Towards Certified Compositional Compilation for Concurrent Programs

Hanru Jiang Hongjin Liang Xinyu Feng
University of Science and Technology of China
hanru219@mail.ustc.edu.cn lhj1018@ustc.edu.cn xyfeng@ustc.edu.cn

Abstract

Certified compositional compilation is important for establishing end-to-end guarantees for certified systems consisting of separately compiled modules. In this paper, we propose a framework consisting of the key semantics components and verification steps that bridge the gap between the compilers for sequential programs and for (race-free) concurrent ones, so that the existing efforts on certified sequential compilation can be reused for concurrent programs. Contributions of the framework include an abstract formulation of race-freedom in an interaction semantics, and a footprint-preserving compositional simulation as the compilation correctness criterion.

1. Introduction

Address the problem, its importance and challenges. Certified compositional compilation has been considered an important (though challenging) problem **[[[by whom?]]]**. A real-world program usually consists of multiple modules which are compiled independently. Correct compilation is expected to have compositionality, ensuring that separately compiled modules can work together and preserve the semantics of the source program as a whole. That is, we need not only that each single target module preserves the semantics of its source, but also that the *interactions* between the source modules are correctly reflected at the target. With the presence of concurrency, the problem becomes much harder since we have to consider non-deterministic interleavings between threads as well.

Advance over previous work. Although there are already many works on certified separate compilation, most of them deal with compilation for sequential programs. The first realistic certified compiler, CompCert [1], establishes the semantic preservation property for *closed sequential* programs. It can guarantee the correctness of separate compilation when the sequential modules do not interact with each other. To support general separate compilation, Stewart et al. [4] developed Compositional CompCert, which allows the modules to call each other's external functions. Their approach relies on *non-preemptive* semantics, where each module is interacted with others only at certain program points specified in the module code (i.e., at external calls only). It is unclear if their approach can also be applied to interleaving concurrency, where a module may be preempted by others at arbitrary and non-deterministically-chosen program points.

On the other hand, existing works on certified compilation for concurrent programs are usually not compositional (e.g., [2, 3], which we will discuss in Sec. 9), thus cannot be applied to separate compilation. As we will see in detail in Sec. 2, it is non-trivial to build certified compositional compilation for preemptive concurrent programs.

In this paper, we propose a framework consisting of the key semantics components and verification steps that bridge the gap between compilation for sequential programs and for (race-free) concurrent ones. The key ingredient here is data-race-free (DRF) assumption. We first formally prove the folklore theorem about the equivalence between the preemptive and non-preemptive semantics for race-free programs, so that it is possible to derive the semantics preservation between preemptive programs from the semantics preservation between non-preemptive ones when both source and target programs are race-free. Then we prove DRF preservation in a compositional style, using a notion of footprint-preservation, thus DRF of target program could be derived from DRF of the source. Finally, together with the semantics preservation proof between non-preemptive programs, we are able to derive the semantics preservation result between preemptive programs, assuming the source is race free.

Contribution. Our work is built upon earlier work on CompCert [1] and Compositional CompCert [4], but makes the following new contributions:

- We design a footprint-preserving simulation as the correctness of compilation for individual modules. As an extension of the structured simulation in Compositional CompCert, our simulation considers interactions at both external function calls and synchronization points, thus is compositional with respect to both module linking and non-preemptive parallelism. It also requires that the footprints of the steps made by the source and target modules be related, which is the key to derive DRF preservation between the source and target whole programs in a compositional style.
- To bridge the gap between our simulations in the non-preemptive semantics and the DRF preservation we need for preemptive programs, we formulate a novel notion of DRF in the non-preemptive semantics, denoted as NPDRF. We show that a program satisfies our NPDRF in the non-preemptive semantics if and only if it satisfies the standard DRF notion in the interleaving semantics. Then DRF preservation can be derived from NPDRF preservation, which is ensured by footprint preservation in our thread-local and module-local simulations.
- Putting all these together, our framework (see Fig. 3) is the first to build certified compositional compilation for concurrent programs from sequential compilation. It highlights the importance of DRF preservation for correct compilation. It also gives the essential semantic requirments for the proof of equivalence between preemptive and non-preemptive semantics.
- As a instantiation of our language independent compilation framework, we extend the CompCert Clight with a set of C11-like SC-atomic primitives as source language, and x86 assembly with `lock` prefix as target language. We successfully

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

(a) CompCert

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

(b) Compositional CompCert

Figure 1. Proof structures of certified compilation.

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

(a) $S \precsim C$

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

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

Figure 2. Simulation diagrams.

proved the CompCert compiler and the atomic primitives with their implementations satisfying our correctness criteria. Thus Clight modules compiled by CompCert linked with the atomic primitives are proved to preserve the behavior of the (DRF) source program as a whole.

[[[Is it possible to bypass Compositional CompCert and reuse CompCert proofs directly without major changes? Are we able to claim it is a "lightweighted" approach compared with Compositional CompCert / CompCertTSO?]]]

- In this instantiation, the only assumptions we made on source programs is DRF. We provide various ways for proving DRF property including a static race checker and a simplified separation logic that are sound with respect to data-race freedom. With an automatic race checker, the entire proof of semantics preservation could be generated with simply one click.

In the rest of this paper, we first analyze the challenge and give an overview of our approach in Sec. 2. We give the basic technical setting in Sec. 3, including the footprint-instrumented preemptive semantics and the refinement definition. Sec. 4 presents the non-preemptive semantics, which is the basis for both our new simulation (Sec. 14) and the NPDRF definition (Sec. 7). We show the final theorem in Sec. 8, and discuss related work in Sec. 9.

2. Informal Development

$$\begin{tikzcd}
S_1 \parallel \ldots \parallel S_n & \approx & C_1 \parallel \ldots \parallel C_n & \\
\textcircled{1}\ \wr & & \wr & \mathsf{DRF}(C_1 \parallel \ldots \parallel C_n) \arrow[l, Rightarrow, "\textcircled{2}"'] \\
S_1 \mid \ldots \mid S_n & \approx & C_1 \mid \ldots \mid C_n & \mathsf{NPDRF}(C_1 \mid \ldots \mid C_n) \arrow[u, Rightarrow, "\textcircled{6}"'] \\
S_1 \mid \ldots \mid S_n & \succcurlyeq \arrow[u, Rightarrow, "\textcircled{3}"] & C_1 \mid \ldots \mid C_n \arrow[r, Rightarrow, "\textcircled{7}"] & {} \\
S_1 \mid \ldots \mid S_n & \preccurlyeq \arrow[u, Rightarrow, "\textcircled{4}", "\forall i.\ \mathsf{Det}(C_i)"'] & C_1 \mid \ldots \mid C_n & \mathsf{NPDRF}(S_1 \mid \ldots \mid S_n) \arrow[uu, Rightarrow] \\
& \forall i.\ \mathsf{R},\mathsf{G} \vdash S_i \preccurlyeq C_i \arrow[u, Rightarrow, "\textcircled{5}"] & & \mathsf{DRF}(S_1 \parallel \ldots \parallel S_n) \arrow[u, Rightarrow, "\textcircled{8}"']
\end{tikzcd}$$

Figure 3. Our framework

3. Basic Technical Settings

3.1 The Abstract Language

$$
\begin{array}{rccl}
(\mathit{Prog}) & P, \mathbb{P} & ::= & \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \\
(\mathit{Entry}) & \mathtt{f} & \in & \mathit{String} \qquad (\mathit{TID})\ \mathtt{t} \in \mathbb{N} \\
(\mathit{MdSet}) & \Pi, \Gamma & ::= & \{(\mathit{tl}_1, \pi_1), \ldots, (\mathit{tl}_m, \pi_m)\} \\
(\mathit{Lang}) & \mathit{tl}, \mathit{sl} & ::= & (\mathit{Module}, \mathit{Core}, \mathsf{InitCore}, \longmapsto, \mathsf{AftExtn}) \\
(\mathit{Module}) & \pi, \gamma & ::= & \ldots \\
(\mathit{Core}) & \kappa, \Bbbk & ::= & \ldots \\
 & \mathsf{InitCore} & \in & \mathit{Module} \to \mathit{Entry} \rightharpoonup \mathit{list}(\mathit{Val}) \rightharpoonup \mathit{Core} \\
 & \longmapsto & \in & (\mathit{Module} \times \mathit{FList}) \times (\mathit{Core} \times \mathit{State}) \to \\
 & & & \{\mathcal{P}((\mathit{Msg} \times \mathit{FtPrt}) \times (\mathit{Core} \times \mathit{State})), \textbf{abort}\} \\
 & \mathsf{AftExtn} & \in & \mathit{Core} \times \mathit{Value} \rightharpoonup \mathit{Core} \\
(\mathit{BlockID}) & b & \in & \mathbb{N} \\
(\mathit{State}) & \sigma, \Sigma & \in & \mathit{BlockID} \rightharpoonup_{\mathrm{fin}} \mathbb{N} \rightharpoonup \mathit{Value} \\
(\mathfrak{L}) & l & ::= & (b, i) \qquad \text{where } i \in \mathbb{N} \\
(\mathit{Value}) & v & ::= & l \mid \ldots \\
(\mathit{BSet}) & S, \mathbb{S} & \in & \mathcal{P}(\mathit{BlockID}) \\
(\mathit{FList}) & F, \mathbb{F} & \in & \mathcal{P}^{\omega}(\mathit{BlockID}) \\
(\mathit{FSpace}) & \mathit{FS}, \mathbb{FS} & ::= & F :: \mathit{FS} \quad \text{(co-inductive)} \\
(\mathit{FtPrt}) & \delta, \Delta & ::= & (\mathit{rs}, \mathit{ws}) \qquad \text{where } \mathit{rs}, \mathit{ws} \in \mathcal{P}(\mathfrak{L}) \\
(\mathit{Msg}) & \iota & ::= & \tau \mid e \mid \mathsf{call}(\mathtt{f}, \vec{v}) \mid \mathsf{ret}(v) \\
 & & \mid & \mathsf{entA} \mid \mathsf{extA} \\
(\mathit{Event}) & e & ::= & \ldots \\
(\mathit{Config}) & \phi, \Phi & ::= & (\kappa, \sigma) \mid \textbf{abort}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{locs}(\sigma) \overset{\text{def}}{=} \{(b, i) \mid b \in \mathsf{dom}(\sigma) \land i \in \mathsf{dom}(\sigma(b))\} \\
\llbracket S \rrbracket \overset{\text{def}}{=} S \times \mathbb{N} \\
\mathsf{forward}(\sigma, \sigma') \ \text{iff} \ (\mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma')) \land \\
\qquad\qquad (\mathsf{locs}(\sigma') \cap \llbracket \mathsf{dom}(\sigma) \rrbracket \subseteq \mathsf{locs}(\sigma)) \\
\sigma \overset{\mathit{rs}}{=\!=} \sigma' \ \text{iff} \ \forall (b, i) \in \mathit{rs}.\ ((b, i) \notin \mathsf{locs}(\sigma) \cup \mathsf{locs}(\sigma')) \lor \\
\qquad\qquad ((b, i) \in \mathsf{locs}(\sigma) \cap \mathsf{locs}(\sigma')) \land (\sigma(b)(i) = \sigma'(b)(i)) \\
\delta \subseteq \delta' \ \text{iff} \ (\delta.\mathit{rs} \subseteq \delta'.\mathit{rs}) \land (\delta.\mathit{ws} \subseteq \delta'.\mathit{ws}) \\
\delta \cup \delta' \overset{\text{def}}{=} (\delta.\mathit{rs} \cup \delta'.\mathit{rs}, \delta.\mathit{ws} \cup \delta'.\mathit{ws})
\end{array}
$$

Figure 4. The Abstract Concurrent Language

Fig. 4 show the syntax of an abstract language for preemptive concurrent programming, which can be instantiated to various concrete and practical languages.

A program P consists of several threads running in parallel. Each thread executes a module declared in Π, starting from an entry $\mathtt{f}$. Each module π may contain several entries. Different modules may be written in different languages. We define a module language tl as a tuple $(\mathit{Module}, \mathit{Core}, \mathsf{InitCore}, \longmapsto)$. *Module* describes the syntax of the modules in this language. Following Compositional CompCert [4], *Core* is the set of internal "core" states, which can be instantiated to control continuations, instruction streams, register files, etc.. Given a module π and an entry $\mathtt{f}$, the function $\mathsf{InitCore}$ returns the initial "core" state κ (it is undefined if the entry is not contained in the module). We assume that the entries in different modules are different.

A memory state σ is defined as a partial function of the type $\mathfrak{L} \rightharpoonup \mathit{Value}$, where $\mathfrak{L}$ represents the type of locations. To simplify the presentation, we specialize $\mathfrak{L}$ to be the set of memory locations in the CompCert memory model [**?**], where memory is organized as a collection of blocks. A memory location l is a pair (b, i) of a block b and an offset i within this block. The purpose of this memory model is to allow pointer arithmetic only within the same block. Note that our framework actually does not rely on this special instantiation of $\mathfrak{L}$. It can be applied to other memory models.

Besides, we assume that every thread is created with a free list F (where $F \subseteq \mathfrak{L}$), used to allocate memory for the thread. The free lists of different threads are disjoint, ensuring that the allocation operations made by different threads never race. To simplify the presentation, we assume that free lists are infinite. In the case of CompCert memory model, F should contain an infinite number of blocks where each block contains all the possible offsets. Here $\mathcal{P}(\cdot)$ denotes the power set. **[[[No need to assume the special form of free lists? Besides, where do we *use* the assumption that F is infinite?]]]**

Footprint-based semantics. The operational semantics of a module is defined using the labeled transitions $(\pi, F) \vdash (\kappa, \sigma) \underset{\delta}{\overset{\iota}{\longmapsto}} (\kappa', \sigma')$ (or $(\pi, F) \vdash (\kappa, \sigma) \underset{\delta}{\overset{\iota}{\longmapsto}}$ **abort** if the step goes wrong). We write ϕ for the configuration (κ, σ) or **abort**. Each step is labeled with a message ι and a footprint δ. The messages contain information about the module-local steps. To simplify the presentation, we only consider externally observable events e (such as outputs), termination $\mathtt{halt}$, and the starts and ends of atomic blocks entA and extA. Any other step is silent, labeled with τ (which is often omitted in the following presentation). The messages define the protocols of communications with the global (whole-program) semantics (which will be described soon), so it is natural to require all the module languages to use the same message formats. They allow us to abstract away the syntax and semantics details of the module languages, and focus on the interactions with other modules and the external observer (the latter can observe e only). We can also extend ι with messages of external function calls, and extend the global semantics to handle them. But supporting external functions is orthogonal to our work on certified compilation of concurrent programs, so we do not present it here.

The footprint δ is defined as a pair $(\mathit{rs}, \mathit{ws})$, which records the memory locations that are read and written in this step. Recording the footprint allows us to discuss races between threads in the following sections. We write $\mathtt{emp}$ for the special footprint where both the read and write sets are empty.

We require the language semantics to be well-defined (see wd in Def. 1 below). First, a step may enlarge the memory domain but cannot reduce it, and the additional memory should be allocated from F. This requirement follows the CompCert memory model where memory disposals do not really remove the locations from the memory (they just become invalid). Second, the footprints should be included in the domain of the memory, and the newly allocated memory should be included in the write sets. Besides, the memory out of the write sets should keep unchanged, as described by $\sigma \overset{\mathsf{dom}(\sigma) - \delta.\mathit{ws}}{=\!=\!=\!=\!=} \sigma'$ (which is defined at the bottom of Fig. 4). Finally, the memory updates at the write sets of a step only depends on the read sets.

Definition 1 (Well-Defined Languages)**.** $\mathsf{wd}(\mathit{tl})$ iff for any π, F, κ, σ, ι, κ', σ' and δ, if $(\pi, F) \vdash (\kappa, \sigma) \underset{\delta}{\overset{\iota}{\longmapsto}} (\kappa', \sigma')$, then all of the following hold (some auxiliary definitions are in Fig. 5):

(1) $\mathsf{forward}(\sigma, \sigma')$;
(2) $\mathtt{LEffect}(\sigma, \sigma', \delta, F)$
(3) for any σ_1, $\mathtt{LEqPre}(\sigma, \sigma_1, \delta, F)$, then there exists σ_1' such that $(\pi, F) \vdash (\kappa, \sigma_1) \underset{\delta}{\overset{\iota}{\longmapsto}} (\kappa', \sigma_1')$ and $\mathtt{LEqPost}(\sigma', \sigma_1', \delta, F)$.
(4) for any δ_0 and σ_1, if

$$\delta_0 = \bigcup \{\delta'' \mid \exists \kappa'', \sigma''.\ (\pi, F) \vdash (\kappa, \sigma) \underset{\delta''}{\overset{\tau}{\longmapsto}} (\kappa'', \sigma'')\}\ ,$$

$$\begin{array}{lcl}
\mathtt{LEqPre}(\sigma_1,\sigma_2,\delta,F) & \overset{\text{def}}{=} & \sigma_1 \overset{\delta.rs}{=\!=} \sigma_2 \wedge \\
& & \mathsf{locs}(\sigma_1)\cap\delta.ws = \mathsf{locs}(\sigma_2)\cap\delta.ws \wedge \\
& & \mathsf{dom}(\sigma_1)\cap F = \mathsf{dom}(\sigma_2)\cap F \\[1em]
\mathtt{LEqPost}(\sigma_1',\sigma_2',\delta,F) & \overset{\text{def}}{=} & \sigma_1' \overset{\delta.ws}{=\!=} \sigma_2' \wedge \\
& & \mathsf{dom}(\sigma_1')\cap F = \mathsf{dom}(\sigma_2')\cap F \\[1em]
\mathtt{LEffect}(\sigma_1,\sigma_2,\delta,F) & \overset{\text{def}}{=} & \sigma_1 \overset{\mathsf{locs}(\sigma_1)-\delta.ws}{=\!=\!=\!=} \sigma_2 \wedge \\
& & \mathsf{locs}(\sigma_1)-\mathsf{locs}(\sigma_2) \subseteq \delta.ws \wedge \\
& & \mathsf{locs}(\sigma_2)-\mathsf{locs}(\sigma_1) \subseteq \delta.ws \wedge \\
& & \mathsf{dom}(\sigma_2)-\mathsf{dom}(\sigma_1) \subseteq F
\end{array}$$

Figure 5. Auxiliary definitions for well-defined languages

$\mathtt{LEqPre}(\sigma,\sigma_1,\delta_0,F)$ then

$$\forall \kappa_1'',\sigma_1'',\iota_1'',\delta_1''.\,(\pi,F)\vdash(\kappa,\sigma_1)\xrightarrow[\delta_1'']{\iota_1''}(\kappa_1'',\sigma_1'') \implies$$

$$\exists\sigma''.\,(\pi,F)\vdash(\kappa,\sigma)\xrightarrow[\delta_1'']{\iota_1''}(\kappa_1'',\sigma'')$$

(5) if $\iota = \mathsf{call}(\mathtt{f},\vec{v})$, then $\forall v'.\,\exists\kappa''.\,\mathsf{AftExtn}(\kappa',v')=\kappa''$;
if $\iota = \mathsf{entA} \vee \iota = \mathsf{extA}$, then $\forall v'.\,\mathsf{AftExtn}(\kappa',v')=\kappa'$.

