# Towards Certified Separate Compilation for Concurrent Programs

Hanru Jiang*
University of Science and Technology of China, China
hanru219@mail.ustc.edu.cn

Hongjin Liang*‡
Nanjing University, China
hongjin@nju.edu.cn

Siyang Xiao Junpeng Zha
University of Science and Technology of China, China
{yutio888,jpzha}@mail.ustc.edu.cn

Xinyu Feng†‡
Nanjing University, China
xyfeng@nju.edu.cn

## Abstract

Certified separate compilation is important for establishing end-to-end guarantees for certified systems consisting of multiple program modules. There has been much work building certified compilers for sequential programs. In this paper, we propose a language-independent framework consisting of the key semantics components and lemmas that bridge the verification gap between the compilers for sequential programs and those for (race-free) concurrent programs, so that the existing verification work for the former can be reused. One of the key contributions of the framework is a novel footprint-preserving compositional simulation as the compilation correctness criterion. The framework also provides a new mechanism to support confined benign races which are usually found in efficient implementations of synchronization primitives.

With our framework, we develop CASCompCert, which extends CompCert for certified separate compilation of race-free concurrent Clight programs. It also allows linking of concurrent Clight modules with x86-TSO implementations of synchronization primitives containing benign races. All our work has been implemented in the Coq proof assistant.

***CCS Concepts*** • **Theory of computation → Program verification**; **Abstraction**; • **Software and its engineering → Correctness**; **Semantics**; *Concurrent programming languages*; *Compilers*.

***Keywords*** Concurrency, Certified Compilers, Data-Race-Freedom, Simulations

**ACM Reference Format:**
Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314595

## 1 Introduction

Separate 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 interactions between modules be preserved at the target.

CompCert [16], 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. [29] develop Compositional CompCert, which allows the modules to call each other's external functions. Like CompCert, Compositional CompCert only supports sequential programs too.

Stewart et al. [29] do argue that Compositional CompCert may be applied for data-race-free (DRF) concurrent programs, since they behave the same in the standard interleaving semantics as in certain non-preemptive semantics where switches between threads occur only at certain designated program points. 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 to actually implement it. We need proper formulation of the non-preemptive semantics and the notion of DRF. Then we need to indeed prove that DRF programs have the same behaviors (including termination) in the interleaving semantics as in the non-preemptive semantics,

[*] The first two authors contributed equally and are listed alphabetically.
[†] Corresponding author.
[‡] Also with State Key Laboratory for Novel Software Technology.

*PLDI '19, June 22–26, 2019, Phoenix, AZ, USA*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314595

and verify that the compilation preserves DRF. More importantly, the formulation and the proofs should be done in a compositional and language-independent way, to allow separate compilation where the modules can be implemented in different languages. Reusing the proofs of CompCert is challenging too, as memory allocation for multiple threads may require a different memory model from that of CompCert, and the non-deterministic semantics also makes it difficult to reuse the downward simulation in CompCert.

In addition, the requirement of DRF could be sometimes overly restrictive. Although Boehm [3] points out there are essentially *no* "benign" races in source programs, and languages like C/C++ essentially give no semantics to racy programs [4], it is still meaningful to allow benign races in machine language code for better performance (e.g., the spin locks in Linux). Also machine languages commonly allow relaxed behaviors. Can we support realistic target code with potential benign races in relaxed machine models?

There has been work on verified compilation for concurrent programs [20, 27], but with only limited support of compositionality. We explain the major challenges in detail in Sec. 2, and discuss more related work in Sec. 8.

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 apply our framework to verify the correctness of CompCert for separate compilation of DRF programs to x86 assembly. We also extend the framework to allow *confined benign races* in x86-TSO (confined in that the racy code must execute in a separate region of memory and have race-free abstraction), so that optimized implementations of synchronization primitives like spin-locks in Linux can be supported. 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 (Sec. 4). It 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 the footprints (i.e., the set of memory locations accessed during the execution) of the source and target modules to be related. This way we effectively reduce the proof of the compiler's preservation of DRF, a whole program property, to the proof of *local* footprint preservation.
- We work with an *abstract* programming language (Sec. 3), 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 access 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 Clight and x86 assembly. With the abstract language, we study the equivalence between preemptive and non-preemptive semantics (Sec. 3.3), the equivalence between DRF and NPDRF (the notion of race-freedom defined in the non-preemptive setting, shown in Sec. 5), and the properties of our new simulation. As a result, the lemmas in our proof framework are re-usable when instantiating to real languages.
- We prove a strengthened DRF-guarantee theorem for the x86-TSO model (Sec. 7.3). It allows an x86-TSO program to call a (x86-TSO) module with benign races. If one can replace the racy module with a more abstract version so that the resulting program is DRF in the sequentially consistent (SC) semantics, then the original racy x86-TSO program would behave the same with the more abstract program in the SC semantics. This way we allow the target programs to have confined benign races in the relaxed x86-TSO model. We use this approach to support efficient x86-TSO implementations of locks.
- Putting all these together, our framework (see Fig. 2 and Fig. 3) successfully builds certified separate 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 develop the certified compiler CASCompCert[1] (Sec. 7). We instantiate the source and target languages as Clight and x86 assembly. We also provide an efficient x86-TSO implementation of locks as a synchronization library. CASCompCert reuses the compilation passes of CompCert-3.0.1 [6] (including all the translation passes and four optimization passes). We prove that they satisfy our new compilation correctness criterion, where we reuse a considerable amount of the original CompCert proofs, with minor adjustment for footprint-preservation. The proofs for each pass take less than one person week on average.

Supplementary materials for this paper, including the Coq development and a technical report (TR), are publicly available at https://plax-lab.github.io/publications/ccc/.

## 2 Informal Development

Below we first give an overview of the main ideas in CompCert [16, 17] and Compositional CompCert [29] as starting points for our work. Then we explain the challenges and our ideas in reusing them to build certified separate compilation for concurrent programs.

[1] It is short for an extended **CompCert** with the support of **C**oncurrency, **A**bstraction and **S**eparate compilation.

### 2.1 CompCert

Figure 1(a) shows the key proof structure of CompCert. The compilation *Comp* is correct, if for every source program $S$, the compiled code $C$ preserves the semantics of $S$. That is,

$$\mathrm{Correct}(Comp) \overset{\text{def}}{=} \forall S, C.\ Comp(S) = C \implies S \approx C\ .$$

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 $C$.

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

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

The determinism $\mathrm{Det}(C)$ ensures that $C$ admits only one event trace, so we 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). However, the simulation $\precsim$ relates whole programs only and it does not take into account the interactions with other modules which may update the shared resource. So it is not compositional and does *not* support separate compilation.

