t, \hat{\delta}, \delta)$, and $\mu = \mu_t \circ \hat{\mu} \circ \mu_s$, then $(\hat{\Bbbk}, \hat{\Sigma}, \hat{\Delta}) \preccurlyeq_{\hat{\mu}} (\hat{\kappa}, \hat{\sigma}, \hat{\delta}) \implies (\mathbb{F}, (\Bbbk, \Sigma), \Delta) \preccurlyeq_{\mu} (F, (\kappa, \sigma), \delta)$. Here $\mu_2 \circ \mu_1 = (S_1, S_2, f_2 \circ f_1)$, if $\mu_1 = (S_1, S, f_1)$ and $\mu_2 = (S, S_2, f_2)$.

8.2.2 Footprint Preservation. Although CompCert does not model footprints, many of the definitions and lemmas can be slightly modified to support footprint preservation. For instance, the `Selection` phase selects appropriate machine operations for operations in `Cminor`, and generates `CminorSel` code. One of the key lemmas, `sel_expr_correct`, is shown in Fig. 13, with our newly-added code highlighted as blue texts. It says the selected expression would evaluate to a value refined by the `Cminor` expression. We simply extends the lemma by requiring *the selected expression has smaller footprint while evaluating on related memory.*

Adapting the simulation invariants (the `match_state` relations) in CompCert proofs is also straightforward. We just need to instrument `match_state` with footprint relations.

8.3 Proof Efforts in Coq

Statistics of our Coq implementation and proofs are depicted in Fig 14. We can see that adapting compilation correctness proofs from CompCert is relatively lightweight. For most phases our proofs

are within 300 lines of code more than the original CompCert proofs. The `Stacking` phase introduces more additional proofs, mostly caused by arguments marshalling for supporting cross-language linking, following Compositional CompCert. In our experience, adapting CompCert's original compilation proofs to our settings takes less than one person week per translation phase (except for `Stacking`). For simpler phases such as `Tailcallproof.v`, `Linearizeproof.v`, `Allocproof.v`, and `RTLgenproof.v`, it takes less than one person day per phase.

In contrast, implementing the framework and proving its correctness are much more challenging, which took us about 1 person year. In particular, proving the equivalence between the non-preemptive and the preemptive semantics under the DRF assumption took us much more time than expected, although it seems to be a well-known folklore theorem. The co-inductive proofs there involve a large number of non-trivial cases of reordering threads' execution.

9 RELATED WORK

Compiler verification. There has been various work extending CompCert [Leroy 2009a] to support separate compilation or concurrency. SepCompCert [Kang et al. 2016] extends CompCert with the support of syntactical linking. Their approach requires all the compilation units be compiled by CompCert. They do not support cross-language linking or concurrency as we do.

CompCertX [Gu et al. 2015] extends CompCert semantics with abstract layers for verified separate compilation. It does not support general DRF concurrent program compilation as we do.

Compositional CompCert [Beringer et al. 2014; Stewart et al. 2015] introduces interaction semantics to support cross-language interactions. Its module-linking theorem allows escape of stack pointers, which we do not support. They also conjecture that their approach can be extended to verify compositional compilation of well-synchronized concurrent programs. This work addresses the key challenges to do so and proposes a verification framework to achieve this goal.

CompCertTSO [Ševčík et al. 2013] extends CompCert to compile concurrent C-like programs in a relaxed memory model. It focuses on the correctness of a particular compiler and does not support cross-language linking. Also there are two compilation phases whose proofs are not compositional.

Vellvm [Zhao et al. 2012, 2013] proves correctness of several optimization passes for *sequential* LLVM programs. Perconti and Ahmed [2014] verify separate compilation by embedding languages in a combined language. They do not support concurrency either. Ševčík [2011] studies safety of a class of optimizations in concurrent settings using an abstract trace semantics. It is unclear if his approach can be applied to verify general compilation. Lochbihler [2010] verifies a compiler for concurrent Java programs. He requires the target and the source always be lock-step on heap updates, which makes his simulation compositional but restricted. It is unclear how to apply his approach to verify more complex compilers like CompCert.

Non-preemptive semantics and data-race-freedom. Non-preemptive (or cooperative) semantics has been developed in various settings for various purposes (e.g., [Abadi and Plotkin 2009; Boudol 2007; Li and Zdancewic 2007; Vouillon 2008; Yi et al. 2011]). Both Ferreira et al. [2010] and Xiao et al.[2018] study the relationships between non-preemptive semantics and DRF, but they do not give any mechanized proofs of termination-preserving semantics equivalence as in our work. DRFx [Marino et al. 2010] proposes a concept called Region-Conflict-Freedom, which looks similar to our NPDRF, but there is no formal operational formulation as we do.

Compilation validation. Validation is another technique to ensure correctness of compilation. CompCert employs a verified validator [Rideau and Leroy 2010] for register allcation. More recently Crellvm [Kang et al. 2018] is developed for verified credible compilation of LLVM IR. The work is based on the Vellvm semantics with no support for concurrency. It is not clear how to apply the validation techniques to concurrent settings.

REFERENCES

Martin Abadi and Gordon Plotkin. 2009. A Model of Cooperative Threads. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 29–40.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. 55–66.

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP*. 107–127.

Gérard Boudol. 2007. Fair Cooperative Multithreading. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–286.

Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized Memory Models and Concurrent Separation Logic. In *ESOP*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–286.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *POPL*. 595–608.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *POPL*. 178–190.

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *PLDI 2018*. 631–645.

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43 (December 2009), 363–446. Issue 4.

Peng Li and Steve Zdancewic. 2007. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-level Concurrency Primitives. In *PLDI '07*. 189–199.

Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-guarantee-based Simulation for Verifying Concurrent Program Transformations. In *POPL*. 455–468.

Andreas Lochbihler. 2010. Verifying a Compiler for Java Threads. In *ESOP*. 427–447.

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI '10*. 351–362.

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems*, Zhong Shao (Ed.). 128–148.

Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling. In *Compiler Construction*, Rajiv Gupta (Ed.). 224–243.

Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI*. 306–316.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. 275–287.

CompCert Developers. 2017. CompCert-3.0.1. http://compcert.inria.fr/release/compcert-3.0.1.tgz

Jérôme Vouillon. 2008. Lwt: A Cooperative Thread Library. In *ML*. 3–12.

Siyang Xiao, Hanru Jiang, Hongjin Liang, and Xinyu Feng. 2018. Non-Preemptive Semantics for Data-Race-Free Programs. In *ICTAC*. to appear.

Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. 2011. Cooperative Reasoning for Preemptive Execution. In *PPoPP*. 147–156.

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*. 427–440.

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *PLDI*. 175–186.