Definition 2 (Deterministic Languages)**.** $\mathsf{det}(tl)$ iff

$$\begin{array}{l}
\forall \pi,F,\phi,\phi_1,\phi_2,\iota_1,\iota_2,\delta_1,\delta_2.\\
(\pi,F)\vdash\phi\xrightarrow[\delta_1]{\iota_1}\phi_1 \wedge (\pi,F)\vdash\phi\xrightarrow[\delta_2]{\iota_2}\phi_2 \implies\\
\phi_1=\phi_2\wedge\iota_1=\iota_2\wedge\delta_1=\delta_2
\end{array}$$

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

Other assumptions. In addition to the above requirements on module languages, we also assume that any non-silent transition (i.e., labeled with e, entA, extA or `halt`) does not access or update the memory state, so its footprint is `emp`. **[[[The assumption that e-step's footprint is empty looks strong...]]]** Besides, the steps producing entA and extA should be paired. Also, we do not allow nested atomic blocks or outputs inside atomic blocks (i.e., the steps producing e or entA never happen between an entA step and an extA step).

Note that we do not enforce these assumptions, but our global semantics (see below) would *abort* if no transitions of the module can satisfy these assumptions. **[[[Can it just "get stuck"?]]]**

Global semantics. Fig. 7 defines a set of global semantics rules to manipulate the preemption among threads. As shown in Fig. 6(a), the global world W consists of the thread pool T, the ID $\mathtt{t}$ of the thread currently being executed, a bit d indicating whether the current thread is in an atomic block or not, and the memory state σ. The thread pool T contains the runtime module K of every thread, which records the module language tl, the module code π, the free list F created for the thread, and the current "core" state κ. Since we assume that there is no external function calls, each thread would execute exactly one module. In the following presentation, we may not distinguish threads and modules.

$$\begin{array}{rrcl}
(\mathit{World}) & W,\mathbb{W} & ::= & ((\Pi,T),(\mathtt{t},d,\sigma,\mathit{FS}))\\
(\mathit{ThrdPool}) & T,\mathbb{T} & ::= & \{\mathtt{t}_1\leadsto K_1,\ldots,\mathtt{t}_n\leadsto K_n\}\\
(\mathit{RtMdStk}) & K,\mathbb{K} & ::= & \epsilon \mid ((tl,\pi,F),\kappa)::K\\
(\mathit{AtomBit}) & d & ::= & 0 \mid 1\\
(\mathit{GMsg}) & o & ::= & \tau \mid e \mid \mathsf{sw}
\end{array}$$

(a) The Preemptive Model

$$\begin{array}{l}
\mathsf{initRtMd}(\Pi,\mathtt{f},\vec{v},F) \overset{\text{def}}{=} ((tl,\pi,F),\ \kappa),\\
\qquad\qquad \text{if } (tl,\pi)\in\Pi \text{ and } tl.\mathsf{InitCore}(\pi)(\mathtt{f})(\vec{v})=\kappa\\
\mathsf{initFList}(\sigma,\mathit{FS}) \ \text{ iff } \ (\mathsf{dom}(\sigma)\perp \mathit{FS})\wedge\mathsf{disjoint}(\mathit{FS})
\end{array}$$

$$\frac{S\cap F=\emptyset \qquad S\perp \mathit{FS}}{S\perp(F::\mathit{FS})} \qquad \frac{F\perp\mathit{FS}\qquad \mathsf{disjoint}(\mathit{FS})}{\mathsf{disjoint}(F::\mathit{FS})}$$

(b) Initialization

Figure 6. The Runtime Global Models

Conventions. We usually write blackboard bold or capital letters (e.g., $\mathbb{P}$, $\mathbb{W}$, $\Bbbk$ and Σ) for the notations at the source level to distinguish from the target-level ones (e.g., P, W, κ and σ). The set of modules at source is written as Γ, to distinguish from the target Π. Similarly, γ is a source module while π is a target one.

$$\frac{\begin{array}{c} T = \{1 \rightsquigarrow K_1, \ldots, n \rightsquigarrow K_n\} \qquad \mathsf{initFList}(\sigma, (F_1 :: \ldots :: F_n :: \mathit{FS})) \\ \forall i \in \{1, \ldots, n\}.\, K_i = \mathsf{initRtMd}(\Pi, \mathtt{f}_i, \epsilon, F_i) :: \epsilon \qquad \mathtt{t} \in \{1, \ldots, n\} \end{array}}{(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \sigma) \xRightarrow{\mathit{load}} ((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS}))}\ \text{Load}$$

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

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau} (\kappa', \sigma') \\ T' = T\{\mathtt{t} \rightsquigarrow ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, d, \sigma, \mathit{FS})) \xRightarrow[\delta]{\tau} ((\Pi, T'), (\mathtt{t}, d, \sigma', \mathit{FS}))}\ \tau\text{-step}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{e} (\kappa', \sigma) \\ T' = T\{\mathtt{t} \rightsquigarrow ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{e} ((\Pi, T'), (\mathtt{t}, 0, \sigma, \mathit{FS}))}\ \text{Print}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{call}(\mathtt{f}, \vec{v})} (\kappa', \sigma) \\ \mathit{FS} = F_1 :: \mathit{FS}' \qquad \mathsf{initRtMd}(\Pi, \mathtt{f}, \vec{v}, F_1) = ((\mathit{tl}_1, \pi_1, F_1), \kappa_1) \qquad T' = \{\mathtt{t} \rightsquigarrow ((\mathit{tl}_1, \pi_1, F_1), \kappa_1) :: ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\tau} ((\Pi, T'), (\mathtt{t}, 0, \sigma, \mathit{FS}'))}\ \text{EFCall}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}_1, \pi_1, F_1), \kappa_1) :: ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi_1, F_1) \vdash (\kappa_1, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{ret}(v)} (\kappa_1', \sigma) \\ \mathsf{AftExtn}(\kappa, v) = \kappa' \qquad T' = T\{\mathtt{t} \rightsquigarrow ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\tau} ((\Pi, T'), (\mathtt{t}, 0, \sigma, \mathit{FS}))}\ \text{EFRet}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{entA}} (\kappa', \sigma) \\ T' = T\{\mathtt{t} \rightsquigarrow ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\tau} ((\Pi, T'), (\mathtt{t}, 1, \sigma, \mathit{FS}))}\ \text{EntAt}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{extA}} (\kappa', \sigma) \\ T' = T\{\mathtt{t} \rightsquigarrow ((\mathit{tl}, \pi, F), \kappa') :: K\} \end{array}}{((\Pi, T), (\mathtt{t}, 1, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\tau} ((\Pi, T'), (\mathtt{t}, 0, \sigma, \mathit{FS}))}\ \text{ExtAt}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: \epsilon \\ (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{ret}(v)} (\kappa', \sigma) \qquad \mathtt{t}' \in \mathsf{dom}(T \backslash \mathtt{t}) \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\mathsf{sw}} ((\Pi, T \backslash \mathtt{t}), (\mathtt{t}', 0, \sigma, \mathit{FS}))}\ \text{Term}$$

$$\frac{\begin{array}{c} T(\mathtt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: \epsilon \\ (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\mathtt{emp}]{\mathsf{ret}(v)} (\kappa', \sigma) \qquad \mathsf{dom}(T) = \{\mathtt{t}\} \end{array}}{((\Pi, T), (\mathtt{t}, 0, \sigma, \mathit{FS})) \xRightarrow[\mathtt{emp}]{\tau} \mathbf{done}}\ \text{Done}$$

Figure 7. The Preemptive Global Semantics

3.2 Event-Trace Refinement and Equivalence

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

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

$$\dfrac{W \Rightarrow^{+} \textbf{done}}{\mathit{Etr}(W,\textbf{done})} \qquad \dfrac{W \Rightarrow^{+} W' \quad \mathit{Etr}(W',\epsilon)}{\mathit{Etr}(W,\epsilon)}$$

Figure 8. Definition of $\mathit{ProgEtr}((P,\sigma),\mathcal{B})$.

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

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

Definition 3 (Event-Trace Refinement and Equivalence)**.**
$(\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)$ iff
$\quad \forall\mathcal{B}.\ \mathit{ProgEtr}((P,\sigma),\mathcal{B}) \implies \mathit{ProgEtr}((\mathbb{P},\Sigma),\mathcal{B})$.
$(\mathbb{P},\Sigma) \approx (P,\sigma)$ iff
$\quad \forall\mathcal{B}.\ \mathit{ProgEtr}((P,\sigma),\mathcal{B}) \iff \mathit{ProgEtr}((\mathbb{P},\Sigma),\mathcal{B})$.
$\mathbb{P} \sqsupseteq_{ge,\varphi} P$ iff

$$\begin{array}{l}\forall\Sigma,\sigma.\ (ge \subseteq \mathsf{locs}(\Sigma)) \wedge (\mathsf{cl}(\mathsf{dom}(\Sigma),\Sigma) = \mathsf{dom}(\Sigma)) \wedge \\ \quad (\varphi\{\!\{\mathsf{dom}(\Sigma)\}\!\} = \mathsf{dom}(\sigma)) \wedge (\varphi\langle\!\langle\mathsf{locs}(\Sigma)\rangle\!\rangle = \mathsf{locs}(\sigma)) \wedge \\ \quad \mathsf{Inv}(\varphi,\Sigma,\sigma) \implies (\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)\end{array}$$

4. The Non-Preemptive Semantics

A key step in our framework is to relate the source and target preemptive programs to the corresponding non-preemptive programs. In this section we define the global semantics of the non-preemptive programs, where a thread interacts with other threads at only synchronization points (i.e., when it enters and exits atomic blocks, and outputs). The non-preemptive semantics is the basis for both our new simulation (see Sec. 14) and our NPDRF definition (see Sec. 7).

To distinguish from the preemptive parallelism, we write **let** Π **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ for the non-preemptive programs, denoted by $\hat{P}$. As shown in Fig. 6(b), the non-preemptive global world $\widehat{W}$ is defined similarly to the preemptive world W, except that $\widehat{W}$ records the atomic bits of all the threads (denoted by $\mathbb{d}$). Unlike the preemptive semantics in Fig. 7(a) using the atomic bit d to know whether or not it is allowed to switch, here we need $\mathbb{d}$ to define NPDRF later in Sec. 7.

The non-preemptive global steps are formulated as $\widehat{W} :\overset{o}{\underset{\delta}{\Rightarrow}} \widehat{W}'$ (and $\widehat{W} :\overset{\tau}{\underset{\mathtt{emp}}{\Longrightarrow}}$ **abort** for the aborting steps, and $\widehat{W} :\overset{\tau}{\underset{\mathtt{emp}}{\Longrightarrow}}$ **done** for the terminating steps), defined in Fig. 7(b). The rules (Print_{np}), (EntAt_{np}) and (ExtAt_{np}) execute one step of the current thread $\mathtt{t}$, and then non-deterministically switch to a thread $\mathtt{t}'$ (which could just be $\mathtt{t}$). The corresponding global steps produce the sw events (or the external event e in the (Print_{np}) rule) which will be used to define NPDRF later. Other rules are very similar to their counterparts in the preemptive semantics in Fig. 7(a).

$$\begin{array}{llll}(\mathit{NPProg}) & \hat{P} & ::= & \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \\ (\mathit{NPWorld}) & \widehat{W}, \widehat{\mathbb{W}} & ::= & ((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS})) \\ (\mathit{AtomBitSet}) & \mathbb{d} & ::= & \{\mathtt{t}_1 \leadsto d_1,\ldots,\mathtt{t}_n \leadsto d_n\}\end{array}$$

Figure 9. The Non-preemptive Runtime Global Models

$$\frac{\begin{array}{c}\mathsf{initFList}(\sigma,(F_1::\ldots::F_n::\mathit{FS})) \qquad \forall i\in\{1,\ldots,n\}.\,K_i=\mathsf{initRtMd}(\Pi,\mathtt{f}_i,\epsilon,F_i)::\epsilon \\ T=\{\mathtt{t}_1\leadsto K_1,\ldots,\mathtt{t}_n\leadsto K_n\} \qquad \mathtt{t}\in\{\mathtt{t}_1,\ldots,\mathtt{t}_n\} \qquad \mathbb{d}=\{\mathtt{t}_1\leadsto 0,\ldots,\mathtt{t}_n\leadsto 0\}\end{array}}{(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1\mid\ldots\mid\mathtt{f}_n,\sigma)\ :\xRightarrow{\mathit{load}}((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))}\ \text{Load}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\delta]{\tau}(\kappa',\sigma') \\ T'=T\{\mathtt{t}\leadsto((\mathit{tl},\pi,F),\kappa')::K\}\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\delta]{\tau}((\Pi,T'),(\mathtt{t},\mathbb{d},\sigma',\mathit{FS}))}\ \tau\text{-step}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{e}(\kappa',\sigma) \\ \mathbb{d}(\mathtt{t})=0 \qquad T'=T\{\mathtt{t}\leadsto((\mathit{tl},\pi,F),\kappa')::K\} \qquad \mathtt{t}'\in\mathsf{dom}(T)\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{e}((\Pi,T'),(\mathtt{t}',\mathbb{d},\sigma,\mathit{FS}))}\ \text{Print}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{call}(\mathtt{f},\vec{v})}(\kappa',\sigma) \qquad \mathbb{d}(\mathtt{t})=0 \\ \mathit{FS}=F_1::\mathit{FS}' \qquad \mathsf{initRtMd}(\Pi,\mathtt{f},\vec{v},F_1)=((\mathit{tl}_1,\pi_1,F_1),\kappa_1) \qquad T'=\{\mathtt{t}\leadsto((\mathit{tl}_1,\pi_1,F_1),\kappa_1)::((\mathit{tl},\pi,F),\kappa')::K\}\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\tau}((\Pi,T'),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}'))}\ \text{EFCall}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl}_1,\pi_1,F_1),\kappa_1)::((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi_1,F_1)\vdash(\kappa_1,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{ret}(v)}(\kappa_1',\sigma) \qquad \mathbb{d}(\mathtt{t})=0 \\ \mathsf{AftExtn}(\kappa,v)=\kappa' \qquad T'=T\{\mathtt{t}\leadsto((\mathit{tl},\pi,F),\kappa')::K\}\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\tau}((\Pi,T'),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))}\ \text{EFRet}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{entA}}(\kappa',\sigma) \\ \mathbb{d}(\mathtt{t})=0 \qquad T'=T\{\mathtt{t}\leadsto((\mathit{tl},\pi,F),\kappa')::K\} \qquad \mathtt{t}'\in\mathsf{dom}(T)\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\mathsf{sw}}((\Pi,T'),(\mathtt{t}',\mathbb{d}\{\mathtt{t}\leadsto 1\},\sigma,\mathit{FS}))}\ \text{EntAt}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::K \qquad (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{extA}}(\kappa',\sigma) \\ \mathbb{d}(\mathtt{t})=1 \qquad T'=T\{\mathtt{t}\leadsto((\mathit{tl},\pi,F),\kappa')::K\} \qquad \mathtt{t}'\in\mathsf{dom}(T)\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\mathsf{sw}}((\Pi,T'),(\mathtt{t}',\mathbb{d}\{\mathtt{t}\leadsto 0\},\sigma,\mathit{FS}))}\ \text{ExtAt}_{\text{np}}$$

$$\frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::\epsilon \qquad \mathbb{d}(\mathtt{t})=0 \\ (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{ret}(v)}(\kappa',\sigma) \qquad \mathtt{t}'\in\mathsf{dom}(T\backslash\mathtt{t})\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\mathsf{sw}}((\Pi,T\backslash\mathtt{t}),(\mathtt{t}',\mathbb{d}\backslash\mathtt{t},\sigma,\mathit{FS}))}\ \text{Term}_{\text{np}} \qquad \frac{\begin{array}{c}T(\mathtt{t})=((\mathit{tl},\pi,F),\kappa)::\epsilon \qquad \mathbb{d}(\mathtt{t})=0 \\ (\pi,F)\vdash(\kappa,\sigma)\xmapsto[\mathsf{emp}]{\mathsf{ret}(v)}(\kappa',\sigma) \qquad \mathsf{dom}(T)=\{\mathtt{t}\}\end{array}}{((\Pi,T),(\mathtt{t},\mathbb{d},\sigma,\mathit{FS}))\ :\xRightarrow[\mathsf{emp}]{\tau}\mathbf{done}}\ \text{Done}_{\text{np}}$$

Figure 10. The Non-Preemptive Global Semantics

5. The Footprint-Preserving Compositional Simulation (with EFCalls)

6. The Footprint-Preserving Compositional Simulation

In this section, we define a module-local simulation as the correctness obligation of each module's compilation, which is compositional and preserves footprints, allowing us to derive a whole-program simulation that preserves NPDRF. We will discuss compositionality in Sec. 6.2 and postpone the discussions of DRF and NPDRF preservation to Sec. 7.

6.1 Definition of the Module-Local Simulation

As informally explained in Sec. 2, the simulation establishes a consistency relation between executions of the source module γ and the target one π. To achieve compositionality, our simulation is also parameterized with rely/guarantee conditions, specifying the interactions between the current module and its environment at synchronization points. The consistency relation should be preserved under the environment transitions allowed in the rely condition.

Before explaining the simulation definition in Def. 28, we first define some key conditions in Fig. 19. Our simulation is parameterized with μ, defined as follows.