### 2.2 Compositional CompCert

Compositional CompCert supports separate compilation by re-defining the simulation relation for modules. Figure 1(b) shows its proof structure. Suppose we make 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);        // Module S2
int f(){                      int g(int *x){
  int a = 0, b = 0;             *x = 3;
  g(&b);                      }
  return a + b; }
```

(2.1)

Suppose the two modules S1 and S2 are independently compiled to the target modules C1 and 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 $G$ [14]. The rely condition $R$ of a module specifies the general callee behaviors happen at the *external function calls* of the current module (shown as the thick arrows in Fig. 1(d)). The guarantee $G$ specifies the possible transitions made by the module itself (the thin arrows). The simulation is compositional as long as the $R$ and $G$ of linked modules are compatible.

$$\begin{tikzcd}
S \approx C \\
S \sqsubseteq C \arrow[u, Rightarrow, "\mathrm{Det}(C)"'] \\
S \precsim C \arrow[u, Rightarrow]
\end{tikzcd}$$

(a) CompCert

$$\begin{tikzcd}
S_1 \circ \ldots \circ S_n \approx C_1 \circ \ldots \circ C_n \\
S_1 \circ \ldots \circ S_n \sqsubseteq C_1 \circ \ldots \circ C_n \arrow[u, Rightarrow, "\mathrm{Det}(C_1 \circ \ldots \circ C_n)"'] \\
S_1 \circ \ldots \circ S_n \precsim C_1 \circ \ldots \circ C_n \arrow[u, Rightarrow] \\
\forall i.\ \mathrm{R}, \mathrm{G} \vdash S_i \precsim' C_i \arrow[u, Rightarrow]
\end{tikzcd}$$

(b) Compositional CompCert

$$\begin{tikzcd}
S \arrow[r] \arrow[d, dash, "\precsim"'] & S' \arrow[d, dash, dashed, "\precsim"] \\
C \arrow[r, dashed, "+"] & C'
\end{tikzcd}$$

(c) $S \precsim C$

$$\begin{tikzcd}
S_1 \arrow[r] \arrow[d, dash, "\precsim'"'] & S_2 \arrow[r, thick] \arrow[d, dash, dashed, "\precsim'"'] \arrow[l, phantom, "G"] & S_3 \arrow[r] \arrow[d, dash, dashed, "\precsim'"'] \arrow[l, phantom, "R"] & S_4 \arrow[d, dash, dashed, "\precsim'"'] \arrow[l, phantom, "G"] \\
C_1 \arrow[r, dashed, "+"] & C_2 \arrow[r, thick] & C_3 \arrow[r, dashed, "+"] & C_4
\end{tikzcd}$$

(d) $R, G \vdash S \precsim' C$

**Figure 1.** Proof structures of certified compilation

Compositional CompCert proves that the CompCert compilation passes satisfy the new 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 example (2.1), the compiler cannot do optimizations based on the (wrong) assumption that b is 0 when f returns.

Since the $R$ steps happen only at the external calls, 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 has been a 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.

### 2.3 Challenges and Our Approaches

Although the idea of applying Compositional CompCert to compile DRF programs is plausible, we have to address several key challenges to really implement it.

***Q: How to give language-independent formulations of DRF and non-preemptive semantics?*** The interaction semantics introduced in Compositional CompCert describes modules' interactions 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?

***A:*** 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 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.

***Q: 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 ask 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.

***A:*** We decide to use a different memory model. 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 (see Sec. 7.2). We address this problem by establishing some semantics equivalence between our memory model and the CompCert memory model (shown in Sec. 7.2).

***Q: How to compositionally prove DRF-preservation?*** The simulation $\precsim'$ in Compositional CompCert cannot 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 DRF-preservation to some thread-local property.

***A:*** We propose a new compositional simulation $\preccurlyeq$ (see Sec. 4). Based on the simulation 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 lines `4-5` of the left thread in (2.2), the target could be `y=2;x=1` where the writes to `x` and `y` are swapped.

***Q: Can we flip refinement/simulation in spite of non-determinism?*** 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 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.

***A:*** With non-preemptive semantics, the non-determinism occurs only at certain switch points. Then, in our simulation, we require the source and target to switch at the same time and to the same thread. As a result, although the switching step is still non-deterministic, there exists a one-to-one correspondence between the switching steps of the source and of the target. Thus such non-determinism will not affect the flip of our simulation.

***Q: How to support benign races and relaxed machine models?*** It is difficult to write useful DRF programs within *sequential* languages (e.g., CompCert Clight) because they do not provide synchronization primitives (e.g., locks). Nevertheless, we can implement synchronization primitives as external modules so that the threads written in the sequential languages can call functions (e.g., lock-acquire and lock-release) in the external modules. In practice, the efficient implementation $\pi_o$ of the synchronization primitives may be written directly in assembly languages, and may introduce

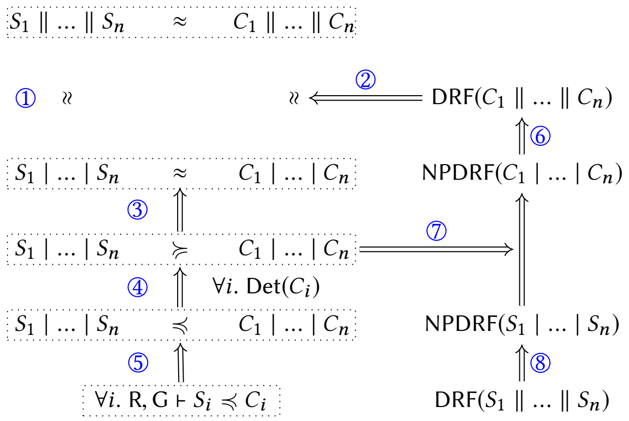
$S_1 \parallel \ldots \parallel S_n \approx C_1 \parallel \ldots \parallel C_n$
①
②
$\mathsf{DRF}(C_1 \parallel \ldots \parallel C_n)$
⑥
$S_1 \mid \ldots \mid S_n \approx C_1 \mid \ldots \mid C_n$
$\mathsf{NPDRF}(C_1 \mid \ldots \mid C_n)$
③
$S_1 \mid \ldots \mid S_n \succcurlyeq C_1 \mid \ldots \mid C_n$
⑦
④
$\forall i.\ \mathsf{Det}(C_i)$
$S_1 \mid \ldots \mid S_n \preccurlyeq C_1 \mid \ldots \mid C_n$
$\mathsf{NPDRF}(S_1 \mid \ldots \mid S_n)$
⑤
⑧
$\forall i.\ \mathsf{R}, \mathsf{G} \vdash S_i \preccurlyeq C_i$
$\mathsf{DRF}(S_1 \parallel \ldots \parallel S_n)$

**Figure 2.** Our basic framework

benign races when used by multiple threads simultaneously (e.g., see Fig. 10 for such an efficient implementation of spin locks). We may view that there is a separate compiler transforming the abstract primitives $\gamma_o$ to their implementations $\pi_o$. But how do we prove the compilation correctness in the case when the target code may contain races?

In addition, our previous discussions all assumed that the source and target programs have sequentially consistent (SC) behaviors. But real-world target machines commonly use relaxed semantics. Although most relaxed models have DRF guarantees saying that DRF programs have SC behaviors in the relaxed semantics, there are no such guarantees for racy programs. It is unclear whether the compilation is still correct when the target code (which may have benign races due to the use of efficient implementations of synchronization primitives) uses relaxed semantics.

*A:* Although the target assembly program using $\pi_o$ may contain races, we do know that the assembly program using $\gamma_o$ is DRF if the source program using $\gamma_o$ is DRF and the compilation is DRF-preserving. We propose a compositional simulation $\pi_o \preccurlyeq^{\mathsf{o}} \gamma_o$, which ensures a strengthened DRF-guarantee theorem for the x86-TSO model. It says, the x86-TSO program using $\pi_o$ behaves the same as the program using $\gamma_o$ in the SC semantics if the latter is DRF.

### 2.4 Frameworks and Key Semantics Components

Figure 2 shows the semantics components and proof steps in our basic framework for DRF and SC target code. The goal of our compilation correctness proof is to show the semantics preservation (i.e., $S_1 \| \ldots \| S_n \approx C_1 \| \ldots \| C_n$ at the top of the figure). This follows the correctness of 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. We do the proofs in the following steps. Double arrows in the figure are logical implications.

**First**, we restrict the compilation to source preemptive code that is 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 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 $\approx$ between their non-preemptive counterparts.

**Second**, DRF of the target programs is obtained through the path ⑥,⑦ and ⑧. We define a notion of DRF for non-preemptive programs (called NPDRF), making it equivalent to DRF, from which we 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**, by composing our footprint-preserving local simulation, the DRF-preserving simulation $\preccurlyeq$ for non-preemptive whole programs can be derived (step ⑤). Given 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).

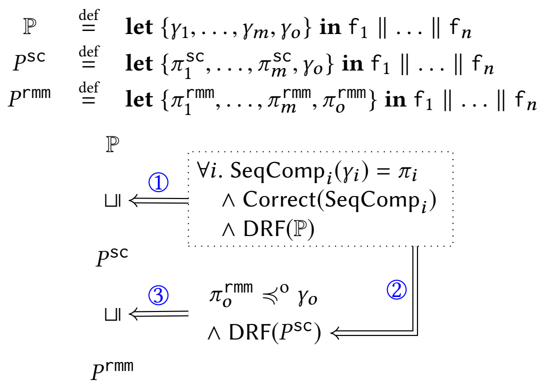
$\mathbb{P} \overset{\text{def}}{=} \textbf{let } \{\gamma_1, \ldots, \gamma_m, \gamma_o\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$
$P^{\mathsf{sc}} \overset{\text{def}}{=} \textbf{let } \{\pi_1^{\mathsf{sc}}, \ldots, \pi_m^{\mathsf{sc}}, \gamma_o\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$
$P^{\mathsf{rmm}} \overset{\text{def}}{=} \textbf{let } \{\pi_1^{\mathsf{rmm}}, \ldots, \pi_m^{\mathsf{rmm}}, \pi_o^{\mathsf{rmm}}\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$
$\mathbb{P}$
①
$\sqsupseteq$
$\forall i.\ \mathsf{SeqComp}_i(\gamma_i) = \pi_i$