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

Here S and $\mathbb{S}$ specify the memory locations that are currently shared or were once shared. Keeping track of S and $\mathbb{S}$ allows us to define the fixed specifications of rely/guarantee conditions in our simulation. The mapping f maps shared locations (including those once shared) at the source level to shared locations at the target. We require it to be an injective function, i.e., different source locations should be mapped to different target locations.

We require μ be well-formed with respect to the current thread's free spaces $\mathbb{F}$ and F at the source and target levels. As defined in Fig. 19, $\mathsf{wf}(\mu, \mathbb{F}, F)$ says that the injective function f in μ maps shared locations (in $\mathbb{S}$) to shared locations (in S). In particular, a location in $\mathbb{F}$ that is shared (i.e., that has been exported to other threads) must be mapped to a shared location in F. Here $f\{\!\!\{\mathbb{S}\}\!\!\}$ (defined at the bottom of the figure) returns the set of target locations that are mapped from locations in $\mathbb{S}$. $f\{\!\!\{\mathbb{F} \cap \mathbb{S}\}\!\!\}$ is defined similarly.

$\mathsf{FPmatch}(\Delta, \delta, \mu, F)$ relates the footprints Δ and δ at source and target levels. It says, every location accessed at the target (i.e., in δ) must either be from the current thread F, or correspond to a shared location at the source. For the latter case, the location must be accessed at the source (i.e., in Δ).

In our simulation μ may change after related steps at the source and target levels. Fig. 19 defines how μ is allowed to evolve to μ' under steps of the current thread (see EvolveG) and under the environment steps (see EvolveR). First, as defined in Evolve, the new μ' should be well-formed. The mapping f in μ' should relate the new states σ' and Σ' using Inv. Here Inv requires that the locations related by f be contained in the states, and the contents of these locations be also related (see $\overset{f}{\hookrightarrow}$). Besides, S and $\mathbb{S}$ are evolved to reachable closures in the new states σ' and Σ' respectively. We define the closure function $\mathsf{cl}(\mathbb{S}, \Sigma)$ at the bottom of the figure, which is specialized to the CompCert memory model to simplify the presentation. It returns all the locations reachable from $\mathbb{S}$. Since the CompCert memory model allows pointer arithmetics within blocks, we know all the locations in a reachable block (i.e., a block in which some location is reachable) are reachable. The last condition of Evolve says that f in the original μ should be preserved in the new μ' (here function subset is defined by viewing functions as special relations). EvolveG for the current thread's steps requires that the

$\mathsf{HFPG}(\Delta, (\mu, \mathbb{F}))$ iff $\mathsf{FPG}(\Delta, \mathbb{F}, \mu.\mathbb{S})$

$\mathsf{LFPG}(\delta, (\mu, \Delta, F))$ iff $\mathsf{FPG}(\delta, F, \mu.S) \wedge \mathsf{FPmatch}(\mu, \Delta, \delta)$

$\mathsf{FPmatch}(\mu, \Delta, \delta)$ iff
$\quad (\delta.\mathit{rs} \cap \lfloor\!\lfloor \mu.S \rfloor\!\rfloor \subseteq \mu.f\langle\!\langle \Delta.\mathit{rs} \rangle\!\rangle) \wedge (\delta.\mathit{ws} \cap \lfloor\!\lfloor \mu.S \rfloor\!\rfloor \subseteq \mu.f\langle\!\langle \Delta.\mathit{ws} \rangle\!\rangle)$

$\mathsf{FPG}(\Delta, \mathbb{F}, \mathbb{S})$ iff $\mathsf{blocks}(\Delta.\mathit{rs} \cup \Delta.\mathit{ws}) \subseteq \mathbb{F} \cup \mathbb{S}$

$\mathsf{HG}((\Delta, \Sigma'), (\mu, \mathbb{F}))$ iff $\mathsf{G}(\Delta, \Sigma', \mathbb{F}, \mu.\mathbb{S})$

$\mathsf{LG}((\delta, \sigma'), (\mu, \Delta, \Sigma', F))$ iff
$\quad \mathsf{G}(\delta, \sigma', F, \mu.S) \wedge \mathsf{FPmatch}(\mu, \Delta, \delta) \wedge \mathsf{Inv}(\mu.f, \Sigma', \sigma')$

$\mathsf{G}(\Delta, \Sigma', \mathbb{F}, \mathbb{S})$ iff $\mathsf{FPG}(\Delta, \mathbb{F}, \mathbb{S}) \wedge (\mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma'))$

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

$\mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S})$ iff
$\quad (\Sigma \overset{\lfloor\!\lfloor \mathbb{F} \rfloor\!\rfloor}{=\!=\!=} \Sigma') \wedge (\mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma')) \wedge$
$\quad \mathsf{forward}(\Sigma, \Sigma') \wedge ((\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma)) \cap \mathbb{F} = \emptyset)$

$\mathsf{Inv}(f, \Sigma, \sigma)$ iff
$\quad \forall b, n, b', n'.\ ((b, n) \in \mathsf{locs}(\Sigma)) \wedge (f\langle(b, n)\rangle = (b', n')) \Longrightarrow$
$\quad ((b', n') \in \mathsf{locs}(\sigma)) \wedge (\Sigma(b)(n) \overset{f}{\hookrightarrow} \sigma(b')(n'))$

$v_1 \overset{f}{\hookrightarrow} v_2$ iff
$\quad (v_1 \notin \mathfrak{U}) \wedge (v_1 = v_2) \vee v_1 \in \mathfrak{U} \wedge v_2 \in \mathfrak{U} \wedge f\langle v_1 \rangle = v_2$

$\vec{v} \overset{f}{\hookrightarrow} \vec{v}'$ iff $\exists v_1, v'_1, \ldots, v_n, v'_n.$
$\quad \vec{v} = \{v_1, \ldots, v_n\} \wedge \vec{v}' = \{v'_1, \ldots, v'_n\} \wedge \forall i.\ v_i \overset{f}{\hookrightarrow} v'_i$

$\iota \overset{f}{\hookrightarrow} \iota'$ iff
$\quad \iota = \iota' = e \vee \iota = \iota' = \mathsf{entA} \vee \iota = \iota' = \mathsf{extA} \vee$
$\quad \exists \mathtt{f}, \vec{v}, \vec{v}'.\ \iota = \mathsf{call}(\mathtt{f}, \vec{v}) \wedge \iota' = \mathsf{call}(\mathtt{f}, \vec{v}') \wedge \vec{v} \overset{f}{\hookrightarrow} \vec{v}' \vee$
$\quad \exists v, v'.\ \iota = \mathsf{ret}(v) \wedge \iota' = \mathsf{ret}(v') \wedge v \overset{f}{\hookrightarrow} v'$

$\mathsf{Gargs}(\iota, \mathbb{S})$ iff
$\quad \iota = e \vee \iota = \mathsf{entA} \vee \iota = \mathsf{extA} \vee$
$\quad \exists \mathtt{f}, \vec{v}.\ \iota = \mathsf{call}(\mathtt{f}, \vec{v}) \wedge (\forall b, n.\ (b, n) \in \vec{v} \implies b \in \mathbb{S}) \vee$
$\quad \exists v, .\ \iota = \mathsf{ret}(v) \wedge (\forall b, n.\ v = (b, n) \implies b \in \mathbb{S})$

$\mathsf{injective}(f)$ iff
$\quad \forall b_1, b_2, b'_1, b'_2.\ b_1 \neq b_2 \wedge f(b_1) = b'_1 \wedge f(b_2) = b'_2 \implies b'_1 \neq b'_2$

$f\{\!\!\{\mathbb{S}\}\!\!\} \overset{\text{def}}{=} \{b' \mid \exists b.\ (b \in \mathbb{S}) \wedge f(b) = b'\}$

$f\langle l \rangle \overset{\text{def}}{=} (b', n)\ ,\quad$ if $l = (b, n) \wedge f(b) = b'$

$f\langle\!\langle \mathit{ws} \rangle\!\rangle \overset{\text{def}}{=} \{l' \mid \exists l.\ (l \in \mathit{ws}) \wedge f\langle l \rangle = l'\}$

$f|_{\mathbb{S}} \overset{\text{def}}{=} \{(b, f(b)) \mid b \in (\mathbb{S} \cap \mathsf{dom}(f))\}$

$f_2 \circ f_1 \overset{\text{def}}{=} \{(b_1, b_3) \mid \exists b_2.\ f_1(b_1) = b_2 \wedge f_2(b_2) = b_3\}$

$\mathsf{cl}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \bigcup_k \mathsf{cl}_k(\mathbb{S}, \Sigma)\ ,$ where $\mathsf{cl}_k(\mathbb{S}, \Sigma)$ is inductively defined:
$\quad \mathsf{cl}_0(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \mathbb{S}$
$\quad \mathsf{cl}_{k+1}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \{b' \mid \exists b, n, n'.\ (b \in \mathsf{cl}_k(\mathbb{S}, \Sigma)) \wedge \Sigma(b)(n) = (b', n')\}$

Figure 11. Footprint Matching and Rely/Guarantee Conditions in Our Simulation

additional locations in $\mu'.f$ be allocated from the current thread's $\mathbb{F}$, while $\mathsf{EvolveR}$ for the environment steps says the opposite.

Definition 4 (Module-Local Downward Simulation)**.**
$(sl, \gamma) \preccurlyeq_{ge,\varphi} (tl, \pi)$ iff
$\mathsf{dom}(sl.\mathsf{InitCore}(\gamma)) = \mathsf{dom}(tl.\mathsf{InitCore}(\pi))$ and
for any $\mathtt{f}$, κ, $\mathbb{k}$, σ, Σ, μ, $\mathbb{F}$, F, $\mathbb{S}$ and S, if $sl.\mathsf{InitCore}(\gamma)(\mathtt{f}) = \mathbb{k}$, $tl.\mathsf{InitCore}(\pi)(\mathtt{f}) = \kappa$, $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{dom}(\Sigma) = \mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma)$, $\mathsf{dom}(\sigma) = S = \varphi\{\!\{\mathbb{S}\}\!\}$, $\mathsf{locs}(\sigma) = \varphi\langle\!\langle \mathsf{locs}(\Sigma)\rangle\!\rangle$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$, $\mathbb{F} \cap \mathbb{S} = F \cap S = \emptyset$ and $\mu = (\mathbb{S}, S, \varphi|_{\mathbb{S}})$, then there exists $i \in \mathtt{index}$ such that $((\gamma, \mathbb{F}), (\mathbb{k}, \Sigma), \bullet) \preccurlyeq_{\mu}^{(i,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa, \sigma), \bullet)$.

Here we define $((\gamma, \mathbb{F}), (\mathbb{k}, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$ (where the bit β is either $\circ$ or $\bullet$) as the largest relation such that whenever $((\gamma, \mathbb{F}), (\mathbb{k}, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$, then the following are true:

1. $\forall \mathbb{k}', \Sigma', \Delta.$ if $\beta = \circ$, $(\gamma, \mathbb{F}) \vdash (\mathbb{k}, \Sigma) \xmapsto[\Delta]{\tau} (\mathbb{k}', \Sigma')$ and $\mathsf{HFPG}(\Delta_0 \cup \Delta, (\mu, \mathbb{F}))$, then one of the following holds:
   (a) there exists j such that
      i. $j < i$, and
      ii. $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0))} ((\pi, F), (\kappa, \sigma), \circ)$;
   (b) or, there exist κ', σ', δ and j such that
      i. $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{+} (\kappa', \sigma')$, and
      ii. $\mathsf{LFPG}(\delta_0 \cup \delta, (\mu, \Delta_0 \cup \Delta, F))$, and
      iii. $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0 \cup \delta))} ((\pi, F), (\kappa', \sigma'), \circ)$;
2. $\forall \mathbb{k}', \iota.$ if $\beta = \circ$, $\iota \neq \tau$, $(\gamma, \mathbb{F}) \vdash (\mathbb{k}, \Sigma) \xmapsto[\mathtt{emp}]{\iota} (\mathbb{k}', \Sigma)$ and $\mathsf{HG}((\Delta_0, \Sigma), (\mu, \mathbb{F}))$ and $\mathsf{Gargs}(\iota, \mu.\mathbb{S})$, then there exist κ', δ, ι', σ', κ'' and j such that
   (a) $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{*} (\kappa', \sigma')$, and
   $(\pi, F) \vdash (\kappa', \sigma') \xmapsto[\mathtt{emp}]{\iota'} (\kappa'', \sigma')$, and
   (b) $\mathsf{LG}((\delta_0 \cup \delta, \sigma'), (\mu, \Delta_0, \Sigma, F))$, and $\iota \overset{\mu.f}{\hookrightarrow} \iota'$, and
   (c) $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma), \bullet) \preccurlyeq_{\mu}^{(j,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa'', \sigma'), \bullet)$
   or $\iota = \mathsf{ret}(v)$;
3. $\forall \sigma', \Sigma', v_s, \mathbb{k}', v_t,$ if $\beta = \bullet$ and $\mathsf{AftExtn}(\mathbb{k}, v_s) = \mathbb{k}'$ and $v_s \overset{\mu.f}{\hookrightarrow} v_t$ $\mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F))$,
   then there exists κ', j such that
   (a) $\mathsf{AftExtn}(\kappa, v_t) = \kappa'$, and
   (b) $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa', \sigma'), \circ)$.

The simulation $(sl, \gamma) \preccurlyeq (tl, \pi)$ relates the executions of the source module γ and the target module π. We first use the languages' $\mathsf{InitCore}$ functions (see Fig. 4) to initialize the "core" states $\mathbb{k}$ and κ. We assume their executions start from the same initial memory states at the source and target levels, and the whole memory is shared between the current thread and its environment. We define μ with the injective function $\mathsf{id}_{\mathbb{S}}$, which is the identity function on $\mathbb{S}$. That is, $\mathsf{dom}(\mathsf{id}_{\mathbb{S}}) = \mathbb{S}$ and $\forall l \in \mathbb{S}.\ \mathsf{id}_{\mathbb{S}}(l) = l$.

The index i and the history footprints (Δ_0, δ_0) are introduced to ensure footprint preservation. Since compilations may reorder instructions, it may be more natural to relate the footprints of *multiple* source steps to the corresponding target footprints. Our simulation allows one to accumulate the footprints using the history footprints (Δ_0, δ_0), and check $\mathsf{FPmatch}$ (the footprint matching condition, defined in Fig. 19) later. Note that one may choose to check $\mathsf{FPmatch}$ at every source step. Whether to accumulate footprints or check $\mathsf{FPmatch}$ depends on the compilation applications, and we leave the choices to verifiers. However, for infinite executions, it is not allowed to continuously accumulate the footprints and never check $\mathsf{FPmatch}$. Therefore our simulation is parameterized with a metric i from a well-founded set $\mathtt{index}$. The metric should decrease at steps that accumulate the footprints and could be reset at steps that check $\mathsf{FPmatch}$.

We also introduce a bit β to indicate the synchronization points (i.e., the program points when the control may switch to the environment). It takes two values $\bullet$ and $\circ$. Initially it is $\bullet$, indicating a possible switch that allows the environment threads to make steps before the current thread starts. Internal τ-steps of the current thread keeps β to be $\circ$ (see condition 1 in Def. 28). When the current thread makes a transition labeled with e, entA or extA (see condition 2 in Def. 28), we set β from $\circ$ to $\bullet$. The environment can interfere with the current thread when β is $\bullet$ (see condition 3 in Def. 28). After the environment interference, β should be reset to $\circ$.

Our simulation definition follows the diagram in Fig. 2(b). Every source τ-step should correspond to zero-or-multiple target τ-steps (see condition 1 in Def. 28). One may check $\mathsf{FPmatch}$ for footprints accumulated till the current steps and reset the metric (see 1(a)), or choose to continue accumulating the footprints and decrease the metric (see 1(b)). To simplify the presentation, [[[??]]] the footprints should be accumulated when the source step correspond to zero target steps.

At switch points (see condition 2 in Def. 28), we check $\mathsf{FPmatch}$ for the accumulated footprints and reset the metric. We also evolve μ to μ', which satisfies $\mathsf{EvolveG}$ (defined in Fig. 19).

We require the simulation relation is preserved after the environment interference (see condition 3 in Def. 28). The environment steps should not change the current thread's local memory. The condition 3(a) disallow the environment threads to update locations in the current thread's $\mathbb{F}$ (or F at the target level) except in the shared parts $\mathbb{S}$ (or corresponds to $\mathbb{S}$). Here $\doteq$ is defined at the bottom of Fig. 4. After the environment step, μ is evolved to μ' satisfying $\mathsf{EvolveR}$ (defined in Fig. 19).

6.2 Compositionality and the Non-Preemptive Global Simulation

Definition 5 (Whole-Program Downward Simulation)**.**
$\hat{\mathbb{P}} \preccurlyeq_{ge,\varphi} \hat{P}$ iff
there exist $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$ and $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$, such that $\hat{\mathbb{P}} = \textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\hat{P} = \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$,
$\forall i \in \{1, \ldots, m\}.\ \mathsf{dom}(sl_i.\mathsf{InitCore}(\gamma_i)) = \mathsf{dom}(tl_i.\mathsf{InitCore}(\pi_i))$
and
for any Σ, σ, μ, $\widehat{\mathbb{W}}$, if $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{dom}(\Sigma) = \mathbb{S} \subseteq \mathsf{dom}(\varphi)$, $\mathsf{cl}(\mathbb{S}, \Sigma) = \mathbb{S}$, $\mathsf{locs}(\sigma) = \varphi\langle\!\langle \mathsf{locs}(\Sigma) \rangle\!\rangle$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$, $\mu = (\mathbb{S}, \varphi\{\!\{\mathbb{S}\}\!\}, \varphi|_{\mathbb{S}})$, and $(\hat{\mathbb{P}}, \Sigma) :\xRightarrow{load} \widehat{\mathbb{W}}$,
then there exists $\widehat{W}$, $i \in \mathtt{index}$ such that
$(\hat{P}, \sigma) :\xRightarrow{load} \widehat{W}$ and $\widehat{\mathbb{W}} \preccurlyeq_{\mu}^{(i,(\mathtt{emp},\mathtt{emp}))} \widehat{W}$.

Here we define $\widehat{\mathbb{W}} \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} \widehat{W}$ as the largest relation such that whenever $\widehat{\mathbb{W}} \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} \widehat{W}$, then the following are true:

1. $\widehat{\mathbb{W}}.\mathtt{t} = \widehat{W}.\mathtt{t}$, and $\widehat{\mathbb{W}}.\mathbb{d} = \widehat{W}.\mathbb{d}$, and $\mathsf{wf}(\mu, \Sigma)$;
2. $\forall \widehat{\mathbb{W}}', \Delta$. if $\widehat{\mathbb{W}} :\xRightarrow[\Delta]{\tau} \widehat{\mathbb{W}}'$, then one of the following holds:
   (a) there exists j such that
      i. $j < i$, and
      ii. $\widehat{\mathbb{W}}' \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0))} \widehat{W}$;
   (b) or, there exist $\widehat{W}'$, δ and j such that
      i. $\widehat{W} :\xRightarrow[\delta]{\tau}{}^{+} \widehat{W}'$, and
      ii. $\mathsf{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$, and
      iii. $\widehat{\mathbb{W}}' \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0 \cup \delta))} \widehat{W}'$;
3. $\forall \mathbb{T}', \mathbb{d}', \Sigma', o$. if $o \neq \tau$ and $\exists \mathtt{t}'$. $\widehat{\mathbb{W}} :\xRightarrow[\mathtt{emp}]{o} (\mathbb{T}', \mathtt{t}', \mathbb{d}', \Sigma')$, then there exist $\widehat{W}'$, δ, μ', T', σ' and j such that for any $\mathtt{t}' \in \mathsf{dom}(\mathbb{T}')$, we have
   (a) $\widehat{W} :\xRightarrow[\delta]{\tau}{}^{*} \widehat{W}'$, and $\widehat{W}' :\xRightarrow[\mathtt{emp}]{o} (T', \mathtt{t}', \mathbb{d}', \sigma')$, and
   (b) $\mathsf{FPmatch}(\Delta_0, \delta_0 \cup \delta, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$, and
   (c) $(\mathbb{T}', \mathtt{t}', \mathbb{d}', \Sigma') \preccurlyeq_{\mu'}^{(j,(\mathtt{emp},\mathtt{emp}))} (T', \mathtt{t}', \mathbb{d}', \sigma')$;
4. if $\widehat{\mathbb{W}} :\xRightarrow[\mathtt{emp}]{\tau} \textbf{done}$, then there exist $\widehat{W}'$ and δ such that
   (a) $\widehat{W} :\xRightarrow[\delta]{\tau}{}^{*} \widehat{W}'$, and $\widehat{W}' :\xRightarrow[\mathtt{emp}]{\tau} \textbf{done}$, and
   (b) $\mathsf{FPmatch}(\Delta_0, \delta_0 \cup \delta, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$.

Lemma 6 (Compositionality, ④ in Fig. 3)**.**
For any $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $ge_1, \ldots, ge_m$, ge, φ,
$\Gamma = \{(sl_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$, $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$ such that for any $i \in \{1, \ldots, m\}$ we have $\mathsf{wd}(sl_i)$ and $\mathsf{wd}(tl_i)$, if

$$\forall i \in \{1, \ldots, m\}.\ (sl_i, \gamma_i) \preccurlyeq_{ge_i,\varphi} (tl_i, \pi_i)\ ,$$

and $ge = \bigcup_i ge_i$, then

$$\textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \preccurlyeq_{ge,\varphi} \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n\ .$$

6.3 Flip of the Non-Preemptive Global Simulation

Definition 7 (Whole-Program Upward Simulation)**.**
$\hat{P} \leqslant_{ge,\varphi} \hat{\mathbb{P}}$ iff
there exist $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$ and $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$, such that $\hat{\mathbb{P}} = \textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\hat{P} = \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$,
$\forall i \in \{1, \ldots, m\}.\ \mathsf{dom}(sl_i.\mathsf{InitCore}(\gamma_i)) = \mathsf{dom}(tl_i.\mathsf{InitCore}(\pi_i))$
and

for any Σ, σ, μ, $\widehat{\mathbb{W}}$, if $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{dom}(\Sigma) = \mathbb{S} \subseteq \mathsf{dom}(\varphi)$, $\mathsf{cl}(\mathbb{S}, \Sigma) = \mathbb{S}$, $\mathsf{locs}(\sigma) = \varphi\langle\!\langle \mathsf{locs}(\Sigma) \rangle\!\rangle$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$, $\mu = (\mathbb{S}, \varphi\{\!\{\mathbb{S}\}\!\}, \varphi|_{\mathbb{S}})$, and $(\hat{P}, \sigma) :\xRightarrow{load} \widehat{W}$,
then there exists $\widehat{\mathbb{W}}$, $i \in \mathtt{index}$ such that
$(\hat{\mathbb{P}}, \Sigma) :\xRightarrow{load} \widehat{\mathbb{W}}$ and $\widehat{\mathbb{W}} \preccurlyeq_{\mu}^{(i,(\mathtt{emp},\mathtt{emp}))} \widehat{\mathbb{W}}$.

Here we define $\widehat{W} \leqslant_{\mu}^{(i,(\Delta_0,\delta_0))} \widehat{\mathbb{W}}$ as the largest relation such that whenever $\widehat{W} \leqslant_{\mu}^{(i,(\Delta_0,\delta_0))} \widehat{\mathbb{W}}$, then the following are true:

1. $\widehat{\mathbb{W}}.\mathtt{t} = \widehat{W}.\mathtt{t}$, and $\widehat{\mathbb{W}}.\mathbb{d} = \widehat{W}.\mathbb{d}$, and $\mathsf{wf}(\mu, \Sigma)$;
2. $\forall \widehat{W}', \delta$. if $\widehat{W} :\xRightarrow[\delta]{\tau} \widehat{W}'$, then one of the following holds:
   (a) there exists j such that
      i. $j < i$, and
      ii. $\widehat{W}' \leqslant_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0))} \widehat{\mathbb{W}}$;
   (b) there exist $\widehat{\mathbb{W}}'$, Δ and j such that
      i. $\widehat{\mathbb{W}} :\xRightarrow[\Delta]{\tau}{}^{+} \widehat{\mathbb{W}}'$, and
      ii. $\mathsf{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$, and
      iii. $\widehat{W}' \leqslant_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0 \cup \delta))} \widehat{\mathbb{W}}'$;
3. $\forall T', \mathbb{d}', \sigma', o$. if $o \neq \tau$ and $\exists \mathtt{t}'$. $\widehat{W} :\xRightarrow[\mathtt{emp}]{o} (T', \mathtt{t}', \mathbb{d}', \sigma')$, then there exist $\widehat{\mathbb{W}}'$, Δ, μ', $\mathbb{T}'$, Σ' and j such that for any $\mathtt{t}' \in \mathsf{dom}(T')$, we have
   (a) $\widehat{\mathbb{W}} :\xRightarrow[\Delta]{\tau}{}^{*} \widehat{\mathbb{W}}'$, and $\widehat{\mathbb{W}}' :\xRightarrow[\mathtt{emp}]{o} (\mathbb{T}', \mathtt{t}', \mathbb{d}', \Sigma')$, and
   (b) $\mathsf{FPmatch}(\Delta_0 \cup \Delta, \delta_0, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$, and
   (c) $(T', \mathtt{t}', \mathbb{d}', \sigma') \leqslant_{\mu'}^{(j,(\mathtt{emp},\mathtt{emp}))} (\mathbb{T}', \mathtt{t}', \mathbb{d}', \Sigma')$;
4. if $\widehat{W} :\xRightarrow[\mathtt{emp}]{\tau} \textbf{done}$, then there exist $\widehat{\mathbb{W}}'$ and Δ such that
   (a) $\widehat{\mathbb{W}} :\xRightarrow[\Delta]{\tau}{}^{*} \widehat{\mathbb{W}}'$, and $\widehat{\mathbb{W}}' :\xRightarrow[\mathtt{emp}]{\tau} \textbf{done}$, and
   (b) $\mathsf{FPmatch}(\Delta_0 \cup \Delta, \delta_0, \mu, ((\widehat{W}.T)(\widehat{W}.\mathtt{t})).head.F)$.
5. $\neg(\widehat{W} :\xRightarrow[\mathtt{emp}]{\tau} \textbf{abort})$ and $\neg(\widehat{\mathbb{W}} :\xRightarrow[\mathtt{emp}]{\tau} \textbf{abort})$.

Lemma 8 (Flip, ③ in Fig. 3)**.**
For any $\mathtt{f}_1, \ldots, \mathtt{f}_n$, ge, φ, $\Gamma = \{(sl_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$, $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$, if $\forall i.\ \mathsf{det}(tl_i)$ and $\mathsf{Safe}(\textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m, ge)$

$$\textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \preccurlyeq_{ge,\varphi} \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m\ ,$$

then

$$\textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \leqslant_{ge,\varphi} \textbf{let}\ \Gamma\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m\ .$$

7. Data-Race-Freedom

$$\begin{array}{rll} \delta_1 \smile \delta_2 & \text{iff} & (\delta_1.ws \cap (\delta_2.rs \cup \delta_2.ws) = \emptyset) \wedge \\ & & (\delta_1.rs \cap \delta_2.ws = \emptyset) \\ (\delta_1, d_1) \smile (\delta_2, d_2) & \text{iff} & (\delta_1 \smile \delta_2) \vee (d_1 = d_2 = 1) \end{array}$$

$$\begin{array}{l} \mathsf{DRF}(P, \sigma) \ \text{ iff } \ \neg((P, \sigma) \Longmapsto \mathtt{Race}) \\ \mathsf{DRF}(\mathbb{P}, ge) \ \text{ iff} \\ \quad \forall \Sigma.\ (ge \subseteq \mathsf{locs}(\Sigma)) \wedge (\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)) \implies \\ \quad \mathsf{DRF}(\mathbb{P}, \Sigma) \\ \mathsf{DRF}(\hat{P}, \sigma) \ \text{ iff } \ \neg((\hat{P}, \sigma) :\Longmapsto \mathtt{Race}) \end{array}$$

Lemma 9 (NPDRF Preservation, ⑦ in Fig. 3)**.**
For any $\hat{\mathbb{P}}$, $\hat{P}$, ge, φ, Σ and σ, if $\hat{P} \preccurlyeq_{ge,\varphi} \hat{\mathbb{P}}$, $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)$, $\varphi\{\!\{\mathsf{dom}(\Sigma)\}\!\} = \mathsf{dom}(\sigma)$, $\varphi\langle\!\langle \mathsf{locs}(\Sigma) \rangle\!\rangle = \mathsf{locs}(\sigma)$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$ and $\mathsf{NPDRF}(\hat{\mathbb{P}}, \Sigma)$, then $\mathsf{NPDRF}(\hat{P}, \sigma)$.

Lemma 10 (Equivalence between DRF and NPDRF, ⑥ and ⑧ in Fig. 3)**.**
For any $\mathtt{f}_1, \ldots, \mathtt{f}_m$, σ, $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$ such that $\forall i.\ \mathsf{wd}(tl_i)$,

$$\begin{array}{l} \mathsf{DRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, \sigma) \iff \\ \quad \mathsf{NPDRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m, \sigma)\ . \end{array}$$

Lemma 11 (Equivalence between Preemptive and Non-Preemptive Programs, ① and ② in Fig. 3)**.**
For any Π, $\mathtt{f}_1, \ldots, \mathtt{f}_m$, σ, if $\mathsf{DRF}(\textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, \sigma)$, then

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

$$\frac{(P,\sigma) \xRightarrow{\mathit{load}} W \qquad W \Longmapsto^{+} \texttt{Race}}{(P,\sigma) \Longmapsto \texttt{Race}} \qquad\qquad \frac{W \overset{o}{\underset{\delta}{\Rightarrow}} W'}{W \Longmapsto W'}\ \text{Progress}$$

$$\frac{\texttt{t}_1 \neq \texttt{t}_2 \qquad \texttt{predict}(W, \texttt{t}_1, (\delta_1, d_1)) \qquad \texttt{predict}(W, \texttt{t}_2, (\delta_2, d_2)) \qquad \neg((\delta_1, d_1) \smile (\delta_2, d_2))}{W \Longmapsto \texttt{Race}}\ \text{Race}$$

$$\frac{W = ((\Pi, T), (\_, 0, \sigma, \mathit{FS})) \qquad T(\texttt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau} (\kappa', \sigma')}{\texttt{predict}(W, \texttt{t}, (\delta, 0))}\ \text{Predict0}$$

$$\frac{W = ((\Pi, T), (\_, 0, \sigma, \mathit{FS})) \qquad T(\texttt{t}) = ((\mathit{tl}, \pi, F), \kappa) :: K \qquad (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\texttt{emp}]{\textsf{entA}} (\kappa', \sigma) \qquad (\pi, F) \vdash (\kappa', \sigma) \xmapsto[\delta]{\tau}{}^{*} (\kappa'', \sigma'')}{\texttt{predict}(W, \texttt{t}, (\delta, 1))}\ \text{Predict1}$$

(a) Preemptive Semantics

$$\frac{(\hat{P}, \sigma) :\xRightarrow{\mathit{load}} \widehat{W} \qquad \texttt{t}_1 \neq \texttt{t}_2 \qquad \texttt{NPpredict}(\widehat{W}, \texttt{t}_1, (\delta_1, d_1)) \qquad \texttt{NPpredict}(\widehat{W}, \texttt{t}_2, (\delta_2, d_2)) \qquad \neg((\delta_1, d_1) \smile (\delta_2, d_2))}{(\hat{P}, \sigma) :\Longmapsto \texttt{Race}}$$

$$\frac{(\hat{P}, \sigma) :\xRightarrow{\mathit{load}} \widehat{W} \qquad \widehat{W} :\Longmapsto^{+} \texttt{Race}}{(\hat{P}, \sigma) :\Longmapsto \texttt{Race}} \qquad\qquad \frac{\widehat{W} :\overset{o}{\underset{\delta}{\Rightarrow}} \widehat{W}'}{\widehat{W} :\Longmapsto \widehat{W}'}\ \text{Progress}_{\text{np}}$$

$$\frac{\widehat{W} :\xRightarrow[\texttt{emp}]{o} \widehat{W}' \qquad o \neq \tau \qquad \texttt{t}_1 \neq \texttt{t}_2 \qquad \texttt{NPpredict}(\widehat{W}', \texttt{t}_1, (\delta_1, d_1)) \qquad \texttt{NPpredict}(\widehat{W}', \texttt{t}_2, (\delta_2, d_2)) \qquad \neg((\delta_1, d_1) \smile (\delta_2, d_2))}{\widehat{W} :\Longmapsto \texttt{Race}}\ \text{Race}_{\text{np}}$$

$$\frac{\widehat{W} = ((\Pi, T), (\_, \mathbb{d}, \sigma, \mathit{FS})) \qquad ((\Pi, T), (\texttt{t}, \mathbb{d}, \sigma, \mathit{FS})) :\overset{\tau}{\underset{\delta}{\Rightarrow}}{}^{*} ((\Pi, T'), (\texttt{t}, \mathbb{d}, \sigma', \mathit{FS})) \qquad \mathbb{d}(\texttt{t}) = d}{\texttt{NPpredict}(\widehat{W}, \texttt{t}, (\delta, d))}\ \text{Predict}_{\text{np}}$$

(b) Non-Preemptive Semantics

Figure 12. Predictive Semantics for Defining Race

8. The Final Theorem

$$\begin{array}{rcl}
\mathsf{SeqComp} & ::= & (\mathsf{CodeT}, \mathsf{GE}, \varphi) \\
\mathsf{CodeT} & \in & \mathit{Module} \rightharpoonup \mathit{Module} \\
\mathsf{GE} & \in & \mathit{Module} \rightharpoonup \mathcal{P}(\mathfrak{U}) \\
\mathit{ge} & \in & \mathcal{P}(\mathfrak{U}) \\
\varphi & \in & \mathit{BlockID} \rightharpoonup \mathit{BlockID} \times \mathbb{N} \qquad \text{where } \mathsf{injective}(\varphi)
\end{array}$$

Theorem 12 (Final Theorem)**.**
For any $\mathsf{SeqComp}_1, \ldots, \mathsf{SeqComp}_m$, $sl_1, \ldots, sl_m$, $tl_1, \ldots, tl_m$ such that for any $i \in \{1, \ldots, m\}$ we have $\mathsf{wd}(sl_i)$, $\mathsf{wd}(tl_i)$ and $\mathsf{det}(tl_i)$, if

$$\forall i \in \{1, \ldots, m\}.\ \mathsf{Correct}(\mathsf{SeqComp}_i, sl_i, tl_i)\ ,$$

then

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

Definition 13 (Sequential Compiler Correctness)**.**
$\mathsf{Correct}(\mathsf{SeqComp}, sl, tl)$ iff

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

Definition 14 (Concurrent Compiler Correctness)**.**
$\mathsf{GCorrect}((\mathsf{SeqComp}_1, sl_1, tl_1), \ldots, (\mathsf{SeqComp}_m, sl_m, tl_m))$ iff for any $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$, $\Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$, for any $ge_1, \ldots, ge_m$, ge, φ, if

1. $\forall i \in \{1, \ldots, m\}.\ \mathsf{SeqComp}_i.\mathsf{CodeT}(\gamma_i) = \pi_i$,
2. $\forall i \in \{1, \ldots, m\}.\ \mathsf{SeqComp}_i.\mathsf{GE}(\gamma_i) = ge_i$, $ge = \bigcup_i ge_i$,
   $\forall i \in \{1, \ldots, m\}.\ \mathsf{SeqComp}_i.\varphi = \varphi$,
3. $\mathsf{DRF}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, ge)$, and
   $\mathsf{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, ge)$,
4. $\forall i \in \{1, \ldots, m\}.\ \mathsf{ReachClose}(sl_i, \gamma_i, ge_i)$,

then

$$\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_{ge,\varphi} \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n\ .$$

Definition 15 (Safety)**.** $\mathsf{Safe}(\mathbb{W})$ iff $\neg\exists tr.\ \mathit{Etr}(\mathbb{W}, tr :: \mathbf{abort})$.
$\mathsf{Safe}(\mathbb{P}, \Sigma)$ iff $\forall \mathbb{W}.\ ((\mathbb{P}, \Sigma) \overset{load}{\Longrightarrow} \mathbb{W}) \implies \mathsf{Safe}(\mathbb{W})$.
$\mathsf{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, ge)$ iff

$$\forall \Sigma.\ (ge \subseteq \mathsf{locs}(\Sigma)) \wedge (\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)) \implies \mathsf{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, \Sigma)\ .$$

Definition 16. $\mathsf{ReachClose}(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \mathbb{F}, \mathbb{S}$. if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $\mathsf{blocks}(ge) \subseteq \mathbb{S}$,
3. $\mathbb{S} \cap \mathbb{F} = \emptyset$,

then $\mathsf{RC}((\gamma, \Bbbk), (\mathbb{F}, \mathbb{S}))$.