$\wedge\ \mathsf{Correct}(\mathsf{SeqComp}_i)$
$\wedge\ \mathsf{DRF}(\mathbb{P})$
$P^{\mathsf{sc}}$
②
③
$\sqsupseteq$
$\pi_o^{\mathsf{rmm}} \preccurlyeq^{\mathsf{o}} \gamma_o$
$\wedge\ \mathsf{DRF}(P^{\mathsf{sc}})$
$P^{\mathsf{rmm}}$

**Figure 3.** The extended framework

***The extended framework.*** Figure 3 shows our ideas for adapting the results of Fig. 2 to relaxed target models and to allow the target programs to contain confined benign races. The goal of the compilation correctness proof is still to show the refinement $\mathbb{P} \sqsupseteq P^{\mathsf{rmm}}$. Here the source $\mathbb{P}$ contains a library module $\gamma_o$ of the abstract synchronization primitives, as well as normal client modules $\gamma_1, \ldots, \gamma_m$. Threads in $\mathbb{P}$ can call functions in these modules. In the target code $P^{\mathsf{rmm}}$, each client module $\gamma_i$ is compiled to $\pi_i$ using a sequential compiler $\mathsf{SeqComp}_i$. The library module $\gamma_o$ is implemented as $\pi_o$, which may contain benign races. The superscript $\mathtt{rmm}$ indicates the use of relaxed memory models.

To prove the refinement, we first apply the results of Fig. 2 and prove $\mathbb{P} \sqsupseteq P^{\mathsf{sc}}$ assuming $\mathsf{DRF}(\mathbb{P})$ and correctness of each $\mathsf{SeqComp}_i$ (step ① in Fig. 3). Here the superscript $\mathtt{sc}$ means the use of SC assembly semantics. Each client module $\pi_i$ in $P^{\mathsf{sc}}$ has the same code as in $P^{\mathsf{rmm}}$ but uses the SC semantics. Note that $P^{\mathsf{sc}}$ still uses the abstract library $\gamma_o$ rather than its racy implementation $\pi_o$. We can view that at this step the library is compiled by an identity transformation.

$$\begin{array}{rccl}
(\mathit{Entry}) & \mathsf{f} & \in & \mathit{String} \\
(\mathit{Prog}) & P, \mathbb{P} & ::= & \textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_n \\
(\mathit{GEnv}) & ge & \in & \mathit{Addr} \rightharpoonup_{\mathsf{fin}} \mathit{Val} \\
(\mathit{MdSet}) & \Pi, \Gamma & ::= & \{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\} \\
(\mathit{Lang}) & tl, sl & ::= & (\mathit{Module},\ \mathit{Core},\ \mathsf{InitCore},\ \longmapsto) \\
(\mathit{Module}) & \pi, \gamma & ::= & \ldots \\
(\mathit{Core}) & \kappa, \Bbbk & ::= & \ldots \\
\mathsf{InitCore} & \multicolumn{3}{l}{\in\ \mathit{Module} \rightarrow \mathit{Entry} \rightharpoonup \mathit{Core}} \\
\longmapsto & \multicolumn{3}{l}{\in\ \mathit{FList} \times (\mathit{Core} \times \mathit{State})\ \rightarrow} \\
 & \multicolumn{3}{l}{\qquad \mathcal{P}((\mathit{Msg} \times \mathit{FtPrt}) \times ((\mathit{Core} \times \mathit{State}) \cup \textbf{abort}))} \\
(\mathit{ThrdID}) & \mathtt{t} & \in & \mathbb{N} \\
(\mathit{Addr}) & l & ::= & \ldots \\
(\mathit{Val}) & v & ::= & l \mid \ldots \\
(\mathit{FList}) & F, \mathbb{F} & \in & \mathcal{P}^{\omega}(\mathit{Addr}) \\
(\mathit{State}) & \sigma, \Sigma & \in & \mathit{Addr} \rightharpoonup_{\mathsf{fin}} \mathit{Val} \\
(\mathit{FtPrt}) & \delta, \Delta & ::= & (rs, ws) \qquad \text{where } rs, ws \in \mathcal{P}(\mathit{Addr}) \\
(\mathit{Msg}) & \iota & ::= & \tau \mid e \mid \mathtt{ret} \mid \mathsf{EntAtom} \mid \mathsf{ExtAtom} \\
(\mathit{Event}) & e & ::= & \ldots \\
(\mathit{Config}) & \phi, \Phi & ::= & (\kappa, \sigma) \mid \textbf{abort}
\end{array}$$

**Figure 4.** The abstract concurrent language

Figure 2 also shows DRF preservation, so we know DRF($P^{\mathsf{sc}}$) given DRF($\mathbb{P}$) (step ② in Fig. 3). The last step ③ is our strengthened DRF-guarantee theorem. We require a compositional simulation relation $\pi_o^{\mathsf{rmm}} \preccurlyeq^{\mathsf{o}} \gamma_o$ holds, saying that $\pi_o$ in the relaxed semantics indeed implements $\gamma_o$.

The approach not only supports racy implementations of synchronization primitives, but also applies in more general cases when $\pi_o$ is a racy implementation of a general concurrent object such as a stack or a queue. For instance, $\pi_o$ could be the Treiber stack implementation [30], and then $\gamma_o$ could be an atomic abstract stack. Then $\pi_o^{\mathsf{rmm}} \preccurlyeq^{\mathsf{o}} \gamma_o$ can be viewed as a correctness criterion of the object implementation in a relaxed model. It ensures a kind of "contextual refinement", i.e., any client threads using $\pi_o$ (in relaxed semantics) generate no more observable behaviors than using $\gamma_o$ instead (in SC semantics), as long as the clients using $\gamma_o$ is DRF.

Although we expect the approach is general enough to work for different relaxed machine models, so far we have only proved the result for `x86-TSO`, as shown in Sec. 7.3.

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 below.

## 3 The Language and Semantics

### 3.1 The Abstract Language

Figure 4 shows the syntax and the state model of an abstract language for preemptive concurrent programming. A program $P$ consists of $n$ threads, each with an entry $\mathsf{f}$ that labels a code segment in a module in $\Pi$. In high-level languages such as Clight, $\mathsf{f}$ is usually the name of a function in a module. A module declaration in $\Pi$ is a triple consisting of the language declaration $tl$, the global environment $ge$, and the code $\pi$. Here $ge$ contains the global variables declared in the module. It is a finite partial mapping from a global variable's address to its initial value.

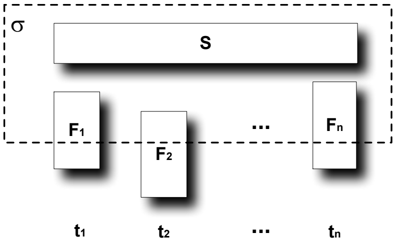

**Figure 5.** The state model

The abstract module language $tl$ is defined as a tuple ($\mathit{Module}$, $\mathit{Core}$, $\mathsf{InitCore}$, $\longmapsto$), whose components can be instantiated for different concrete languages. *Module* describes the syntax of programs. As in Compositional CompCert [29], *Core* is the set of internal "core" states $\kappa$, such as control continuations or register files. The function $\mathsf{InitCore}$ returns an initial "core" state $\kappa$ whenever a thread is created or an external function call is made. In this paper we mainly focus on threads as different modules, and omit the external calls 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. We use $\mathcal{P}(S)$ to represent the powerset of the set $S$.

***Module-local semantics.*** The local execution step inside a module is in the form of $F \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma')$. The *global* memory state $\sigma$ is a finite partial mapping from memory addresses to values.[2] Each step is also labeled with a message $\iota$ and a footprint $\delta$.

Each module also has a *free list* $F$, an *infinite* set of memory addresses. It models the preserved space for allocating local stack frames. Initially we require $F \cap \mathsf{dom}(\sigma) = \emptyset$, and the only memory accessible by the module is the statically allocated global variables declared in the $ge$ of all modules, represented as the *shared* part $S$ in Fig. 5. The local execution of a module may allocate memory from its $F$,[3] which enlarges the state $\sigma$. So "$F - \mathsf{dom}(\sigma)$" is the set of free addresses, depicted in Fig. 5 as the part outside of the boundary of $\sigma$. The memory allocated from $F$ is exclusively owned by this module. $F$ for different modules must be *disjoint* (see the Load rule in Fig. 7).

The messages $\iota$ contain information about the module-local steps. Here we only consider externally observable

[2] In our Coq implementation, we use the more concrete CompCert's block-based memory model, where memory addresses $l$ are instantiated as pairs of block IDs and offsets. This allows us to reuse the CompCert code.

[3] The readers should not confuse the allocation from $F$ with dynamic heap memory allocation like `malloc`. The former is for allocation of stack frames only. For the latter, we assume there is a designated module implementing `malloc`, whose free memory blocks are pointed to by a global variable in its $ge$. Therefore, these memory blocks are in $S$ in Fig. 5, but not in $F$.

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 a special symbol $\tau$. The label $\tau$ 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. The messages define the protocols of communications with the global whole-program semantics (presented below).

The footprint $\delta$ is a pair $(rs, ws)$ of the read set and write set of memory locations in this step.[4] We write emp for the footprint where both sets are empty.

Note we use two sets of symbols to distinguish the source and the target level notations. Symbols such as $\mathbb{P}$, $\mathbb{F}$, $\Bbbk$, $\Sigma$, $\Gamma$ and $\gamma$ are used for the source. Their counterparts, $P$, $F$, $\kappa$, $\sigma$, $\Pi$ and $\pi$, are for the target.

***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 formulate these requirements in Def. 1. It gives us an extensional interpretation of footprints.

**Definition 1** (Well-Defined Languages). wd($tl$) iff , for any execution step $F \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$ in this language, all of the following hold (some auxiliary definitions are in Fig. 6):

(1) $\mathsf{forward}(\sigma, \sigma')$;
(2) $\mathtt{LEffect}(\sigma, \sigma', \delta, F)$;
(3) For any $\sigma_1$, if $\mathtt{LEqPre}(\sigma, \sigma_1, \delta, F)$, then there exists $\sigma_1'$ such that $F \vdash (\kappa, \sigma_1) \xmapsto[\delta]{\iota} (\kappa', \sigma_1')$ and $\mathtt{LEqPost}(\sigma', \sigma_1', \delta, F)$.
(4) Let $\delta_0 = \bigcup\{\delta \mid \exists \kappa', \sigma'.\, F \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau} (\kappa', \sigma')\}$. For any $\sigma_1$, if $\mathtt{LEqPre}(\sigma, \sigma_1, \delta_0, F)$, then for any $\kappa_1', \sigma_1', \iota_1, \delta_1$, $F \vdash (\kappa, \sigma_1) \xmapsto[\delta_1]{\iota_1} (\kappa_1', \sigma_1') \implies \exists \sigma'.\, F \vdash (\kappa, \sigma) \xmapsto[\delta_1]{\iota_1} (\kappa_1', \sigma')$.

It requires that a step may enlarge the memory domain but cannot reduce it (Item (1)), and the additional memory should be allocated from $F$ and included in the write set (Item (2)). Item (2) also requires that the memory out of the write set should keep unchanged. Item (3) says that the memory updates and allocation only depend on the memory content in the read set, the availability of the memory cells in the write 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 in $\delta_0$. Here we do not relate $\sigma_1'$ and $\sigma'$, which can be derived from Item (3).

We have proved in Coq that some real languages satisfy wd, including Clight, Cminor, and x86 assembly [13].

$$
\begin{array}{rcl}
\sigma \overset{rs}{=\!=} \sigma' & \text{iff} & \forall l \in rs.\ l \notin (\mathsf{dom}(\sigma) \cup \mathsf{dom}(\sigma')) \vee \\
 & & \quad l \in (\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma')) \wedge \sigma(l) = \sigma'(l) \\
\delta \subseteq \delta' & \text{iff} & (\delta.rs \subseteq \delta'.rs) \wedge (\delta.ws \subseteq \delta'.ws) \\
\delta \cup \delta' & \overset{\text{def}}{=} & (\delta.rs \cup \delta'.rs,\ \delta.ws \cup \delta'.ws)
\end{array}
$$

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

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

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

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

**Figure 6.** Auxiliary definitions about states and footprints

### 3.2 The Global Preemptive Semantics

Figure 7 shows the global states and selected global semantics rules to model the interaction between modules. The world $W$ consists of the thread pool $T$, the ID $\mathtt{t}$ of the current thread, a bit $d$ indicating whether the current thread is in an atomic block or not, and the memory state $\sigma$. 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 $\kappa$.[5]

The Load rule in Fig. 7 shows the initialization of the world from the program. The memory $\sigma$ is initialized as $\mathsf{GE}(\Pi)$, the union of $ge$ from all the modules. The union is defined only if all the $ge$'s are compatible. The rule also requires that $\sigma$ contain no wild pointers. This requirement is formalized as $\mathsf{closed}(\sigma)$ in Fig. 7, which says the addresses stored in $\sigma$ must be also in $\mathsf{dom}(\sigma)$.

Global transitions of $W$ are also labeled with footprints $\delta$ and messages $o$. Each global step executes the module locally and processes the message of the local transition. 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. The Switch rule shows that context-switch can occur at any program point outside of atomic blocks ($d = 0$).

Below we write $F \vdash \phi \xmapsto[\delta]{\tau}{}^{*} \phi'$ for zero or multiple silent steps, where $\delta$ is the accumulation of the footprint of each step. $F \vdash \phi \xmapsto[\delta]{\tau}{}^{+} \phi'$ represents at least one step. Similar notations are used for global steps. Also $W \Rightarrow^{*} W'$ is for zero or multiple steps that either are silent or produce sw events.

***Event-trace refinement and equivalence.*** An externally observable event trace $\mathcal{B}$ is a finite or infinite sequence of external events $e$, and may end with a termination marker

[4] In our Coq code, a footprint contains two additional fields *cmps* and *frees* for operations that observe and modify memory permissions. They allow footprints to capture memory operations more precisely, but they are orthogonal to our main ideas for supporting concurrency and hence merged into *rs* and *ws* in the paper to simplify the presentation.

[5] As we mentioned, our Coq implementation supports external function calls, so $T$ actually maps a thread ID to a *stack* of triples $(tl, F, \kappa)$. The Coq code also contains additional semantics rules for external calls. The formalization reuses the definitions in Compositional CompCert.

$$
\begin{array}{rrcl@{\qquad}rrcl}
(\mathit{World}) & W, \mathbb{W} & ::= & (T, \mathtt{t}, d, \sigma) & (\mathit{AtomBit}) & d & ::= & 0 \mid 1 \\
(\mathit{NPWorld}) & \widehat{W}, \widehat{\mathbb{W}} & ::= & (T, \mathtt{t}, \mathbb{d}, \sigma) & (\mathit{GMsg}) & o & ::= & \tau \mid e \mid \mathsf{sw} \\
(\mathit{AtomBits}) & \mathbb{d} & ::= & \multicolumn{5}{l}{\{\mathtt{t}_1 \rightsquigarrow d_1, \ldots, \mathtt{t}_n \rightsquigarrow d_n\}} \\
(\mathit{ThrdPool}) & T, \mathbb{T} & ::= & \multicolumn{5}{l}{\{\mathtt{t}_1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, \mathtt{t}_n \rightsquigarrow (tl_1, F_n, \kappa_n)\}}
\end{array}
$$

$$
\mathsf{GE}(\{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\}) \overset{\text{def}}{=}
\begin{cases}
\bigcup_{i=1}^{m} ge_i, & \text{if } \forall i, j.\ ge_i \xlongequal{\mathsf{dom}(ge_i) \cap \mathsf{dom}(ge_j)} ge_j \\
\text{undefined}, & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{lll}
\mathsf{closed}(S, \sigma) & \text{iff} & \forall l, l'.\, l \in S \wedge l' = \sigma(l) \Longrightarrow l' \in S \\
\mathsf{closed}(\sigma) & \text{iff} & \mathsf{closed}(\mathsf{dom}(\sigma), \sigma)
\end{array}
$$

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

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\kappa', \sigma')}{(T, \mathtt{t}, d, \sigma) \underset{\delta}{\overset{\tau}{\Rightarrow}} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, d, \sigma')}\ \tau\text{-step}
$$

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathsf{EntAtom}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\mathsf{emp}]{\tau} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, 1, \sigma)}\ \mathsf{EntAt}
$$

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathsf{ExtAtom}} (\kappa', \sigma)}{(T, \mathtt{t}, 1, \sigma) \xRightarrow[\mathsf{emp}]{\tau} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, 0, \sigma)}\ \mathsf{ExtAt}
$$

$$
\frac{\mathtt{t}' \in \mathsf{dom}(T)}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\mathsf{emp}]{\mathsf{sw}} (T, \mathtt{t}', 0, \sigma)}\ \mathsf{Switch}
$$

$$
\frac{\begin{array}{c}
T(\mathtt{t}) = (tl, F, \kappa) \quad \mathbb{d}(\mathtt{t}) = 0 \quad \mathtt{t}' \in \mathsf{dom}(T) \\
F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathsf{EntAtom}} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}
\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\xRightarrow[\mathsf{emp}]{\mathsf{sw}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 1\}, \sigma)}\ \mathsf{EntAt}_{\mathsf{np}}
$$

$$
\frac{\begin{array}{c}
T(\mathtt{t}) = (tl, F, \kappa) \quad \mathbb{d}(\mathtt{t}) = 1 \quad \mathtt{t}' \in \mathsf{dom}(T) \\
F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathsf{ExtAtom}} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}
\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\xRightarrow[\mathsf{emp}]{\mathsf{sw}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 0\}, \sigma)}\ \mathsf{ExtAt}_{\mathsf{np}}
$$

**Figure 7.** Preemptive and non-preemptive global semantics

**done** or an abortion marker **abort**. Following the definition in CompCert, we use $P \sqsubseteq \mathbb{P}$ to represent the event-trace refinement, and $P \approx \mathbb{P}$ for equivalence.

### 3.3 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. We write **let** $\Pi$ **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ or $\hat{P}$ for the program with non-preemptive semantics, to distinguish it from the preemptive concurrency.

The 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.

The last two rules in Fig. 7 define the non-preemptive global steps $\widehat{W} :\underset{\delta}{\overset{o}{\Rightarrow}} \widehat{W}'$. More rules are in the supplementary TR [13]. There is no rule like Switch of the preemptive semantics, since context-switch occurs only at synchronization points. $\mathsf{EntAt}_{\mathsf{np}}$ and $\mathsf{ExtAt}_{\mathsf{np}}$ execute one step of the current thread $\mathtt{t}$, and then non-deterministically switch to a thread $\mathtt{t}'$. The corresponding global steps produce the $\mathsf{sw}$ events.

## 4 The Footprint-Preserving 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 DRF.

***Footprint consistency.*** As in CompCert, the simulation requires that the source and the target generate the same external events. In addition, it also requires 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. 5. DRF informally requires that the threads never have conflicting accesses to the memory in $\mathbb{S}$ at the same time.

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