Here RC is defined as the largest relation such that whenever $\mathsf{RC}((\gamma, \Bbbk), (\mathbb{F}, \mathbb{S}))$, then the following holds: $\forall \Bbbk', \Sigma, \Sigma', \iota, \Delta$. if $\mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma) \subseteq \mathsf{dom}(\Sigma)$ and $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \xmapsto[\Delta]{\iota} (\Bbbk', \Sigma')$, then $\mathsf{G}(\Delta, \Sigma', \mathbb{F}, \mathbb{S})$ and $\mathsf{Gargs}(\iota, \mathbb{S})$ and $\mathsf{RC}((\gamma, \Bbbk'), (\mathbb{F}, \mathbb{S}))$.

Without EFCalls.

Definition 17 (Version I)**.** $\mathsf{ReachClose}_1(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \Sigma, \mathbb{F}, ac.$ if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $ge \subseteq \llbracket ac \rrbracket$, $ac = \mathsf{cl}(ac, \Sigma) \subseteq \mathsf{dom}(\Sigma)$,
3. $\mathsf{dom}(\Sigma) \cap \mathbb{F} = \emptyset$,

then $\mathsf{RC}_1((\gamma, \mathbb{F}), (\Bbbk, \Sigma), ac)$.

Here RC_1 is defined as the largest relation such that whenever $\mathsf{RC}_1((\gamma, \mathbb{F}), (\Bbbk, \Sigma), ac)$, both the following are true:

1. $\forall \Bbbk', \Sigma', \Delta, ac'.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\longmapsto}} (\Bbbk', \Sigma')$ and $ac' = ac \cup (\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma))$, then
   (a) $\mathsf{cl}(ac', \Sigma') = ac'$, and
   (b) $\Delta \subseteq \llbracket ac' \rrbracket$, and
   (c) $\mathsf{RC}_1((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac')$;
2. $\forall \Bbbk', \iota, \Sigma', ac'.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\mathtt{emp}}{\longmapsto}} (\Bbbk', \Sigma)$, $\iota \neq \tau$ and $ac \subseteq ac' = \mathsf{cl}(ac', \Sigma') \subseteq \mathsf{dom}(\Sigma')$, then $\mathsf{RC}_1((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac')$.

Definition 18 (Version II)**.** $\mathsf{ReachClose}_2(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \Sigma, \mathbb{F}, ac.$ if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $ge \subseteq \llbracket ac \rrbracket$, $ac \subseteq \mathsf{dom}(\Sigma)$, $\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)$,
3. $\mathsf{dom}(\Sigma) \cap \mathbb{F} = \emptyset$,

then $\mathsf{RC}_2((\gamma, \mathbb{F}), (\Bbbk, \Sigma), ac)$.

Here RC_2 is defined as the largest relation such that whenever $\mathsf{RC}_2((\gamma, \mathbb{F}), (\Bbbk, \Sigma), ac)$, both the following are true:

1. $\forall \Bbbk', \Sigma', \Delta, ac'.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\longmapsto}} (\Bbbk', \Sigma')$ and $ac' = \mathsf{cl}(ac \cup (\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma)), \Sigma')$, then
   (a) $\Delta \subseteq \llbracket ac' \rrbracket$, and
   (b) $\mathsf{RC}_2((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac')$;
2. $\forall \Bbbk', \iota.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\mathtt{emp}}{\longmapsto}} (\Bbbk', \Sigma)$ and $\iota \neq \tau$, then
   (a) $\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)$, and
   (b) $\forall \Sigma', ac'.$ if $\mathsf{forward}(\Sigma, \Sigma')$, $\mathsf{cl}(\mathsf{dom}(\Sigma'), \Sigma') = \mathsf{dom}(\Sigma')$ and $ac' = \mathsf{cl}(ac, \Sigma')$, then $\mathsf{RC}_2((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac')$.

Definition 19 (Version III)**.** $\mathsf{ReachClose}_3(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \mathbb{F}, ac.$ if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $ge \subseteq \llbracket ac \rrbracket$,
3. $ac \cap \mathbb{F} = \emptyset$,

then $\mathsf{RC}_3((\gamma, \mathbb{F}), \Bbbk, ac)$.

Here RC_3 is defined as the largest relation such that $\forall \gamma, \mathbb{F}, \Bbbk, ac.$ if $\mathsf{RC}_3((\gamma, \mathbb{F}), \Bbbk, ac)$, then $\forall \Bbbk', \Sigma, \Sigma', \iota, \Delta, ac'.$ if $ac \subseteq \mathsf{dom}(\Sigma)$, $\mathsf{cl}(ac, \Sigma) = ac$, $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\Delta}{\longmapsto}} (\Bbbk', \Sigma')$ and $ac' = ac \cup (\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma))$, then

1. $\Delta \subseteq \llbracket ac' \rrbracket$, and
2. $\mathsf{cl}(ac', \Sigma') = ac'$, and
3. $\mathsf{RC}_3((\gamma, \mathbb{F}), \Bbbk', ac')$.

Definition 20 (ReachClose in CompComp)**.** $\mathsf{ReachClose}(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \Sigma, \mathbb{F}.$ if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $\mathsf{dom}(\Sigma) \cap \mathbb{F} = \emptyset$,

then $\mathsf{RC}((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \emptyset)$.

Here RC is defined as the largest relation such that whenever $\mathsf{RC}((\gamma, \mathbb{F}), (\Bbbk, \Sigma), ac)$, both the following are true:

1. $\forall \Bbbk', \Sigma', \Delta, ac'.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\longmapsto}} (\Bbbk', \Sigma')$ and $ac' = \mathsf{cl}(\mathsf{cl}(\mathsf{getBlocks}(ge) \cup ac, \Sigma) \cup (\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma)), \Sigma')$, then
   (a) $\Delta \subseteq \llbracket \mathsf{cl}(\mathsf{getBlocks}(ge) \cup ac, \Sigma) \rrbracket$, and
   (b) $\mathsf{RC}((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac')$;
2. $\forall \Bbbk', \iota, \Sigma'.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\mathtt{emp}}{\longmapsto}} (\Bbbk', \Sigma)$ and $\iota \neq \tau$, then $\mathsf{RC}((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), ac)$.

Definition 21 (Nucular in CompComp)**.** $\mathsf{Nucular}(sl, \gamma, ge)$ iff $\forall \Bbbk, \mathtt{f}, \Sigma, \mathbb{F}.$ if

1. $sl.\mathsf{InitCore}(\gamma, \mathtt{f}) = \Bbbk$,
2. $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)$,
3. $\mathsf{dom}(\Sigma) \cap \mathbb{F} = \emptyset$,

then $\mathsf{nucular}((\gamma, \mathbb{F}), (\Bbbk, \Sigma))$.

Here nucular is defined as the largest relation such that whenever $\mathsf{nucular}((\gamma, \mathbb{F}), (\Bbbk, \Sigma))$, both the following are true:

1. $\forall \Bbbk', \Sigma', \Delta.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\longmapsto}} (\Bbbk', \Sigma')$, then $\mathsf{nucular}((\gamma, \mathbb{F}), (\Bbbk', \Sigma'))$;
2. $\forall \Bbbk', \iota.$ if $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\mathtt{emp}}{\longmapsto}} (\Bbbk', \Sigma)$ and $\iota \neq \tau$, then
   (a) $\mathsf{cl}(\mathsf{dom}(\Sigma), \Sigma) = \mathsf{dom}(\Sigma)$, and
   (b) $\forall \Sigma'.$ if $\mathsf{forward}(\Sigma, \Sigma')$ and $\mathsf{cl}(\mathsf{dom}(\Sigma'), \Sigma') = \mathsf{dom}(\Sigma')$, then $\mathsf{nucular}((\gamma, \mathbb{F}), (\Bbbk', \Sigma'))$.

9. Related Work and Conclusions

9.1 Compare with CompCertX

The CompCertX compiler is developed to support separate compilation of OS code. Taking the advantage of their push/pull memory model and log-based CPU-local machine, they are able to support parallel composition without major modification to CompCert. **[[HOW?]]**

But there are also limitations caused by the push/pull memory model:

- Program have to use push/pull primitives to synchronize between CPUs. E.g., given a properly implemented lock l, to get access to a shared variable x, the program has to be like the left one. Where in our approach, there is no need to introduce push/pull primitives, as shown on the right.

`lock(`l`);`
`pull(&`x`);`
`write(&`x`, 1);`
`push(&`x`);`
`unlock(`l`);`
	`lock(`l`);`
	`write(&`x`, 1);`
	`unlock(`l`);`

  [[Why are we able to be simpler? No local copy?]]
  What's more, the push/pull primitives introduce redundant function calls, causing significant latency on their spin-lock implementations. For performance they remoded push/pull primitives during "pretty-printing" phase, which is not verified. This is not an issue in our approach since we do not introduce unnecessary "null calls".

- There is no way to share a memory block unless it corresponds to some global identifier. Since the push/pull primitives are instantiated as CompCert external calls, their arguments must be blocks corresponding to some global identifier, as required by CompCert. Thus, for example, they are not able to implement a general `malloc` function and share `malloc`ed memory amoung CPUs, e.g. appending to a shared linked list. In their approach, appending new node to a linked list is implemented by modifying a pre-allocated memory spaces (such as a globally declared array).

Of course they could not allow two CPUs simultaneously accessing different localtions in the same block, or simultaneously reading the same location, since they push/pull by block and each block is accessable by only one CPU.

References

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

[2] A. Lochbihler. Verifying a compiler for java threads. In *ESOP*, pages 427–447, 2010.

[3] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, pages 43–54, 2011.

[4] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *POPL*, pages 275–287, 2015.

$$\begin{array}{lcll}
(\mathit{HMod}) & \gamma & ::= & \{\mathtt{f}_1 \rightsquigarrow (\vec{x}_1, \mathbb{C}_1), \ldots, \mathtt{f}_k \rightsquigarrow (\vec{x}_k, \mathbb{C}_k)\} \\
(\mathit{HStmt}) & \mathbb{C} & ::= & \textbf{skip} \mid \mathbb{E} := \mathbb{E} \mid x := \mathtt{f}(\vec{\mathbb{E}}) \mid \textbf{return}(\mathbb{E}) \\
 & & \mid & \mathbb{C};\mathbb{C} \mid \textbf{if } (\mathbb{B})\ \mathbb{C} \textbf{ else } \mathbb{C} \mid \textbf{while } (\mathbb{B})\{\mathbb{C}\} \\
 & & \mid & \textbf{sc\_store}(\mathbb{E}, \mathbb{E}) \mid x := \textbf{sc\_load}(\mathbb{E}) \\
(\mathit{HExpr}) & \mathbb{E} & ::= & n \mid x \mid *\mathbb{E} \mid \&\mathbb{E} \mid \mathbb{E} + \mathbb{E} \mid \mathbb{E} - \mathbb{E} \mid \mathbb{E} \times \mathbb{E} \\
(\mathit{HBExp}) & \mathbb{B} & ::= & \textbf{true} \mid \textbf{false} \mid \mathbb{E} = \mathbb{E} \mid \mathbb{E} \leq \mathbb{E} \mid \textbf{not}\ \mathbb{B} \mid \mathbb{B} \textbf{ and } \mathbb{B} \\
(\mathit{HCore}) & \Bbbk & ::= & \mathbb{C} \mid \Bbbk_1;\Bbbk_2 \mid \ldots
\end{array}$$

$$\mathsf{InitCore}(\gamma)(\mathtt{f})(\vec{v}) \stackrel{\text{def}}{=} (\vec{x} := \vec{v}; \mathbb{C}) \qquad \text{if } \gamma(\mathtt{f}) = (\vec{x}, \mathbb{C})$$

$$(\gamma, \mathbb{F}) \vdash (x := \mathtt{f}(\vec{v}), \Sigma) \overset{\mathsf{call}(\mathtt{f}, \vec{v})}{\underset{\mathsf{emp}}{\longmapsto}} (x := \mathtt{f}(\vec{v}), \Sigma) \qquad \text{if } \mathtt{f} \notin \mathsf{dom}(\gamma)$$

$$(\gamma, \mathbb{F}) \vdash (x := \mathtt{f}(\vec{v}), \Sigma) \overset{\mathsf{recv}(v')}{\dashrightarrow} (x := v', \Sigma) \qquad \text{if } \mathtt{f} \notin \mathsf{dom}(\gamma)$$

$$(\gamma, \mathbb{F}) \vdash (\textbf{return}(v'), \Sigma) \overset{\mathsf{ret}(v')}{\underset{\mathsf{emp}}{\longmapsto}} (\textbf{skip}, \Sigma)$$

$$(\gamma, \mathbb{F}) \vdash (x := \textbf{sc\_load}(src); \Bbbk) \overset{\mathsf{call}(\textbf{sc\_load}, src)}{\underset{\mathsf{emp}}{\longmapsto}} (x := \textbf{sc\_load}(src); \Bbbk)$$

$$(\gamma, \mathbb{F}) \vdash (\textbf{sc\_store}(dst, v); \Bbbk) \overset{\mathsf{call}(\textbf{sc\_store}, dst::v)}{\underset{\mathsf{emp}}{\longmapsto}} (\textbf{sc\_store}(dst, v); \Bbbk)$$

Figure 13. Source language: Clight + SC atomics

10. The Source Language: Clight + SC Atomics

The whole program **let** $\Gamma \bigcup \{(lang_{\mathrm{SC}}, \gamma_{\mathrm{SC}})\}$ **in** $\mathtt{f}_1(\vec{v}_1) \parallel \ldots \parallel \mathtt{f}_n(\vec{v}_n)$: Here Γ is the collection of source modules written by the application programmers.

sc_store and **sc_load**: In the views of the application programmers, they are primitives that would generate external function calls.

To define the semantics of sc-primitives, we introduce a built-in module $(lang_{\mathrm{SC}}, \gamma_{\mathrm{SC}})$, where $lang_{\mathrm{SC}}$ is defined in Fig.14. On a sc-primitive call, it first generate an entA event, then performs the corresponding memory operation; before returning the resulting value, it generate an extA event. The entA and extA events guarantees that the memory operation is exclusive to the environment.

$$\begin{array}{lcll}
(\mathit{SCSyntax}) & \mathbb{C} & ::= & \mathtt{Ent\ sc\_op} \mid \mathtt{sc\_op} \mid \mathtt{Ext\ sc\_ret} \mid \mathtt{sc\_ret} \mid \mathtt{END} \\
(\mathit{SCMod}) & \gamma_{\mathrm{sc}} & ::= & \{\textbf{sc\_load} \rightsquigarrow \mathbb{C}_1, \textbf{sc\_store} \rightsquigarrow \mathbb{C}_2\} \\
(\mathit{SCCore}) & \Bbbk_{\mathrm{sc}} & ::= & \mathbb{C} \\
(\mathit{SCOp}) & \mathtt{sc\_op} & ::= & \mathtt{load}(src) \mid \mathtt{store}(dst, v) \\
(\mathit{SCVar}) & src, dst, v & ::= & \ldots \\
(\mathit{SCRet}) & \mathtt{sc\_ret} & ::= & \mathtt{load\_ret}(v) \mid \mathtt{store\_ret}
\end{array}$$

$$\mathsf{InitCore}_{\mathrm{SC}}(\gamma)(\mathtt{f})(\vec{v}) \stackrel{\text{def}}{=}$$
$$\begin{cases} (\mathtt{Ent}\ (\mathtt{load}(src))) & \text{if } \mathtt{f} = \textbf{sc\_load} \wedge \vec{v} = src :: nil \\ (\mathtt{Ent}\ (\mathtt{store}(dst, v'))) & \text{if } \mathtt{f} = \textbf{sc\_store} \wedge \vec{v} = dst :: v' :: nil \end{cases}$$

$$(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{Ent\ sc\_op}, \Sigma) \overset{\mathsf{entA}}{\longmapsto} (\mathtt{Ent\ sc\_op}, \Sigma)$$

$$(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{Ent\ sc\_op}, \Sigma) \overset{\mathsf{recv}()}{\dashrightarrow} (\mathtt{sc\_op}, \Sigma)$$

$$\frac{\begin{array}{c}\mathtt{sc\_op} = \mathtt{load}(src) \qquad \delta = (\{src\}, \emptyset) \\ \Sigma(src) = v \qquad \mathtt{sc\_ret} = \mathtt{load\_ret}(v)\end{array}}{(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{sc\_op}, \Sigma) \overset{\tau}{\underset{\delta}{\longmapsto}} (\mathtt{Ext\ sc\_ret}, \Sigma)} \text{ Load}$$

$$\frac{\begin{array}{c}\mathtt{sc\_op} = \mathtt{store}(dst, v) \qquad \delta = (\emptyset, \{dst\}) \\ \Sigma' = \Sigma\{src \rightsquigarrow v\} \qquad \mathtt{sc\_ret} = \mathtt{store\_ret}\end{array}}{(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{sc\_op}, \Sigma) \overset{\tau}{\underset{\delta}{\longmapsto}} (\mathtt{Ext\ sc\_ret}, \Sigma')} \text{ Store}$$

$$(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{Ext\ sc\_ret}, \Sigma) \overset{\mathsf{extA}}{\longmapsto} (\mathtt{Ext\ sc\_ret}, \Sigma)$$

$$(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{Ext\ sc\_ret}, \Sigma) \overset{\mathsf{recv}()}{\dashrightarrow} (\mathtt{sc\_ret}, \Sigma)$$

$$\frac{\begin{array}{c}\mathtt{sc\_ret} = \mathtt{store\_ret} \wedge \vec{v} = nil \quad \text{or} \\ \mathtt{sc\_ret} = \mathtt{load\_ret}(v') \wedge \vec{v} = v' :: nil\end{array}}{(\gamma_{\mathrm{sc}}, \mathbb{F}) \vdash (\mathtt{sc\_ret}, \Sigma) \overset{\mathsf{ret}(\vec{v})}{\underset{\mathsf{emp}}{\longmapsto}} (\mathtt{END}, \Sigma)} \text{ Returen}$$

$$lang_{\mathrm{SC}} \stackrel{\text{def}}{=} (\mathit{SCMod}, \mathit{SCCore}, \mathsf{InitCore}_{\mathrm{SC}}, \longmapsto, \dashrightarrow)$$

Figure 14. SC_atomic lang

$$
\begin{array}{rcll}
(\mathit{LMod}) & \pi & ::= & \{\mathtt{f}_1 \rightsquigarrow C_1, \ldots, \mathtt{f}_k \rightsquigarrow C_k\} \\
(\mathit{LCode}) & C & ::= & \epsilon \mid c :: C \\
(\mathit{LInstr}) & c & ::= & \mathtt{mov}\ \mathtt{r}_d, \mathtt{r}_s \mid \mathtt{mov}\ \mathtt{r}_d, v \mid \mathtt{mov}\ l, \mathtt{r}_s \\
 & & \mid & \mathtt{add}\ \mathtt{r}_d, \mathtt{r}_s \mid \ldots \\
 & & \mid & \mathtt{call}\ \mathtt{f} \mid \mathtt{ret} \\
 & & \mid & \mathtt{lock}\ \mathtt{xchg}\ l\ \mathtt{r}_s \mid \mathtt{lock}\ \mathtt{xadd}\ l\ \mathtt{r}_s \\
 & & \mid & \mathtt{lock}\ \mathtt{cmpxchg}\ l\ \mathtt{r}_s \\
(\mathit{DynInstr}) & \tilde{c} & ::= & \text{ENTER\_AT} \mid \text{EXIT\_AT} \\
(\mathit{DynCode}) & \tilde{C} & ::= & \epsilon \mid c :: \tilde{C} \mid \tilde{c} :: \tilde{C} \\
(\mathit{Register}) & \mathtt{r} & ::= & \mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31} \\
(\mathit{RegFile}) & R & \in & \mathit{Register} \to \mathit{Value} \\
(\mathit{LCore}) & \kappa & ::= & (\tilde{C}, R)
\end{array}
$$

$$\mathsf{InitCore}(\pi)(\mathtt{f})(\vec{v}) \overset{\text{def}}{=} (\pi(\mathtt{f}), R) \qquad \text{where } R = ????$$

$$\frac{\mathtt{f} \notin \mathsf{dom}(\pi) \qquad R' = ????}{(\pi, F) \vdash (((\mathtt{call}\ \mathtt{f}) :: \tilde{C}, R), \sigma) \xmapsto[\mathsf{emp}]{\mathsf{call}(\mathtt{f})} (((\mathtt{call}\ \mathtt{f}) :: \tilde{C}, R'), \sigma)}$$

$$\frac{R' = ????}{(\pi, F) \vdash ((\mathtt{ret} :: \tilde{C}, R), \sigma) \xmapsto[\mathsf{emp}]{\mathsf{ret}} ((\tilde{C}, R'), \sigma)}$$

$$\frac{}{(\pi, F) \vdash ((\mathtt{lock}\ \mathtt{mov}\ a_1\ a_2) :: \tilde{C}) \xmapsto[\mathsf{emp}]{\tau} (\text{ENTER\_AT} :: (\mathtt{mov}\ a_1\ a_2) :: \text{EXIT\_AT} :: \tilde{C})}$$

$$(\pi, F) \vdash ((\text{ENTER\_AT} :: C, R), \sigma) \xmapsto[\mathsf{emp}]{\mathsf{entA}} ((C, R), \sigma)$$

$$(\pi, F) \vdash ((\text{EXIT\_AT} :: C, R), \sigma) \xmapsto[\mathsf{emp}]{\mathsf{extA}} ((C, R), \sigma)$$

$$\frac{\mathtt{f} \notin \mathsf{dom}(\gamma) \qquad R' = ????}{(\pi, F) \vdash (((\mathtt{call}\ \mathtt{f}) :: C, R), \Sigma) \overset{\mathsf{recv}}{\dashrightarrow} ((C, R'), \Sigma)}$$

Figure 15. Pseudo-x86 Assembly

11. The Target Language: x86 Assembly

The target programs written in the x86 assembly language are simply "printed" from programs in the "pseudo-x86" language defined in Fig. 15. The "pseudo-x86" language is an instantiation of our abstract language.

Pseudo-x86. As in the source language, ENTER_AT and EXIT_AT are used only as dynamic instructiions.

Note that **sc_store** and **sc_load** are external function calls in the source code written by the application programmers. So they will be compiled to "`call` **sc_store**" and "`call` **sc_load**" at the pseudo-x86 level.

Recall that we have a built-in module for sc-primitives. Here we implement them in pseudo-x86:

sc_store :
```
        mov   ecx   [esp + 8]
        mov   edx   [esp + 4]
 lock xchg   [edx]  ecx
        ret
```

sc_load :
```
        xor   eax   eax
        mov   edx   [esp + 4]
 lock xadd   [edx]  eax
        ret
```

sc_cas :
```
           mov   eax   [esp + 8]
           mov   eax   [eax]
           mov   edx   [esp + 4]
           mov   ecx   [esp + 12]
 lock cmpxchg   [edx]  ecx
          sete   al
           ret
```

Printing from pseudo-x86 to x86. Identical transformation.
[[Have to study x86 code for function calls.]]

$$\begin{tikzcd}
 & & & \sigma_3 \\
\sigma \arrow[r, "\mathtt{t}_2", "\delta"'] & \sigma_1 \arrow[r, "\mathtt{t}_1", "\delta_1"'] \arrow[dd, dashed, "\mathtt{t}_2"', "\delta_3"] & \sigma_2 \arrow[ur, "\mathtt{t}_1", "\delta_2"'] \arrow[dr, "\mathtt{t}_2", "\delta_3"'] & \\
 & & & \sigma_4 \\
 & \sigma' & &
\end{tikzcd}$$

Figure 16. example

11.1 Why the (4)th condition was required?

The condition is required in proving Lemma 10, in particular, the "$\Leftarrow$" direction, i.e. $\mathsf{DRF}(\hat{P},\sigma) \implies \mathsf{DRF}(P,\sigma)$. This could be proved by proving

$$((P,\sigma) \Longmapsto \mathtt{Race}) \implies \left((\hat{P},\sigma) :\Longmapsto \mathtt{Race}\right).$$

Intuitively, the (4)th condition is necessary because we need to construct a race-condition under non-preemptive execution, based on some interleaving execution. In this case, we need all footprint generated during a interleaving execution is predictable from some state in a non-preemptive execution.

We illustrate this by an example.

By definition of $\Longmapsto \mathtt{Race}$, we know

$$\exists W.\,(P,\sigma) \overset{load}{\Longrightarrow} W \wedge W \Longmapsto^{+} \mathtt{Race}$$

Consider the special case shown in Fig.16

$$W = (T, 0, \mathtt{t}_2, \sigma)$$

$$(T, 0, \mathtt{t}_2, \sigma) \underset{\delta}{\overset{\tau}{\Rightarrow}} (T_1, 0, \mathtt{t}_2, \sigma_1)$$

$$(T_1, 0, \mathtt{t}_1, \sigma_1) \underset{\delta_1}{\overset{\tau}{\Rightarrow}} (T_2, 0, \mathtt{t}_1, \sigma_2)$$

$$(T_2, 0, \mathtt{t}_1, \sigma_2) \Longmapsto \mathtt{Race}$$

where race is caused by

$$(T_2(\mathtt{t}_1), \sigma_2) \underset{\delta_2}{\overset{\tau}{\longmapsto}} (\kappa_1, \sigma_2) \quad \wedge \quad (T_2(\mathtt{t}_2), \sigma_2) \underset{\delta_3}{\overset{\tau}{\longmapsto}} (\kappa_2, \sigma_3)$$

and $\neg\delta_2 \smile \delta_3$.

By definition we are able to construct the corresponding non-preemptive state $\hat{W} = (T, \vec{0}, \mathtt{t}_2, \sigma)$ such that

$$(\hat{P}, \sigma) \overset{load}{\Longrightarrow} \hat{W} :\underset{\delta}{\overset{\tau}{\Rightarrow}} (T_1, \vec{0}, \mathtt{t}_2, \sigma_1)$$

, and we have to show $(T_1, \vec{0}, \mathtt{t}_2, \sigma_1) :\Longmapsto \mathtt{Race}$, since we are not able to reach σ_2 in a non-preemptive execution.

We consider two cases:

Case 1: $(T_1, 0, \mathtt{t}_2, \sigma_1) \Longmapsto \mathtt{Race}$
we could derive $(T_1, \vec{0}, \mathtt{t}_2, \sigma_1) :\Longmapsto \mathtt{Race}$ trivially

Case 2: $\neg(T_1, 0, \mathtt{t}_2, \sigma_1) \Longmapsto \mathtt{Race}$
It's trivial to see

$$\mathtt{NPpredict}((T_1, \vec{0}, \mathtt{t}_2, \sigma_1), \mathtt{t}_1, (\delta_1 \cup \delta_2, 0))$$

The difficulty is to predict $(\delta_3, 0)$ for thread $\mathtt{t}_2$ under $(T_1, \vec{0}, \mathtt{t}_2, \sigma_1)$.

Now we could apply the (4)th condition and $\neg(T_1, 0, \mathtt{t}_2, \sigma_1) \Longmapsto \mathtt{Race}$ to prove $\mathtt{NPpredict}((T_1, \vec{0}, \mathtt{t}_2, \sigma_1), \mathtt{t}_2, (\delta_3, 0))$

Let $\delta_0 = \bigcup\{\delta' \mid \exists\kappa', \sigma'.\,(T(\mathtt{t}_2), \sigma_1) \underset{\delta'}{\overset{\tau}{\longmapsto}} (\kappa', \sigma')\}$.

By $\neg(T_1, 0, \mathtt{t}_2, \sigma_1) \Longmapsto \mathtt{Race}$, we know

$\sigma_1 \overset{\delta_0.rs}{=\!=} \sigma_2$ and $\mathsf{locs}(\sigma_1) \cap \delta_0.ws = \mathsf{locs}(\sigma_2) \cap \delta_0.ws$.

Then by the (4)th condition, we have

$\exists\kappa'', \sigma''.\,(T(\mathtt{t}_2), \sigma_1) \underset{\delta_3}{\overset{\tau}{\longmapsto}} (\kappa'', \sigma'')$

[[[Is the condition necessary?]]]

We are able to give a language satisfying (1)-(3) but does not satisfy (4), and in this case $\mathsf{DRF}(\hat{P},\sigma) \implies \mathsf{DRF}(P,\sigma)$ is not provable.

Assume we have this kind of statement "$C_1 \hat{|} C_2$", meaning we non-deterministically choose one statement that is able to progress, and execute.

Assume we have 2 global variables x and y that are initialized to 0.

$$\frac{(C_1, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')}{(C_1 \hat{|} C_2, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')} \text{ Nondet\_1}$$

$$\frac{(C_2, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')}{(C_1 \hat{|} C_2, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')} \text{ Nondet\_2}$$

$$\frac{[\![E]\!]_\sigma = \mathtt{true} \wedge (C, \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')}{(\mathtt{with}(E)\mathtt{do}(C), \sigma) \underset{\delta}{\overset{\tau}{\longmapsto}} (\mathbf{skip}, \sigma')} \text{ Withdo}$$

$\mathtt{t}_1$:	$\mathtt{t}_2$:
$\mathtt{with}(x == 1)\mathtt{do}\ y := 1 \hat{|} \mathtt{nop};$	$x := 1;$
$\mathtt{with}(y == 1)\mathtt{do}\ y := 2 \hat{|} \mathtt{nop};$	$\mathtt{with}(y == 1)\mathtt{do}\ y := 3 \hat{|} \mathtt{nop};$

Figure 17. Wierd counter example

All non-preemptive executions are data-race free (according to our definition of NPDRF), but the preemptive execution would result in data race.

And it is obvious that this language satisfied condition (1)-(3) while failed to satisfy condition (4).

11.2 Alternative Well-Defined Definition

$$\sigma_1 \perp \sigma_2 \overset{\text{def}}{=} \mathsf{locs}(\sigma_1) \cap \mathsf{locs}(\sigma_2) = \emptyset$$

$$\sigma_1 \cdot \sigma_2 \overset{\text{def}}{=} \begin{cases} \{(b, \sigma_1(b) \uplus \sigma_2(b)) \mid b \in \mathsf{dom}(\sigma_1) \cup \mathsf{dom}(\sigma_2)\} & , \sigma_1 \perp \sigma_2 \\ \texttt{Undefined} & , \text{otherwise} \end{cases}$$

Definition 22 (Well-Defined Languages (Alternative))**.** $\mathsf{wd}'(tl)$ iff for any π, F, κ, σ,

(1) [Forward]
[[[Is this necessary? where was it used?]]]

$\forall \kappa', \sigma', \iota, \delta.$, if $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$ then $\mathsf{forward}(\sigma, \sigma')$ and $\mathsf{dom}(\sigma') - \mathsf{dom}(\sigma) \subseteq F$

(2) [Alloc/Free is captured by write set, contents of location outside write set stay unchanged]
[[[Could some part of this property derived from frame?]]]
[[[Why do we need locs captured by δ?]]]

$\forall \kappa', \sigma', \iota, \delta.$ if $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$ then
$(\mathsf{locs}(\sigma') - \mathsf{locs}(\sigma)) \cup (\mathsf{locs}(\sigma) - \mathsf{locs}(\sigma')) \subseteq \delta.ws$ and
$\sigma \overset{\mathsf{locs}(\sigma) - \delta.ws}{=\!=\!=} \sigma'$
(I.e. $\exists \sigma_x, \sigma_y, \sigma'_y.\ \sigma = \sigma_x \cdot \sigma_y \wedge \sigma' = \sigma_x \cdot \sigma'_y$ and $\mathsf{locs}(\sigma_x) = \mathsf{locs}(\sigma) - \delta.ws$)

(3) [Frame Property (instrumented with footprint)]
$\forall \sigma_1, \sigma_2, \kappa', \sigma', \iota, \delta.$ if $\sigma_1 \perp \sigma_2$, $\sigma = \sigma_1 \cdot \sigma_2$, and $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$, then either $(\pi, F) \vdash (\kappa, \sigma_1) \xmapsto[\delta]{\iota}$ **abort** or $\exists \sigma'_1.\ \sigma' = \sigma'_1 \cdot \sigma_2 \wedge (\pi, F) \vdash (\kappa, \sigma_1) \xmapsto[\delta]{\iota} (\kappa', \sigma'_1)$

(4) [Safety Monotonicity (instrumented with footprint and freelist)]
[[[It is different from the original safety monotonicity]]]
[[[for deterministic allocation, we need σ_2 orthogonal to freelist]]]
$\forall \sigma_1, \sigma_2.$ if $\sigma = \sigma_1 \cdot \sigma_2$, $(\mathsf{dom}(\sigma_2) - \mathsf{dom}(\sigma_1)) \cap F = \emptyset$ then
$\forall \iota, \delta, \kappa', \sigma'_1.\ (\pi, F) \vdash (\kappa, \sigma_1) \xmapsto[\delta]{\iota} (\kappa', \sigma'_1)$
$\implies (\pi, F) \vdash (\kappa, \sigma_1 \cdot \sigma_2) \xmapsto[\delta]{\iota} (\kappa', \sigma'_1 \cdot \sigma_2)$

(5) [Footprint captures enough resource for safe execution]
$\forall \sigma_1, \sigma_2, \delta_0.$ if $\neg\Big((\pi, F) \vdash (\kappa, \sigma) \longmapsto \textbf{abort}\Big)$,
$\delta_0 = \bigcup\{\delta \mid \exists \kappa', \sigma', \iota.\ (\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')\}$,
$\sigma = \sigma_1 \cdot \sigma_2$, and $\mathsf{locs}(\sigma_2) \cap \delta_0 = \emptyset$, then
$\neg\Big((\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}\Big)$

(x) if $\iota = \mathsf{call}(\mathtt{f}, \vec{v})$, then $\forall v'.\ \exists \kappa''.\ \mathsf{AftExtn}(\kappa', v') = \kappa''$;
if $\iota = \mathsf{entA} \vee \iota = \mathsf{extA}$, then $\forall v'.\ \mathsf{AftExtn}(\kappa', v') = \kappa'$.

The alternative condition wd' is not able to imply the original wd. But might be sufficient for the proof of DRF equivalence lemmas.

Recall the condition needed for DRF equivalence proof:

Definition 23 (Condition A_0)**.** $A_0(tl)$ if and only if $\forall \sigma_0, \pi_1, F_1, \kappa_1, \iota_1, \delta_1, \sigma_1, \kappa'_1, \pi_2, F_2, \kappa_2.$ if

1. $(\pi_1, F_1) \vdash (\kappa_1, \sigma_0) \xmapsto[\delta_1]{\iota_1} (\kappa'_1, \sigma_1)$, and
2. $\neg\Big((\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \longmapsto \textbf{abort}\Big)$ and
$\forall \iota'_1, \kappa'_2, \sigma'_1, \delta'_1.$
$(\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \xmapsto[\delta'_1]{\iota'_1} (\kappa'_2, \sigma'_1) \implies \delta'_1 \smile \delta_1$
3. $F_1 \cap F_2 = \emptyset$

then $\forall \iota_2, \kappa'_2, \delta_2.$
$\exists \sigma_2.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma_2)$
$\Leftrightarrow \exists \sigma'_2.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_1) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma'_2)$.

Lemma 24. $\forall tl.\ \mathsf{wd}'(tl) \implies A_0(tl)$.

Proof. By assumption (1) of A_0 and condition (2) of $\mathsf{wd}'(tl)$, we have $\sigma_0 \overset{\mathsf{locs}(\sigma_0) - \delta_1.ws}{=\!=\!=} \sigma_1$.