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

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 $\mu$ to be well-formed, defined as $\mathsf{wf}(\mu)$ in Fig. 8.

Then, given footprints $\Delta$ and $\delta$, we define their consistency with respect to $\mu$ as $\mathsf{FPmatch}(\mu, \Delta, \delta)$ in Fig. 8. It says the *shared* locations in $\delta$ must be contained in $\Delta$, modulo the mapping $\mu.f$. We only consider the shared locations in $\mu.S$ because accesses of local memory would *not* cause races. The shared locations in $\delta.\mathit{rs}$ of the target module are allowed to come from $\Delta.\mathit{ws}$ of the source, since transforming a write to a read would not introduce more races.

***Rely/guarantee conditions.*** We use rely and guarantee conditions to specify interaction between modules. They need to enforce the view of accessibility of shared and local memory in Fig. 5. More specifically, a module expects others to keep its local memory (in $\mathbb{F}$) intact. In addition, although other modules may update the shared memory $\mathbb{S}$, they must preserve certain properties of $\mathbb{S}$. One such property is $\mathsf{closed}(\mathbb{S}, \Sigma)$ (defined in Fig. 7), which ensures $\mathbb{S}$ cannot contain memory pointers pointing to local memory cells in any $\mathbb{F}_i$. Otherwise

$$
\begin{array}{l}
f\{\!\!\{\mathbb{S}\}\!\!\} \overset{\text{def}}{=} \{l' \mid \exists l.\, l \in \mathbb{S} \wedge f(l) = l'\} \\
f|_{\mathbb{S}} \overset{\text{def}}{=} \{(l, f(l)) \mid l \in (\mathbb{S} \cap \mathsf{dom}(f))\} \\
\mathsf{wf}(\mu) \text{ iff } \mathsf{injective}(\mu.f) \wedge \mathsf{dom}(\mu.f) = \mu.\mathbb{S} \wedge \mu.f\{\!\!\{\mu.\mathbb{S}\}\!\!\} = \mu.S \\
\mathsf{FPmatch}(\mu, \Delta, \delta) \text{ iff } (\delta.rs \cap \mu.S \subseteq \mu.f\{\!\!\{\Delta.rs \cup \Delta.ws\}\!\!\}) \\
\qquad\qquad\qquad\qquad \wedge (\delta.ws \cap \mu.S \subseteq \mu.f\{\!\!\{\Delta.ws\}\!\!\})
\end{array}
$$

$$
\widehat{f}(v) \overset{\text{def}}{=} \begin{cases} v, & \text{if } v \notin Addr \\ f(v), & \text{if } v \in Addr \wedge v \in \mathsf{dom}(f) \\ \text{undefined}, & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathsf{Inv}(f, \Sigma, \sigma) \text{ iff } \forall l, l'.\ l \in \mathsf{dom}(\Sigma) \wedge f(l) = l' \\
\qquad\qquad\qquad \Longrightarrow l' \in \mathsf{dom}(\sigma) \wedge \widehat{f}(\Sigma(l)) = \sigma(l') \\
\mathsf{HG}(\Delta, \Sigma, \mathbb{F}, \mathbb{S}) \text{ iff } \Delta \subseteq (\mathbb{F} \cup \mathbb{S}) \wedge \mathsf{closed}(\mathbb{S}, \Sigma) \\
\mathsf{LG}(\mu, (\delta, \sigma, F), (\Delta, \Sigma)) \text{ iff } \delta \subseteq (F \cup \mu.S) \wedge \mathsf{closed}(\mu.S, \sigma) \\
\qquad\qquad\qquad\qquad \wedge \mathsf{FPmatch}(\mu, \Delta, \delta) \wedge \mathsf{Inv}(\mu.f, \Sigma, \sigma) \\
\mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S}) \text{ iff } (\Sigma \overset{\mathbb{F}}{=\!=} \Sigma') \wedge \mathsf{closed}(\mathbb{S}, \Sigma') \wedge \mathsf{forward}(\Sigma, \Sigma') \\
\mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F)) \text{ iff } \mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mu.\mathbb{S}) \wedge \mathsf{R}(\sigma, \sigma', F, \mu.S) \\
\qquad\qquad\qquad\qquad \wedge \mathsf{Inv}(\mu.f, \Sigma', \sigma')
\end{array}
$$

$$
\lfloor\varphi\rfloor(ge) \overset{\text{def}}{=} \begin{cases} \{(\varphi(l), \widehat{\varphi}(v)) \mid (l, v) \in ge\}, & \\ \qquad \text{if } (\mathsf{dom}(ge) \cup (\mathsf{range}(ge) \cap Addr)) \subseteq \mathsf{dom}(\varphi) & \\ \text{undefined}, & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathsf{initM}(\varphi, ge, \Sigma, \sigma) \text{ iff } ge \subseteq \Sigma \wedge \mathsf{closed}(\Sigma) \\
\qquad\qquad\qquad \wedge \mathsf{dom}(\sigma) = \varphi\{\!\!\{\mathsf{dom}(\Sigma)\}\!\!\} \wedge \mathsf{Inv}(\varphi, \Sigma, \sigma)
\end{array}
$$

**Figure 8.** Footprint matching and rely/guarantee conditions

a thread $\mathtt{t}_j$ can update the memory in $\mathbb{F}_i$ by tracing these pointers. [6] Also the invariant Inv should be preserved, which relates the contents of the corresponding memory locations in $\Sigma$ and $\sigma$. It expresses the same thing as memory injection in CompCert [6]. We encode these requirements in the rely condition Rely in Fig. 8, and define the guarantee conditions HG and LG correspondingly.

***The simulation.*** Below we define $(sl, ge, \gamma) \preccurlyeq_{\varphi} (tl, ge', \pi)$ to relate the *non-preemptive* executions of the source module $(sl, ge, \gamma)$ and the target one $(tl, ge', \pi)$. The *injective function* $\varphi$ maps source addresses to the target ones.

**Definition 2** (Module-Local Downward Simulation)**.**
$(sl, ge, \gamma) \preccurlyeq_{\varphi} (tl, ge', \pi)$ iff

1. $\lfloor\varphi\rfloor(ge) = ge'$; and
2. for all $\mathsf{f}$, $\mathbb{k}$, $\Sigma$, $\sigma$, $\mathbb{F}$, $F$, and $\mu = (\mathsf{dom}(\Sigma), \mathsf{dom}(\sigma), \varphi|_{\mathsf{dom}(\Sigma)})$, if $sl.\mathsf{InitCore}(\gamma, \mathsf{f}) = \mathbb{k}$, $\mathbb{F} \cap \mathsf{dom}(\Sigma) = F \cap \mathsf{dom}(\sigma) = \emptyset$, and $\mathsf{initM}(\varphi, ge, \Sigma, \sigma)$, then there exist $i \in \mathtt{index}$ and $\kappa$ such that $tl.\mathsf{InitCore}(\pi, \mathsf{f}) = \kappa$, and
$$(\mathbb{F}, (\mathbb{k}, \Sigma), \mathsf{emp}) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \mathsf{emp}),$$
where $(\mathbb{F}, (\mathbb{k}, \Sigma), \Delta) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \delta)$ is defined in Def. 3.

It says that, starting from some core states $\mathbb{k}$ and $\kappa$, with any states ($\Sigma$ and $\sigma$) and free lists ($\mathbb{F}$ and $F$) satisfying some initial constraints, we have the simulation $(\mathbb{F}, (\mathbb{k}, \Sigma), \mathsf{emp}) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \mathsf{emp})$ defined in Def. 3. Here $ge$ and $ge'$ must be related through $\lfloor\varphi\rfloor$ (see Fig. 8). The initial states $\Sigma$ and $\sigma$ also need to be related with $ge$ and $\varphi$ through initM.

**Definition 3.** $(\mathbb{F}, (\mathbb{k}, \Sigma), \Delta_0) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \delta_0)$ is the largest relation such that, whenever $(\mathbb{F}, (\mathbb{k}, \Sigma), \Delta_0) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \delta_0)$, then the following are true:

1. for all $\mathbb{k}'$, $\Sigma'$ and $\Delta$, if $\mathbb{F} \vdash (\mathbb{k}, \Sigma) \overset{\tau}{\underset{\Delta}{\longmapsto}} (\mathbb{k}', \Sigma')$ and $(\Delta_0 \cup \Delta) \subseteq (\mathbb{F} \cup \mu.\mathbb{S})$, then one of the following holds:
   a. $\exists j < i.\ (\mathbb{F}, (\mathbb{k}', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^{j}_{\mu} (F, (\kappa, \sigma), \delta_0)$, or
   b. there exist $\kappa'$, $\sigma'$, $\delta$ and $j$ such that:
      i. $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{+} (\kappa', \sigma')$;
      ii. $(\delta_0 \cup \delta) \subseteq (F \cup \mu.S)$ and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0 \cup \delta)$; and
      iii. $(\mathbb{F}, (\mathbb{k}', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^{j}_{\mu} (F, (\kappa', \sigma'), \delta_0 \cup \delta)$.
2. for all $\mathbb{k}'$ and $\iota$, if $\mathbb{F} \vdash (\mathbb{k}, \Sigma) \overset{\iota}{\underset{\mathsf{emp}}{\longmapsto}} (\mathbb{k}', \Sigma)$, $\iota \neq \tau$, and $\mathsf{HG}(\Delta_0, \Sigma, \mathbb{F}, \mu.\mathbb{S})$, there exist $\kappa'$, $\delta$, $\sigma'$ and $\kappa''$ such that:
   a. $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{*} (\kappa', \sigma')$, and $F \vdash (\kappa', \sigma') \overset{\iota}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa'', \sigma')$, and
   b. $\mathsf{LG}(\mu, (\delta_0 \cup \delta, \sigma', F), (\Delta_0, \Sigma))$, and
   c. for all $\sigma''$ and $\Sigma'$, if $\mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma', \sigma'', F))$, then there exists $j$ such that
   $$(\mathbb{F}, (\mathbb{k}', \Sigma'), \mathsf{emp}) \preccurlyeq^{j}_{\mu} (F, (\kappa'', \sigma''), \mathsf{emp}).$$

The simulation $(\mathbb{F}, (\mathbb{k}, \Sigma), \Delta_0) \preccurlyeq^{i}_{\mu} (F, (\kappa, \sigma), \delta_0)$ carries $\Delta_0$ and $\delta_0$, the footprints accumulated at the source and the target, respectively. The definition follows the diagram in Fig. 1(d). For every $\tau$-step in the source (case 1), if the newly generated footprints and the accumulated $\Delta_0$ 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 corresponds to zero or multiple $\tau$-steps in the target, and the simulation holds over the resulting states with the accumulated footprints and a new index $j$. We carry the well-founded index to ensure the simulation preserves termination.

If the source step corresponds to at least one target steps (case 1-b), the footprints at the target must also be in scope and be consistent with the source level footprints. 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 $\iota$ (case 2), if the footprints and states satisfy the high-level guarantee HG, the target must be able to generate the same $\iota$, and the accumulated footprints and the state satisfy the low-level guarantee LG. We also need to consider the interaction with other modules or threads. For any environment steps satisfying Rely, the simulation must hold 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

[6] We disallow cross-module escape of pointers pointing to stack-allocated variables. (We still allow the escape within a module and the transfer of dynamically allocated heap data structures.) Our TR [13] presents the framework with the support of stack pointer escape. Adding the support looks relatively orthogonal to our main ideas for supporting concurrency, and we can follow the approach of Compositional CompCert (the part for stack pointer escape).

clear the accumulated footprints. Rely and the guarantees HG and LG are defined in Fig. 8, as explained before.

Note that each case in Def. 3 has prerequisites about the source level footprints (e.g. the footprints are in scope or satisfy HG). We need to prove that these requirements indeed hold at the source level, to make the simulation meaningful instead of being vacuously true. We formalize these requirements separately as ReachClose in Def. 4. It is a simplified version of the *reach-close* concept by Stewart et al. [29] (simplified because we disallow the escape of local stack pointers into the shared memory). The compilation correctness assumes that all the source modules satisfy ReachClose (see Lem. 6 and Def. 11 below).

**Definition 4** (Reach Closed Module)**.** ReachClose$(sl, ge, \gamma)$ iff , for all $\mathsf{f}$, $\mathbb{k}$, $\Sigma$, $\mathbb{F}$ and $\mathbb{S}$, if $sl.\mathsf{InitCore}(\gamma, \mathsf{f}) = \mathbb{k}$, $\mathbb{S} = \mathsf{dom}(\Sigma)$, $ge \subseteq \Sigma$, $\mathbb{F} \cap \mathbb{S} = \emptyset$, and $\mathsf{closed}(\mathbb{S}, \Sigma)$, then $\mathsf{RC}(\mathbb{F}, \mathbb{S}, (\mathbb{k}, \Sigma))$.

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

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

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

**Lemma 5** (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 $(sl, ge, \gamma) \preccurlyeq_{\varphi' \circ \varphi} (tl, ge'', \pi)$.

The proof of Lem. 5 relies on the auxiliary lemmas similar to "memory interpolations" in Compostional CompCert.

**Lemma 6** (Compositionality, ⑤ in Fig. 2)**.**
For any $\mathsf{f}_1, \ldots, \mathsf{f}_n, \varphi, \Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, 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 $\quad \textbf{let}\ \Gamma\ \textbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n \preccurlyeq \textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \mid \ldots \mid \mathsf{f}_n\ .$

Lemma 6 shows the compositionality of our simulation. The whole program *downward* simulation $\hat{\mathbb{P}} \preccurlyeq \hat{P}$ relates the non-preemptive execution of the whole source program $\mathbb{P}$ and target program $P$. The definition is given in TR [13].

With determinism of the target module language (written as $\mathsf{det}(tl)$), we can flip $\hat{\mathbb{P}} \preccurlyeq \hat{P}$ to derive the upward simulation $\hat{P} \leqslant \hat{\mathbb{P}}$. We give the definition of $\mathsf{det}(tl)$ and the Flip Lemma (④ in Fig. 2) in TR [13]. Lemma 7 shows the non-preemptive global simulation ensures the refinement.

**Lemma 7** (Soundness, ③ in Fig. 2)**.** If $\hat{P} \leqslant \hat{\mathbb{P}}$, then $\hat{P} \sqsubseteq \hat{\mathbb{P}}$.

$$\frac{P \overset{load}{\Longrightarrow} W \qquad W \Rightarrow^{*} W' \qquad W' \Longmapsto \mathsf{Race}}{P \Longmapsto \mathsf{Race}}$$

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

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

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

**Figure 9.** Data races in preemptive semantics

## 5 Data-Race-Freedom

Below we first define the conflict of footprints.

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

Recall that, when used as a set, $\delta$ 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 $\delta$ 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 $(\delta_1, d_1)$ and $(\delta_2, d_2)$ are conflicting if $\delta_1$ and $\delta_2$ are conflicting and at least one of $d_1$ and $d_2$ is 0.

We define data races in Fig. 9 for preemptive semantics. In the Race rule, $W$ steps to Race if there are conflicting footprints of two threads predicted from the current configuration through $\mathtt{predict}(W, \mathtt{t}, (\delta, d))$. Then we define $\mathsf{DRF}(P)$:

$$\mathsf{DRF}(P) \ \text{ iff } \ \neg(P \Longmapsto \mathsf{Race})$$

$\mathsf{NPDRF}(\hat{P})$ is defined similarly. We can prove NPDRF is equivalent to DRF (⑥ and ⑧ in Fig. 2). The following Lem. 8 shows the simulation preserves NPDRF. Given the equivalence, we know it also preserves DRF.

**Lemma 8** (NPDRF Preservation, ⑦ in Fig. 2)**.**
For any $\hat{\mathbb{P}}$, $\hat{P}$, if $\hat{P} \leqslant \hat{\mathbb{P}}$, and $\mathsf{NPDRF}(\hat{\mathbb{P}})$, then $\mathsf{NPDRF}(\hat{P})$.

**Lemma 9** (Semantics Equivalence, ① and ② in Fig. 2)**.**
For any $\Pi$, $\mathsf{f}_1, \ldots, \mathsf{f}_m$, if $\mathsf{DRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_m)$, then $\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 | \ldots | \mathsf{f}_m \approx \textbf{let}\ \Pi\ \textbf{in}\ \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_m$.

## 6 The Final Theorem

Putting all the previous results together, we are able to prove our final theorem in the basic framework (Fig. 2). We first model a sequential compiler SeqComp as follows:

$$\mathsf{SeqComp} ::= (\mathsf{CodeT}, \varphi)\,, \quad \text{where } \mathsf{CodeT} \in \mathit{Module} \rightharpoonup \mathit{Module}$$

As the key proof obligation, we need to verify that each SeqComp is Correct. The correctness is defined based on our footprint-preserving module-local simulation. Recall that $\lfloor \varphi \rfloor$ is defined in Fig. 8.

**Definition 10** (Sequential Compiler Correctness).
Correct(SeqComp, $sl$, $tl$) iff

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

The desired correctness GCorrect (Def. 11) of *concurrent* program compilation is the semantics preservation of whole programs, i.e., every target concurrent program is a refinement of the source. Here all the SeqComp must agree on the transformation $\varphi$ of global environments (see item 1 below).

**Definition 11** (Concurrent Compiler Correctness).
GCorrect((SeqComp$_1$, $sl_1$, $tl_1$), . . . , (SeqComp$_m$, $sl_m$, $tl_m$)) iff for any $\varphi$, $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, if

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

then $\quad \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsubseteq \textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

Our final theorem is then formulated as Thm. 12. 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 12** (Final Theorem).
For any SeqComp$_1$, . . . , 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(SeqComp$_i$, $sl_i$, $tl_i$), then

GCorrect((SeqComp$_1$, $sl_1$, $tl_1$), . . . , (SeqComp$_m$, $sl_m$, $tl_m$)).

## 7 The CASCompCert Compiler

We apply our framework to develop CASCompCert. It uses CompCert-3.0.1 [6] for compilation of multi-threaded Clight programs to `x86-SC`, i.e. x86 with SC semantics. We further extend the framework (as shown in Fig. 3) to support `x86-TSO` [28] as the target language. The source program we compile consists of multiple sequential Clight threads. Inter-thread synchronization can be achieved through *external calls* to an external module. Since the external synchronization module can be shared by the threads, below we also refer to it as an *object* and the Clight threads as *clients.*

As a tiny example, Fig. 10(a) shows a spin-lock implementation in a simple imperative language which we call CImp. $\langle C \rangle$ is an atomic block, which cannot be interrupted by other threads. The EntAtom and ExtAtom events are generated at the beginning and the end of the atomic block respectively. The command `assert`($B$) aborts if $B$ is false. This source implementation of locks serves as an abstract specification, which is *manually* translated to `x86-TSO`, as shown in Fig. 10(b). The implementation is similar to the Linux spin-lock implementation (a.k.a. the TTAS lock [11]). To acquire the lock, the `l_acq` block reads and resets the lock bit to 0 atomically by the `lock`-prefixed `cmpxchgl`, which ensures mutual exclusion. The `spin` loop reads the lock bit until it appears to be available (i.e., not 0). Note that the load and store operations in the `spin` and `unlock` blocks are *not* `lock`-prefixed. This optimization introduces benign races. With the external lock module, we can implement a DRF counter `inc` in Clight, as shown in Fig. 10(c). An example whole program $\mathbb{P}$ is **let** $\{\gamma_{\mathrm{C}}, \gamma_{\mathrm{lock}}\}$ **in** `inc() || inc()`.