I.e. $\exists \sigma_x, \sigma_y, \sigma'_y.\ \sigma_0 = \sigma_x \cdot \sigma_y$ and $\sigma_1 = \sigma_x \cdot \sigma'_y$ and $\mathsf{locs}(\sigma_y) \subseteq \delta_1.ws$ and $\mathsf{locs}(\sigma'_y) \subseteq \delta_1.ws$ and $\mathsf{dom}(\sigma_y) = \mathsf{dom}(\sigma_x)$

Let $\delta_0 = \bigcup\{\delta' \mid \exists \kappa', \sigma', \iota'.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \xmapsto[\delta]{\iota'} (\kappa', \sigma')\}$

By assumption (2) of A_0,
$\mathsf{locs}(\sigma_y) \cap \delta_0 = \mathsf{locs}(\sigma'_y) \cap \delta_0 \subseteq \delta_1.ws \cap \delta_0 = \emptyset$

By wd' (5), and assumption 2. of $A_0(tl)$,

$$\neg\Big((\pi_2, F_2) \vdash (\kappa_2, \sigma_x) \longmapsto \textbf{abort}\Big)$$

Now we prove one direction ($\Leftarrow$) of the goal, i.e.
$\exists \sigma'_2.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_1) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma'_2)$
$\Rightarrow \exists \sigma_2.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma_2)$.

The assumption could be rewritten as
$(\pi_2, F_2) \vdash (\kappa_2, \sigma_x \cdot \sigma'_y) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma'_2)$.

By frame property, exists σ'_x, $(\pi_2, F_2) \vdash (\kappa_2, \sigma_x) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma'_x)$

By safety monotonicity, (2) of $\mathsf{wd}'(tl)$, and assumption 3. of A_0, $(\pi_2, F_2) \vdash (\kappa_2, \sigma_x \cdot \sigma_y) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma'_x \cdot \sigma_y)$, i.e. $\exists \sigma_2.\ (\pi_2, F_2) \vdash (\kappa_2, \sigma_0) \xmapsto[\delta_2]{\iota_2} (\kappa'_2, \sigma_2)$

□

11.3 Problems with wd'

$\texttt{wd}'$ suffers from these problems

- Prpoerties of language and footprints are mixed up. Condition (1), (3), (4) are intended to describe properties of language, while condition (2), (5) describes properties of δ.
- Condition (3) requires the footprint of two steps are the same, while it is not intended to describe property of δ.
- Similarly, condition (4) requires footprint and resulting state of the two steps are the same. It is different from the original safety monotonicity and it is not clear why such property is necessary.
- While we have read set and write set in footprint, no condition describes property of read set

To deal with these problems, I made the following modifications on $\texttt{wd}'$:

- reorganized the $\texttt{wd}'$ definition, define properties of languages and footprints separately.
- removed requirement of same δ, instead we require read set and write set should be subset of $\mathsf{locs}(\sigma) \cup \llbracket F - \mathsf{dom}(\sigma) \rrbracket$
- removed requirements on result states in condition (4), now (4) is almost the same as original safety monotonicity
- restored condition (3) of wd, describes that the execution actually depends on memory contents in read set and memory domain in write set.

The resulting "well-defined" property are defined separately as "well-defined language" and "well-defined footprint":

$$\begin{aligned}
\texttt{LEqPre}(\sigma_1, \sigma_2, \delta, F) \quad &\overset{\text{def}}{=} \quad \sigma_1 \overset{\delta.rs}{=\!=} \sigma_2 \wedge \\
&\qquad (\mathsf{locs}(\sigma_1) \bigtriangleup \mathsf{locs}(\sigma_2)) \cap \delta.ws = \emptyset \wedge \\
&\qquad (\mathsf{dom}(\sigma_1) \bigtriangleup \mathsf{dom}(\sigma_2)) \cap F = \emptyset
\end{aligned}$$

$$\begin{aligned}
\texttt{LEqPost}(\sigma'_1, \sigma'_2, \delta, F) \quad &\overset{\text{def}}{=} \quad \sigma'_1 \overset{\delta.ws}{=\!=} \sigma'_2 \wedge \\
&\qquad (\mathsf{dom}(\sigma'_1) \bigtriangleup \mathsf{dom}(\sigma'_2)) \cap F = \emptyset
\end{aligned}$$

$$\begin{aligned}
\texttt{LEffect}(\sigma_1, \sigma_2, \delta, F) \quad &\overset{\text{def}}{=} \quad \sigma_1 \overset{\mathsf{locs}(\sigma_1) - \delta.ws}{=\!=\!=\!=} \sigma_2 \wedge \\
&\qquad (\mathsf{locs}(\sigma_1) \bigtriangleup \mathsf{locs}(\sigma_2)) \subseteq \delta.ws \wedge \\
&\qquad (\mathsf{dom}(\sigma_1) \bigtriangleup \mathsf{dom}(\sigma_2)) \subseteq F
\end{aligned}$$

Definition 25 ($\texttt{wd-lang}(tl)$)**.** For any language *tl*, $\texttt{wd-lang}(tl)$ iff for any π, F, κ, σ,

1. *Forward*
   $\forall \kappa', \sigma', \iota, \delta.$, if $(\pi, F) \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma')$ then $\mathsf{forward}(\sigma, \sigma')$
2. *Frame Property*
   $\forall \sigma_1, \sigma_2, \kappa', \sigma', \iota, \delta.$ if $\sigma_1 \perp \sigma_2$, $\sigma = \sigma_1 \cdot \sigma_2$, and $(\pi, F) \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma')$, then either $(\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}$ or $\exists \sigma'_1, \delta'. \sigma' = \sigma'_1 \cdot \sigma_2 \wedge (\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta'}{\longmapsto}} (\kappa', \sigma'_1)$
3. *Safety Monotonicity*
   [[[This condition is redundant for DRF equivalence proof]]]
   $\forall \sigma_1, \sigma_2.$ if $\sigma = \sigma_1 \cdot \sigma_2$, $(\mathsf{dom}(\sigma_2) - \mathsf{dom}(\sigma_1)) \cap F = \emptyset$ then $\neg \Big((\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}\Big) \implies \neg \Big((\pi, F) \vdash (\kappa, \sigma) \longmapsto \textbf{abort}\Big)$

x. if $\iota = \mathsf{call}(\mathtt{f}, \vec{v})$, then $\forall v'. \exists \kappa''. \mathsf{AftExtn}(\kappa', v') = \kappa''$;
   if $\iota = \mathsf{entA} \vee \iota = \mathsf{extA}$, then $\forall v'. \mathsf{AftExtn}(\kappa', v') = \kappa'$.

Definition 26 ($\texttt{wd-fp}(tl)$)**.** For any language *tl*, $\texttt{wd-fp}(tl)$ iff for any π, F, κ,

1. *Footprint should not exceed* $\mathsf{locs}(\sigma)$ *plus available location in* F
   $\forall \sigma_1, \iota, \delta, \kappa', \sigma'_1.$ if $(\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)$ then $\delta.ws \cup \delta.rs \subseteq \mathsf{locs}(\sigma_1) \cup \llbracket F - \mathsf{dom}(\sigma_1) \rrbracket$
2. *Footprint captures enough resource for safe execution*
   $\forall \sigma_1, \sigma_2, \delta_0.$ if $\neg \Big((\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}\Big)$, and
   $\delta_0 = \bigcup \{\delta \mid \exists \kappa', \sigma'_1, \iota. (\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)\}$, and
   $\texttt{LEqPre}(\sigma_1, \sigma_2, \delta_0, F)$, then $\neg \Big((\pi, F) \vdash (\kappa, \sigma_2) \longmapsto \textbf{abort}\Big)$
3. *Footprint captures dependency of one step*
   $\forall \sigma_1, \sigma_2, \delta.$ if $\texttt{LEqPre}(\sigma_1, \sigma_2, \delta, F)$ then for any $\kappa', \iota, \sigma'_1$, $(\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)$ implies
   $\exists \sigma'_2. (\pi, F) \vdash (\kappa, \sigma_2) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_2)$ and $\texttt{LEqPost}(\sigma'_1, \sigma'_2, \delta, F)$
4. *Write set captures the effect of one step*
   $\forall \sigma_1, \iota, \delta, \kappa', \sigma'_1.$ if
   $(\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)$ then
   $\texttt{LEffect}(\sigma_1, \sigma'_1, \delta, F)$

These definitiond are not sufficient to prove condition A_0. Counter example:

$$\frac{\delta.rs \subseteq \mathsf{locs}(\sigma)}{(\_, \_) \vdash (\mathtt{nop}, \sigma) \overset{\tau}{\underset{\delta}{\longmapsto}} (\textbf{skip}, \sigma)} \text{ Bad footprint}$$

Fix1 change *wd-fp* condition 1 to

1. *Footprint has this form of "frame property"*
   $\forall \sigma, \sigma_1, \sigma_2.$ if $\sigma_1 \perp \sigma_2$, $\sigma = \sigma_1 \cdot \sigma_2$, then either $(\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}$ or
   $\bigcup \{\delta \mid \exists \kappa', \sigma', \iota. (\pi, F) \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)\}$
   $\subseteq \bigcup \{\delta \mid \exists \kappa', \sigma'_1, \iota. (\pi, F) \vdash (\kappa, \sigma_1) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma'_1)\}$

   Then $\texttt{wd-lang}$ condition 2 and 3 are redundant.
   The fixed version implies the original wd (Definition 1).

Fix2 add condition

5. *Without resource captured by footprint would abort*
   $\forall \sigma, \sigma_1, \sigma_2, \iota, \delta, \kappa', \sigma'.$ if $\sigma_1 \perp \sigma_2$, $\sigma = \sigma_1 \cdot \sigma_2$, $(\pi, F) \vdash (\kappa, \sigma) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', \sigma')$, and $\delta \cap \mathsf{locs}(\sigma_2) \neq \emptyset$ then
   $(\pi, F) \vdash (\kappa, \sigma_1) \longmapsto \textbf{abort}$

The fixed version implies the original wd (Definition 1).

12. The Footprint-Preserving Compositional Simulation

12.1 Definition of the Module-Local Simulation

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

Definition 27 (Module-Local Downward Simulation)**.**
$(sl, \gamma) \preccurlyeq_{ge,\varphi} (tl, \pi)$ iff
$\mathsf{dom}(sl.\mathsf{InitCore}(\gamma)) = \mathsf{dom}(tl.\mathsf{InitCore}(\pi))$ and for any $\mathtt{f}$, κ, $\Bbbk$, σ, Σ, μ, $\mathbb{F}$, F and $\mathbb{S}$, if $sl.\mathsf{InitCore}(\gamma)(\mathtt{f}) = \Bbbk$, $tl.\mathsf{InitCore}(\pi)(\mathtt{f}) = \kappa$, $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{dom}(\Sigma) = \mathbb{S} \subseteq \mathsf{dom}(\varphi)$, $\mathsf{cl}(\mathbb{S}, \Sigma) = \mathbb{S}$, $\mathsf{locs}(\sigma) = \varphi\langle\!\langle \mathsf{locs}(\Sigma) \rangle\!\rangle$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$, $\mathbb{F} \cap \mathbb{S} = F \cap \varphi\{\!\{\mathbb{S}\}\!\} = \emptyset$ and $\mu = (\mathbb{S}, \varphi\{\!\{\mathbb{S}\}\!\}, \varphi|_{\mathbb{S}})$, then there exists $i \in \mathtt{index}$ such that $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \bullet) \preccurlyeq_{\mu}^{(i,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa, \sigma), \bullet)$.

Here we define $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$ (where the bit β is either $\circ$ or $\bullet$) as the largest relation such that whenever $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$, then

$\mathsf{wf}(\mu, \Sigma)$ iff
$\quad \mathsf{injective}(\mu.f, \Sigma) \wedge (\mu.\mathbb{S} \subseteq \mathsf{dom}(\mu.f)) \wedge (\mu.f\{\!\{\mu.\mathbb{S}\}\!\} \subseteq \mu.S)$

$\mathsf{wf}(\mu, \Sigma, \mathbb{F}, F)$ iff
$\quad \mathsf{wf}(\mu, \Sigma) \wedge (\mu.f\{\!\{\mu.\mathbb{S} \cap \mathbb{F}\}\!\} \subseteq F) \wedge (\mu.f\{\!\{\mu.\mathbb{S} - \mathbb{F}\}\!\} \cap F = \emptyset)$

$\mathsf{FPmatch}(\Delta, \delta, \mu, F)$ iff
$\quad (\delta.rs \cup \delta.ws \subseteq \llbracket F \cup \mu.f\{\!\{\mu.\mathbb{S}\}\!\}\rrbracket) \wedge$
$\quad (\delta.rs \cap \llbracket \mu.f\{\!\{\mu.\mathbb{S}\}\!\}\rrbracket \subseteq \mu.f\langle\!\langle \Delta.rs \rangle\!\rangle) \wedge$
$\quad (\delta.ws \cap \llbracket \mu.f\{\!\{\mu.\mathbb{S}\}\!\}\rrbracket \subseteq \mu.f\langle\!\langle \Delta.ws \rangle\!\rangle)$

$\mathsf{EvolveG}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F)$ iff
$\quad \mathsf{Evolve}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F) \wedge (\mu'.\mathbb{S} - \mu.\mathbb{S} \subseteq \mathbb{F})$

$\mathsf{EvolveR}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F)$ iff
$\quad \mathsf{Evolve}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F) \wedge ((\mu'.\mathbb{S} - \mu.\mathbb{S}) \cap \mathbb{F} = \emptyset)$

$\mathsf{Evolve}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F)$ iff
$\quad \mathsf{wf}(\mu', \Sigma', \mathbb{F}, F) \wedge (\mu.f \subseteq \mu'.f) \wedge \mathsf{Inv}(\mu'.f, \Sigma', \sigma') \wedge$
$\quad \mu'.\mathbb{S} = \mathsf{cl}(\mu.\mathbb{S}, \Sigma') \subseteq \mathsf{dom}(\Sigma') \wedge \mu'.S = \mathsf{cl}(\mu.S, \sigma') \subseteq \mathsf{dom}(\sigma')$

$\mathsf{Inv}(f, \Sigma, \sigma)$ iff
$\quad \forall b, n, b', n'.\ ((b, n) \in \mathsf{locs}(\Sigma)) \wedge (f\langle(b, n)\rangle = (b', n')) \Longrightarrow$
$\quad ((b', n') \in \mathsf{locs}(\sigma)) \wedge (\Sigma(b)(n) \overset{f}{\hookrightarrow} \sigma(b')(n'))$

$v_1 \overset{f}{\hookrightarrow} v_2$ iff
$\quad (v_1 \notin \mathfrak{U}) \wedge (v_1 = v_2) \ \vee\ v_1 \in \mathfrak{U} \wedge v_2 \in \mathfrak{U} \wedge f\langle v_2 \rangle = v_2$

$\mathsf{injective}(f, \Sigma)$ iff
$\quad \forall l_1, l_2, l_1', l_2'.\ l_1 \neq l_2 \wedge l_1 \in \mathsf{locs}(\Sigma) \wedge l_2 \in \mathsf{locs}(\Sigma) \wedge$
$\quad f\langle l_1 \rangle = l_1' \wedge f\langle l_2 \rangle = l_2' \implies l_1' \neq l_2'$

$f\{\!\{\mathbb{S}\}\!\} \overset{\text{def}}{=} \{b' \mid \exists b.\ (b \in \mathbb{S}) \wedge f(b) = (b', \_)\}$

$f\langle l \rangle \overset{\text{def}}{=} (b', n + n')\ ,\quad$ if $l = (b, n) \wedge f(b) = (b', n')$

$f\langle\!\langle ws \rangle\!\rangle \overset{\text{def}}{=} \{l' \mid \exists l.\ (l \in ws) \wedge f\langle l \rangle = l'\}$

$f|_{\mathbb{S}} \overset{\text{def}}{=} \{(b, f(b)) \mid b \in (\mathbb{S} \cap \mathsf{dom}(f))\}$

$\mathsf{cl}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \bigcup_k \mathsf{cl}_k(\mathbb{S}, \Sigma)$, where $\mathsf{cl}_k(\mathbb{S}, \Sigma)$ is inductively defined:
$\quad \mathsf{cl}_0(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \mathbb{S}$
$\quad \mathsf{cl}_{k+1}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \{b' \mid \exists b, n, n'.\ (b \in \mathsf{cl}_k(\mathbb{S}, \Sigma)) \wedge \Sigma(b)(n) = (b', n')\}$

Figure 18. Footprint Matching and Evolution of μ in Our Simulation

$\mathsf{wf}(\mu, \Sigma, \mathbb{F}, F)$, $\mathsf{Inv}(\mu.f, \Sigma, \sigma)$, $\mu.\mathbb{S} = \mathsf{cl}(\mu.\mathbb{S}, \Sigma) \subseteq \mathsf{dom}(\Sigma)$, $\mu.S = \mathsf{cl}(\mu.S, \sigma) \subseteq \mathsf{dom}(\sigma)$ and the following are true:

1. $\forall \mathbb{k}', \Sigma', \Delta.$ if $\beta = \circ$ and $(\gamma, \mathbb{F}) \vdash (\mathbb{k}, \Sigma) \xmapsto[\Delta]{\tau} (\mathbb{k}', \Sigma')$, then one of the following holds:
   (a) there exists j such that
      i. $j < i$, and
      ii. $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j, (\Delta_0 \cup \Delta, \delta_0))} ((\pi, F), (\kappa, \sigma), \circ)$;
   (b) or, there exist κ', σ', δ and j such that
      i. $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{+} (\kappa', \sigma')$, and
      ii. $\mathsf{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta, \mu, F)$, and
      iii. $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j, (\Delta_0 \cup \Delta, \delta_0 \cup \delta))} ((\pi, F), (\kappa', \sigma'), \circ)$;
2. $\forall \mathbb{k}', \iota.$ if $\beta = \circ$, $\iota \neq \tau$ and $(\gamma, \mathbb{F}) \vdash (\mathbb{k}, \Sigma) \xmapsto[\texttt{emp}]{\iota} (\mathbb{k}', \Sigma)$, then there exist κ', δ, σ', κ', μ' and j such that
   (a) $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{*} (\kappa', \sigma')$, and
   $(\pi, F) \vdash (\kappa', \sigma') \xmapsto[\texttt{emp}]{\iota} (\kappa'', \sigma')$, and
   (b) $\Delta_0.rs \cup \Delta_0.ws \subseteq \llbracket \mathbb{F} \cup \mu.\mathbb{S} \rrbracket$ and
   (c) $\mathsf{FPmatch}(\Delta_0, \delta_0 \cup \delta, \mu, F)$, and
   (d) $\mathsf{EvolveG}(\mu, \mu', \Sigma, \sigma', \mathbb{F}, F)$, and
   (e) $((\gamma, \mathbb{F}), (\mathbb{k}', \Sigma), \bullet) \preccurlyeq_{\mu'}^{(j, (\texttt{emp}, \texttt{emp}))} ((\pi, F), (\kappa'', \sigma'), \bullet)$ or $\iota = \texttt{halt}$;
3. $\forall \sigma', \Sigma', \mu'$, if $\beta = \bullet$ and
   (a) $\Sigma \xlongequal{\llbracket \mathbb{F} - \mu.\mathbb{S} \rrbracket} \Sigma'$, $\sigma \xlongequal{\llbracket F \rrbracket - \mu.f\langle\!\langle \mathsf{locs}(\Sigma) \cap \llbracket \mu.\mathbb{S} \rrbracket \rangle\!\rangle} \sigma'$, $\mathsf{forward}(\Sigma, \Sigma')$, $\mathsf{forward}(\sigma, \sigma')$, and
   (b) $\mathsf{EvolveR}(\mu, \mu', \Sigma', \sigma', \mathbb{F}, F)$,
   then there exists j such that
   $((\gamma, \mathbb{F}), (\mathbb{k}, \Sigma'), \circ) \preccurlyeq_{\mu'}^{(j, (\texttt{emp}, \texttt{emp}))} ((\pi, F), (\kappa, \sigma'), \circ)$.