```
lock(){  r := 0; while(r==0){ <r:=[L]; [L]:=0;> }  }
unlock(){  < r := [L]; assert(r == 0); [L] := 1; > }
```

(a) lock specification $\gamma_{\mathrm{lock}}$

```
lock:     movl      $L,       %ecx
          movl      $0,       %edx
l_acq:    movl      $1,       %eax
     lock cmpxchgl  %edx,     (%ecx)
          je        enter
spin:     movl      (%ecx),   %ebx
          cmp       $0,       %ebx
          je        spin
          jmp       l_acq
enter:    retl
unlock:   movl      $L,       %eax
          movl      $1,       (%eax)
          retl
```

(b) lock implementation $\pi_{\mathrm{lock}}$

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

(c) a Clight client $\gamma_{\mathrm{C}}$

**Figure 10.** Lock as an external module of Clight programs

As shown in Fig. 3, we compile the code in two steps. First, we use CompCert to compile the Clight client to `x86-SC`, but leave object code (e.g., $\gamma_{\mathrm{lock}}$ in Fig. 10) untouched. Second, we transform the resulting `x86-SC` client code to `x86-TSO`. Syntactically this is an identity transformation, but the semantics changes. Then we manually transform the source object code to `x86-TSO` code. We can prove the resulting whole program in `x86-TSO` preserves the source semantics as long as the `x86-TSO` object code refines its source.

### 7.1 Language Instantiations

We need to first instantiate our abstract languages of Fig. 4 with Clight, `x86-SC` and the intermediate languages introduced in CompCert. We also instantiate it with the simple language CImp for the source object code.

***The Clight language.*** The *Module* in Fig. 4 is instantiated with the same Clight syntax as in CompCert. The core state $\kappa$ is a pair of a local state $c$ and an index $N$ indicating the position of the next block in the freelist $F$ to be allocated, as shown below. InitCore initializes $N$ to 0. Local transitions of Clight are instrumented with footprints.

$$\begin{array}{llllllll} (FList) & F & ::= & b_1 :: b_2 :: \ldots & (Block) & b & \in & \mathbb{N}^+ \\ (BIndex) & N & \in & \mathbb{N} & (Core) & \kappa & ::= & (c, N) \\ (Mem) & \sigma & \in & \multicolumn{5}{l}{Block \rightharpoonup_{\mathrm{fin}} (\mathbb{N} \rightharpoonup \mathtt{val})} \end{array}$$

$$\begin{tikzcd}
\textit{Clight} \arrow[r, Rightarrow, "\texttt{Cshmgen}"] & \textit{C\#minor} \arrow[r, Rightarrow, "\texttt{Cminorgen}"] & \textit{Cminor} \arrow[r, Rightarrow, "\texttt{Selection}"] & \textit{CminorSel} \arrow[r, Rightarrow, "\texttt{RTLgen}"] & \textit{RTL} \arrow[r, Rightarrow, "\texttt{Tailcall, Renumber}"] & \textit{RTL} \arrow[d, Rightarrow, "\texttt{Allocation}"] \\
\textit{x86-SC assembly} & \textit{Mach} \arrow[l, Rightarrow, "\texttt{Asmgen}"'] & \textit{Linear} \arrow[l, Rightarrow, "\texttt{Stacking}"'] & \textit{Linear} \arrow[l, Rightarrow, "\texttt{CleanupLabels}"'] & \textit{LTL} \arrow[l, Rightarrow, "\texttt{Linearize}"'] & \textit{LTL} \arrow[l, Rightarrow, "\texttt{Tunneling}"']
\end{tikzcd}$$

**Figure 11.** Proved CompCert compilation passes

***The source language CImp for objects.*** The instantiation of the abstract language with CImp lets the atomic blocks generate the EntAtom and ExtAtom events. To allow benign races in the target `x86-TSO` code, we need to ensure partition of the client data and the object data. This can be enforced through the permissions in the CompCert memory model. The client programs can only access memory locations whose permission is not None. Therefore we set the permission of the object data (the memory at the location `L` in our example in Fig. 10) to None. Also, we require the CImp program can *only* access memory locations with None permission. It aborts if trying to access memory locations whose permissions are *not* None .

### 7.2 Adapting CompCert

Given the program **let** $\{\gamma_1, \ldots, \gamma_l, \gamma_o\}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$ consisting of Clight modules $\gamma_i$ and the module $\gamma_o$ (we omit *sl* and *ge* in the modules to simplify the presentation), the compilation `Comp` is defined as

$$\mathtt{Comp}(\textbf{let}\ \{\gamma_1, \ldots, \gamma_l, \gamma_o\}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n) \stackrel{\text{def}}{=}$$
$$\textbf{let}\ \{\mathtt{CompCert}(\gamma_1), \ldots, \mathtt{CompCert}(\gamma_l), \mathtt{IdTrans}(\gamma_o)\,\}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$$

where `CompCert` is the adapted compilation consisting of the original CompCert passes, and `IdTrans` is the identity translation which returns the object module unchanged.

**Lemma 13.** Correct(`CompCert`, `Clight`, `x86-SC`).

We have proved Lem. 13, i.e. the original CompCert-3.0.1 passes satisfy our Correct in Def. 10. The verified compilation passes (shown in Fig. 11) include all the translation passes and four optimization passes (`Tailcall`, `Renumber`, `Tunneling` and `CleanupLabels`).[7] Proving other optimization passes would be similar and is left as future work.

We also prove the well-definedness of Clight, `x86-SC`, and CImp, the determinism of `x86-SC` and CImp, and the correctness of `IdTrans` for CImp. Together with our framework's final theorem (Thm. 12), we derive the following result:

**Theorem 14** (Correctness with `x86-SC` backend and obj.)**.** GCorrect((`CompCert`, `Clight`, `x86-SC`), (`IdTrans`, `CImp`, `CImp`)).

To prove Lem. 13, we try to *reuse as much the original CompCert correctness proofs as possible.* We address the following two main challenges in reusing CompCert proofs.

***Converting memory layout.*** Many CompCert lemmas rely on the specific definition of the CompCert memory model, which is different from ours. In CompCert, memory allocations in an execution get *consecutive* natural numbers as block numbers. This fact is used extensively in CompCert's fundamental libraries and its compilation correctness proofs. But it does not hold in our model, where each thread has its own freelist $F$ (an infinite sequence of block numbers). Since the $F$ of different threads must be disjoint, we *cannot* make each $F$ an infinite sequence of consecutive natural numbers to directly simulate CompCert.

```
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', FP.subset fp' fp /\
 eval_expr_fp tge sp e' m' le (sel_expr a) fp' /\
 eval_expr tge sp e' m' le (sel_expr a) v' /\ Val.lessdef v v'.
```

**Figure 12.** Coq code example

Our solution is to 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. This way we reuse most CompCert libraries and compilation proofs without modification.

***Footprint preservation.*** CompCert does not model footprints. Fortunately many of its definitions and lemmas can be slightly modified to support footprint preservation. For instance, Fig. 12 shows a key lemma in the proof of the `Selection` pass, `sel_expr_correct`, with our newly-added code highlighted. It says the selected expression must evaluate to a value refined by the `Cminor` expression. We simply extend the lemma by requiring *the selected expression has smaller footprint while evaluating on related memory.*

### 7.3 `x86-TSO` as the Target Language

Now we show how to generate `x86-TSO` code while preserving the behaviors of the source program. The theorem below shows our final goal. To avoid clutter, below we use *sl* and *tl* to represent $sl_{\mathtt{Clight}}$ and $tl_{\mathtt{x86\text{-}TSO}}$ respectively.

**Theorem 15** (Correctness with `x86-TSO` backend and obj.)**.** For any $f_1 \ldots f_n$, $\Gamma = \{(sl, ge_1, \gamma_1), \ldots, (sl, ge_m, \gamma_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$, and $\Pi = \{(tl, ge'_1, \pi_1), \ldots, (tl, ge'_m, \pi_m), (tl, ge_o, \pi_o)\}$, if

1. $\forall i \in \{1, \ldots, m\}.\ (\mathtt{CompCert.CodeT}(\gamma_i) = \pi_i)\ \wedge\ \mathsf{injective}(\varphi)$ $\wedge\ (\mathtt{CompCert.}\varphi = \varphi)\ \wedge\ \lfloor\varphi\rfloor(ge_i) = ge'_i$,
2. Safe(**let** $\Gamma$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$) and DRF(**let** $\Gamma$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$),
3. $\forall i \in \{1, \ldots, m\}$. ReachClose$(sl, ge_i, \gamma_i)$, and ReachClose$(sl_{\mathtt{CImp}}, ge_o, \gamma_o)$,
4. $(tl, ge_o, \pi_o) \preccurlyeq^{o} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$,

then **let** $\Pi$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsubseteq'$ **let** $\Gamma$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

[7] The compiler option `-g` for insertion of debugging information is disabled.

| Compilation passes and framework | Spec | | Proof | |
|---|---|---|---|---|
| | CompCert | Ours | CompCert | Ours |
| Cshmgen | 515 | 1021 | 1071 | 1503 |
| Cminorgen | 753 | 1556 | 1152 | 1251 |
| Selection | 336 | 500 | 647 | 783 |
| RTLgen | 428 | 543 | 821 | 862 |
| Tailcall | 173 | 328 | 275 | 405 |
| Renumber | 86 | 245 | 117 | 358 |
| Allocation | 704 | 785 | 1410 | 1700 |
| Tunneling | 131 | 339 | 166 | 475 |
| Linearize | 236 | 371 | 349 | 733 |
| CleanupLabels | 126 | 387 | 161 | 388 |
| Stacking | 730 | 1038 | 1108 | 2135 |
| Asmgen | 208 | 338 | 571 | 1128 |
| Compositionality (Lem. 6) | | 580 | | 2249 |
| DRF preservation (Lem. 8) | | 358 | | 1142 |
| Semantics equiv. (Lem. 9) | | 1540 | | 4718 |
| Lifting | | 813 | | 1795 |

**Figure 13.** Lines of code (using `coqwc`) in Coq

Here the premises 1-3 are similar to those required in Def. 11. In addition, the premise 4 requires that the `x86-TSO` code $\pi_o$ of the object be simulated by $\gamma_o$. The simulation $(tl, ge_o, \pi_o) \preccurlyeq^{o} (sl_{\texttt{CImp}}, ge_o, \gamma_o)$ is an extension of Liang and Feng [19] with the support of TSO semantics for the low-level code. Due to space limit, we omit the definition here.

The refinement relation $\sqsubseteq'$ is a weaker version of $\sqsubseteq$ (see Sec. 3.2). It does not preserve termination (the formal definition omitted here). This is because our simulation $\preccurlyeq^{o}$ for the object code does not preserve termination for now, which we leave as future work.

Theorem 15 can be derived from Thm. 14 (for the compilation from Clight to `x86-SC`), and from Lem. 16 below, saying the `x86-TSO` code refines the `x86-SC` client code and the source object code (we use $tl_{\texttt{sc}}$ and $tl_{\texttt{tso}}$ as shorter notations for $tl_{\texttt{x86-SC}}$ and $tl_{\texttt{x86-TSO}}$ respectively).

**Lemma 16** (Restore SC semantics for DRF x86 programs)**.** Let $\Pi_{\texttt{sc}} = \{(tl_{\texttt{sc}}, ge_1, \pi_1), \ldots, (tl_{\texttt{sc}}, ge_m, \pi_m), (sl_{\texttt{CImp}}, ge_o, \gamma_o)\}$, and $\Pi_{\texttt{tso}} = \{(tl_{\texttt{tso}}, ge_1, \pi_1), \ldots, (tl_{\texttt{tso}}, ge_m, \pi_m), (tl_{\texttt{tso}}, ge_o, \pi_o)\}$. For any $f_1 \ldots f_n$, if

1. $\mathrm{Safe}(\textbf{let}\ \Pi_{\texttt{sc}}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n)$ and $\mathrm{DRF}(\textbf{let}\ \Pi_{\texttt{sc}}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n)$,
2. $(tl_{\texttt{tso}}, ge_o, \pi_o) \preccurlyeq^{o} (sl_{\texttt{CImp}}, ge_o, \gamma_o)$,

then $\textbf{let}\ \Pi_{\texttt{tso}}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n \sqsubseteq' \textbf{let}\ \Pi_{\texttt{sc}}\ \textbf{in}\ \texttt{f}_1 \parallel \ldots \parallel \texttt{f}_n$.

As explained before, Lem. 16 can be viewed as a strengthened DRF-guarantee theorem for `x86-TSO` in that, if we let $\gamma_o$ contain only skip and $ge_o = \emptyset$, Lem. 16 implies the DRF-guarantee of `x86-TSO`.

### 7.4 Proof Efforts in Coq

In Coq we have mechanized the framework (Fig. 2) and the extended framework (Fig. 3) and proved all the related lemmas. We have verified all the CompCert passes in Fig. 11.

Statistics of our Coq implementation and proofs are depicted in Fig. 13. Adapting the compilation correctness proofs from CompCert is relatively lightweight. For most passes our proofs are within 300 lines of code more than the original CompCert proofs. The `Stacking` pass introduces more additional proofs, mostly caused by arguments marshalling for supporting cross-language linking. In our experience, adapting CompCert's original compilation proofs to our settings takes less than one person week per translation pass (except for `Stacking`). For simpler passes such as `Tailcall`, `Linearize`, `Allocation`, and `RTLgen`, it takes less than one person day per pass.

By contrast, implementing our framework is more challenging, which took us about 1 person year. In particular, proving the equivalence between non-preemptive and preemptive semantics for DRF programs took us 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' executions.

## 8 Related Work and Conclusion

***Compiler verification.*** Various work extends CompCert [16] to support separate compilation or concurrency. We have discussed Compositional CompCert [2, 29] in Sec. 1 and 2. SepCompCert [15] 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.

CompCertTSO [27] compiles ClightTSO programs to the x86-TSO machine. It does not support cross-language linking, and its proof for the two CompCert passes `Stacking` and `Cminorgen` are not compositional. By contrast, we have verified these two passes using our compositional simulation. For the other compositional passes, CompCertTSO relies on a thread-local simulation, which is stronger than ours. It requires that the source and the target always generate the same memory events (excepts for those local variables that can be stored in registers). As a result, some optimizations (such as constant propagation and CSE) in CompCertTSO have to be more restrictive.

As an extension of CompCertTSO, Jagannathan et al. [12] allow the compiler to inject racy code such as the efficient spin lock in Fig. 10. They propose a refinement calculus on the racy code to ensure the compilation correctness. Their work looks similar to our extended framework in Fig. 3, but since they use TSO semantics for both the source and target programs, they do not need to handle the gap between the SC and TSO semantics, so they do not need the source to be DRF as in our work.

Podkopaev et al. [24] prove correctness of the compilation from the promising semantics (which is a high-level operational relaxed model) to the operational ARMv8-POP machine. They develop whole-program simulations to deal with the complicated relaxed behaviors. Later on they verify compilations from the promising semantics to declarative hardware models such as POWER, ARMv7 and ARMv8 [25].

The simulations are tied to the promising semantics. They do not aim for reusing existing sequential compilations but handle a lot of complicated issues in relaxed models.

As part of their CCAL framework, Gu et al. [10] develop thread-safe CompCertX (TSCompCertX), which supports separate compilation of concurrent C programs, and the linking of C programs with assembly modules. However, the compositionality of TSCompCertX is tied to the specific settings of the CCAL model. In particular, it relies on the abstraction of concurrent objects to derive the partition of private and shared memory, and uses the auxiliary push and pull instructions to ensure race freedom. Also TSCompCertX supports only race-free programs in the sequentially consistent memory model. Our work does not need source-level specifications for race-free programs. Besides, we support confined benign races in x86-TSO (a feature not supported in TSCompCertX), where the racy objects are required to have race-free abstraction.

Vellvm [35, 36] proves correctness of several optimization passes for *sequential* LLVM programs. Wang et al. [32] verify a separate compiler from Cito to Bedrock, which relies on axiomatic specifications for cross-language external calls. It is unclear how to adapt their work to concurrency. Perconti and Ahmed [23] verify separate compilation by embedding languages in a combined language. They do not support concurrency either. Ševčík [26] 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 [20] verifies a compiler for concurrent Java programs. His simulation has similar restrictions as CompCertTSO.

***Non-preemptive semantics and data-race-freedom.*** Non-preemptive (or cooperative) semantics has been developed in various settings for various purposes (e.g., [1, 5, 18, 31, 34]). Both Ferreira et al. [9] and Xiao et al.[33] 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 [21] proposes a concept called Region-Conflict-Freedom, which looks similar to our NPDRF, but there is no formal operational formulation as we do. Owens [22] proposes Triangular-Race-Freedom (TRF) and proves that TRF programs behaves the same in `x86-SC` and `x86-TSO`. TRF is weaker than DRF and can be satisfied by the efficient spin lock code in Fig. 10.

***Conclusion and future work.*** We present a framework for building certified compilation of concurrent programs from sequential compilation. We develop CASCompCert, which reuses CompCert to compile DRF programs, with the support of confined benign races in manually written assembly (`x86-TSO`). We believe our work is a promising start for certified separate compilation of *general* concurrent programs. The latter goal requires longer-term work and has more problems to address, as discussed below.

First, our work is limited in the support of general concurrent languages. For instance, we have not yet considered thread spawn, though we do not see any particular challenges. The spawn step in the operational semantics needs to assign a new $F$ to each newly created thread. In simulations spawns should be handled in a similar way as context switches. Besides, our extended framework currently does not support multiple objects because it lacks a mechanism to ensure the partition between objects' data. To address the problem, we may follow the ideas in LRG [8] and CAP [7] to set the logical boundaries between objects. Also it is worthwhile to support relaxed concurrency, such as C11-style memory models which have relaxed atomics.

Second, it is also interesting to explore other target machine models, such as PowerPC and ARM which have LL/SC instructions rather than lock-prefixed atomic instructions on x86.

Third, we would like to verify more optimization passes, including those relying on concurrency features. For instance, our work may be modified to support roach-motel reorderings, by distinguishing `EntAtom` and `ExtAtom` in the local simulation and recording the footprints that are moved across `EntAtom` or `ExtAtom`. We also would like to finish the proofs for the remaining CompCert optimization passes (which we do not expect any major technical challenges) and add the support of stack pointer escape following the on-paper proofs in our TR [13].

Finally, we are also curious about extensional (language-independent) characterizations of footprints. Our formulation of wd (Def. 1) is one way to characterize footprints but it is not restrictive enough to rule out all benign races. This in practice is harmless since the current wd already allows us to prove our final theorem of the framework (Thm. 12), and we use properly defined concrete languages to prove correctness of specific compilers (Thm. 14). Nevertheless, it is interesting to explore better formulations of wd.

## Acknowledgments

We thank our shepherd John Reppy, Zhong Shao and anonymous referees for their suggestions and comments on earlier versions of this paper. We especially thank Weijie Fan and Jianwei Song for their helps on the Coq proofs of the optimization passes `Tunneling` and `CleanupLabels`. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant No. 61632005, and from Huawei Innovation Research Program (HIRP).

## References

[1] Martin Abadi and Gordon Plotkin. 2009. A Model of Cooperative Threads. In *POPL*. 29–40.

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

[3] Hans-Juergen Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *HotPar*. 3–3.

[4] Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. 68–78.

[5] Gérard Boudol. 2007. Fair Cooperative Multithreading. In *CONCUR*. 272–286.

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

[7] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*. 504–528.

[8] Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL*. 315–327.

[9] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized Memory Models and Concurrent Separation Logic. In *ESOP*. 267–286.

[10] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *PLDI*. 646–661.

[11] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[12] Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, and Jan Vitek. 2014. Atomicity Refinement for Verified Compilation. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 6:1–6:30.

[13] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs (Technical Report and Coq Implementations). https://plax-lab.github.io/publications/ccc/

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

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

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

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

[18] 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*. 189–199.

[19] Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *PLDI*. 459–470.

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

[21] 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*. 351–362.

[22] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP*. 478–503.

[23] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP*. 128–148.

[24] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *ECOOP*. 22:1–22:28.

[25] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *PACMPL* 3, POPL (2019), 69:1–69:31.

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

[27] 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.

[28] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

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

[30] R. K. Treiber. 1986. *System programming: coping with parallelism*. Technical Report RJ 5118. IBM Almaden Research Center.

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

[32] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-language Linking via Data Abstraction. In *OOPSLA*. 675–690.

[33] Siyang Xiao, Hanru Jiang, Hongjin Liang, and Xinyu Feng. 2018. Non-Preemptive Semantics for Data-Race-Free Programs. In *ICTAC*. 513–531.

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

[35] 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.

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