13. The Footprint-Preserving Compositional Simulation (with EFCalls)

14. The Footprint-Preserving Compositional Simulation

In this section, we define a module-local simulation as the correctness obligation of each module's compilation, which is compositional and preserves footprints, allowing us to derive a whole-program simulation that preserves NPDRF. We will discuss compositionality in Sec. 6.2 and postpone the discussions of DRF and NPDRF preservation to Sec. 7.

14.1 Definition of the Module-Local Simulation

As informally explained in Sec. 2, the simulation establishes a consistency relation between executions of the source module γ and the target one π. To achieve compositionality, our simulation is also parameterized with rely/guarantee conditions, specifying the interactions between the current module and its environment at synchronization points. The consistency relation should be preserved under the environment transitions allowed in the rely condition.

Before explaining the simulation definition in Def. 28, we first define some key conditions in Fig. 19. Our simulation is parameterized with μ, defined as follows.

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

Here S and $\mathbb{S}$ specify the memory locations that are currently shared or were once shared. Keeping track of S and $\mathbb{S}$ allows us to define the fixed specifications of rely/guarantee conditions in our simulation. The mapping f maps shared locations (including those once shared) at the source level to shared locations at the target. We require it to be an injective function, i.e., different source locations should be mapped to different target locations.

We require μ be well-formed with respect to the current thread's free spaces $\mathbb{F}$ and F at the source and target levels. As defined in Fig. 19, $\mathsf{wf}(\mu, \mathbb{F}, F)$ says that the injective function f in μ maps shared locations (in $\mathbb{S}$) to shared locations (in S). In particular, a location in $\mathbb{F}$ that is shared (i.e., that has been exported to other threads) must be mapped to a shared location in F. Here $f\{\!\!\{\mathbb{S}\}\!\!\}$ (defined at the bottom of the figure) returns the set of target locations that are mapped from locations in $\mathbb{S}$. $f\{\!\!\{\mathbb{F} \cap \mathbb{S}\}\!\!\}$ is defined similarly.

$\mathsf{FPmatch}(\Delta, \delta, \mu, F)$ relates the footprints Δ and δ at source and target levels. It says, every location accessed at the target (i.e., in δ) must either be from the current thread F, or correspond to a shared location at the source. For the latter case, the location must be accessed at the source (i.e., in Δ).

In our simulation μ may change after related steps at the source and target levels. Fig. 19 defines how μ is allowed to evolve to μ' under steps of the current thread (see EvolveG) and under the environment steps (see EvolveR). First, as defined in Evolve, the new μ' should be well-formed. The mapping f in μ' should relate the new states σ' and Σ' using Inv. Here Inv requires that the locations related by f be contained in the states, and the contents of these locations be also related (see $\overset{f}{\hookrightarrow}$). Besides, S and $\mathbb{S}$ are evolved to reachable closures in the new states σ' and Σ' respectively. We define the closure function $\mathsf{cl}(\mathbb{S}, \Sigma)$ at the bottom of the figure, which is specialized to the CompCert memory model to simplify the presentation. It returns all the locations reachable from $\mathbb{S}$. Since the CompCert memory model allows pointer arithmetics within blocks, we know all the locations in a reachable block (i.e., a block in which some location is reachable) are reachable. The last condition of Evolve says that f in the original μ should be preserved in the new μ' (here function subset is defined by viewing functions as special relations). EvolveG for the current thread's steps requires that the

$$\mathsf{HFPG}(\Delta, (\mu, \mathbb{F})) \ \text{ iff } \ \mathsf{FPG}(\Delta, \mathbb{F}, \mu.\mathbb{S})$$

$$\mathsf{LFPG}(\delta, (\mu, \Delta, F)) \ \text{ iff } \ \mathsf{FPG}(\delta, F, \mu.S) \ \wedge \ \mathsf{FPmatch}(\mu, \Delta, \delta)$$

$$\begin{array}{l} \mathsf{FPmatch}(\mu, \Delta, \delta) \ \text{ iff} \\ \quad (\delta.\mathit{rs} \cap \lfloor\!\lfloor \mu.S \rfloor\!\rfloor \ \subseteq \ \mu.f\langle\!\langle \Delta.\mathit{rs} \rangle\!\rangle) \ \wedge \ (\delta.\mathit{ws} \cap \lfloor\!\lfloor \mu.S \rfloor\!\rfloor \ \subseteq \ \mu.f\langle\!\langle \Delta.\mathit{ws} \rangle\!\rangle) \end{array}$$

$$\mathsf{FPG}(\Delta, \mathbb{F}, \mathbb{S}) \ \text{ iff } \ \mathsf{blocks}(\Delta.\mathit{rs} \cup \Delta.\mathit{ws}) \subseteq \mathbb{F} \cup \mathbb{S}$$

$$\mathsf{HG}((\Delta, \Sigma'), (\mu, \mathbb{F})) \ \text{ iff } \ \mathsf{G}(\Delta, \Sigma', \mathbb{F}, \mu.\mathbb{S})$$

$$\begin{array}{l} \mathsf{LG}((\delta, \sigma'), (\mu, \Delta, \Sigma', F)) \ \text{ iff} \\ \quad \mathsf{G}(\delta, \sigma', F, \mu.S) \ \wedge \ \mathsf{FPmatch}(\mu, \Delta, \delta) \ \wedge \ \mathsf{Inv}(\mu.f, \Sigma', \sigma') \end{array}$$

$$\mathsf{G}(\Delta, \Sigma', \mathbb{F}, \mathbb{S}) \ \text{ iff } \ \mathsf{FPG}(\Delta, \mathbb{F}, \mathbb{S}) \ \wedge \ (\mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma'))$$

$$\begin{array}{l} \mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F)) \ \text{ iff} \\ \quad \mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mu.\mathbb{S}) \wedge \mathsf{R}(\sigma, \sigma', F, \mu.S) \wedge \mathsf{Inv}(\mu.f, \Sigma', \sigma') \end{array}$$

$$\begin{array}{l} \mathsf{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S}) \ \text{ iff} \\ \quad (\Sigma \overset{\lfloor\!\lfloor \mathbb{F} \rfloor\!\rfloor}{=\!=\!=} \Sigma') \ \wedge \ (\mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma')) \wedge \\ \quad \mathsf{forward}(\Sigma, \Sigma') \wedge ((\mathsf{dom}(\Sigma') - \mathsf{dom}(\Sigma)) \cap \mathbb{F} = \emptyset) \end{array}$$

$$\begin{array}{l} \mathsf{Inv}(f, \Sigma, \sigma) \ \text{ iff} \\ \quad \forall b, n, b', n'.\, ((b, n) \in \mathsf{locs}(\Sigma)) \wedge (f\langle (b, n) \rangle = (b', n')) \Longrightarrow \\ \quad ((b', n') \in \mathsf{locs}(\sigma)) \wedge (\Sigma(b)(n) \overset{f}{\hookrightarrow} \sigma(b')(n')) \end{array}$$

$$\begin{array}{l} v_1 \overset{f}{\hookrightarrow} v_2 \ \text{ iff} \\ \quad (v_1 \notin \mathfrak{U}) \wedge (v_1 = v_2) \ \vee \ v_1 \in \mathfrak{U} \wedge v_2 \in \mathfrak{U} \wedge f\langle v_1 \rangle = v_2 \end{array}$$

$$\begin{array}{l} \vec{v} \overset{f}{\hookrightarrow} \vec{v}' \ \text{ iff } \exists v_1, v'_1, \ldots, v_n, v'_n. \\ \quad \vec{v} = \{v_1, \ldots, v_n\} \wedge \vec{v}' = \{v'_1, \ldots, v'_n\} \wedge \forall i.\ v_i \overset{f}{\hookrightarrow} v'_i \end{array}$$

$$\begin{array}{l} \iota \overset{f}{\hookrightarrow} \iota' \ \text{ iff} \\ \quad \iota = \iota' = e \vee \iota = \iota' = \mathsf{entA} \vee \iota = \iota' = \mathsf{extA} \vee \\ \quad \exists \mathtt{f}, \vec{v}, \vec{v}'.\ \iota = \mathsf{call}(\mathtt{f}, \vec{v}) \wedge \iota' = \mathsf{call}(\mathtt{f}, \vec{v}') \wedge \vec{v} \overset{f}{\hookrightarrow} \vec{v}' \vee \\ \quad \exists v, v'.\ \iota = \mathsf{ret}(v) \wedge \iota' = \mathsf{ret}(v') \wedge v \overset{f}{\hookrightarrow} v' \end{array}$$

$$\begin{array}{l} \mathsf{Gargs}(\iota, \mathbb{S}) \ \text{ iff} \\ \quad \iota = e \vee \iota = \mathsf{entA} \vee \iota = \mathsf{extA} \vee \\ \quad \exists \mathtt{f}, \vec{v}.\ \iota = \mathsf{call}(\mathtt{f}, \vec{v}) \wedge (\forall b, n.\ (b, n) \in \vec{v} \implies b \in \mathbb{S}) \vee \\ \quad \exists v, .\ \iota = \mathsf{ret}(v) \wedge (\forall b, n.\ v = (b, n) \implies b \in \mathbb{S}) \end{array}$$

$$\begin{array}{l} \mathsf{injective}(f) \ \text{ iff} \\ \quad \forall b_1, b_2, b'_1, b'_2.\, b_1 \neq b_2 \wedge f(b_1) = b'_1 \wedge f(b_2) = b'_2 \implies b'_1 \neq b'_2 \end{array}$$

$$f\{\!\!\{\mathbb{S}\}\!\!\} \overset{\text{def}}{=} \{b' \mid \exists b.\, (b \in \mathbb{S}) \wedge f(b) = b'\}$$

$$f\langle l \rangle \overset{\text{def}}{=} (b', n)\ , \quad \text{if } l = (b, n) \wedge f(b) = b'$$

$$f\langle\!\langle \mathit{ws} \rangle\!\rangle \overset{\text{def}}{=} \{l' \mid \exists l.\, (l \in \mathit{ws}) \wedge f\langle l \rangle = l'\}$$

$$f|_{\mathbb{S}} \overset{\text{def}}{=} \{(b, f(b)) \mid b \in (\mathbb{S} \cap \mathsf{dom}(f))\}$$

$$f_2 \circ f_1 \overset{\text{def}}{=} \{(b_1, b_3) \mid \exists b_2.\, f_1(b_1) = b_2 \wedge f_2(b_2) = b_3\}$$

$$\begin{array}{l} \mathsf{cl}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \bigcup_k \mathsf{cl}_k(\mathbb{S}, \Sigma)\ , \ \text{ where } \mathsf{cl}_k(\mathbb{S}, \Sigma) \text{ is inductively defined:} \\ \quad \mathsf{cl}_0(\mathbb{S}, \Sigma) \quad \overset{\text{def}}{=} \mathbb{S} \\ \quad \mathsf{cl}_{k+1}(\mathbb{S}, \Sigma) \overset{\text{def}}{=} \{b' \mid \exists b, n, n'.\, (b \in \mathsf{cl}_k(\mathbb{S}, \Sigma)) \wedge \Sigma(b)(n) = (b', n')\} \end{array}$$

Figure 19. Footprint Matching and Rely/Guarantee Conditions in Our Simulation

additional locations in $\mu'.f$ be allocated from the current thread's $\mathbb{F}$, while $\mathsf{EvolveR}$ for the environment steps says the opposite.

Definition 28 (Module-Local Downward Simulation)**.**
$(sl, \gamma) \preccurlyeq_{ge,\varphi} (tl, \pi)$ iff
$\mathsf{dom}(sl.\mathsf{InitCore}(\gamma)) = \mathsf{dom}(tl.\mathsf{InitCore}(\pi))$ and
for any $\mathtt{f}$, κ, $\Bbbk$, σ, Σ, μ, $\mathbb{F}$, F, $\mathbb{S}$ and S, if $sl.\mathsf{InitCore}(\gamma)(\mathtt{f}) = \Bbbk$, $tl.\mathsf{InitCore}(\pi)(\mathtt{f}) = \kappa$, $ge \subseteq \mathsf{locs}(\Sigma)$, $\mathsf{dom}(\Sigma) = \mathbb{S} = \mathsf{cl}(\mathbb{S}, \Sigma)$, $\mathsf{dom}(\sigma) = S = \varphi\{\!\{\mathbb{S}\}\!\}$, $\mathsf{locs}(\sigma) = \varphi\langle\!\langle \mathsf{locs}(\Sigma)\rangle\!\rangle$, $\mathsf{Inv}(\varphi, \Sigma, \sigma)$, $\mathbb{F} \cap \mathbb{S} = F \cap S = \emptyset$ and $\mu = (\mathbb{S}, S, \varphi|_{\mathbb{S}})$, then there exists $i \in \mathtt{index}$ such that $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \bullet) \preccurlyeq_{\mu}^{(i,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa, \sigma), \bullet)$.

Here we define $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$ (where the bit β is either $\circ$ or $\bullet$) as the largest relation such that whenever $((\gamma, \mathbb{F}), (\Bbbk, \Sigma), \beta) \preccurlyeq_{\mu}^{(i,(\Delta_0,\delta_0))} ((\pi, F), (\kappa, \sigma), \beta)$, then the following are true:

1. $\forall \Bbbk', \Sigma', \Delta.$ if $\beta = \circ$, $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \xmapsto[\Delta]{\tau} (\Bbbk', \Sigma')$ and $\mathsf{HFPG}(\Delta_0 \cup \Delta, (\mu, \mathbb{F}))$, then one of the following holds:
   (a) there exists j such that
      i. $j < i$, and
      ii. $((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0))} ((\pi, F), (\kappa, \sigma), \circ)$;
   (b) or, there exist κ', σ', δ and j such that
      i. $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{+} (\kappa', \sigma')$, and
      ii. $\mathsf{LFPG}(\delta_0 \cup \delta, (\mu, \Delta_0 \cup \Delta, F))$, and
      iii. $((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\Delta_0 \cup \Delta,\delta_0 \cup \delta))} ((\pi, F), (\kappa', \sigma'), \circ)$;
2. $\forall \Bbbk', \iota.$ if $\beta = \circ$, $\iota \neq \tau$, $(\gamma, \mathbb{F}) \vdash (\Bbbk, \Sigma) \xmapsto[\mathtt{emp}]{\iota} (\Bbbk', \Sigma)$ and $\mathsf{HG}((\Delta_0, \Sigma), (\mu, \mathbb{F}))$ and $\mathsf{Gargs}(\iota, \mu.\mathbb{S})$, then there exist κ', δ, ι', σ', κ'' and j such that
   (a) $(\pi, F) \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau}{}^{*} (\kappa', \sigma')$, and
   $(\pi, F) \vdash (\kappa', \sigma') \xmapsto[\mathtt{emp}]{\iota'} (\kappa'', \sigma')$, and
   (b) $\mathsf{LG}((\delta_0 \cup \delta, \sigma'), (\mu, \Delta_0, \Sigma, F))$, and $\iota \overset{\mu.f}{\hookrightarrow} \iota'$, and
   (c) $((\gamma, \mathbb{F}), (\Bbbk', \Sigma), \bullet) \preccurlyeq_{\mu}^{(j,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa'', \sigma'), \bullet)$
   or $\iota = \mathsf{ret}(v)$;
3. $\forall \sigma', \Sigma', v_s, \Bbbk', v_t,$ if $\beta = \bullet$ and $\mathsf{AftExtn}(\Bbbk, v_s) = \Bbbk'$ and $v_s \overset{\mu.f}{\hookrightarrow} v_t$ $\mathsf{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F))$,
   then there exists κ', j such that
   (a) $\mathsf{AftExtn}(\kappa, v_t) = \kappa'$, and
   (b) $((\gamma, \mathbb{F}), (\Bbbk', \Sigma'), \circ) \preccurlyeq_{\mu}^{(j,(\mathtt{emp},\mathtt{emp}))} ((\pi, F), (\kappa', \sigma'), \circ)$.

The simulation $(sl, \gamma) \preccurlyeq (tl, \pi)$ relates the executions of the source module γ and the target module π. We first use the languages' $\mathsf{InitCore}$ functions (see Fig. 4) to initialize the "core" states $\Bbbk$ and κ. We assume their executions start from the same initial memory states at the source and target levels, and the whole memory is shared between the current thread and its environment. We define μ with the injective function $\mathsf{id}_{\mathbb{S}}$, which is the identity function on $\mathbb{S}$. That is, $\mathsf{dom}(\mathsf{id}_{\mathbb{S}}) = \mathbb{S}$ and $\forall l \in \mathbb{S}.\ \mathsf{id}_{\mathbb{S}}(l) = l$.

The index i and the history footprints (Δ_0, δ_0) are introduced to ensure footprint preservation. Since compilations may reorder instructions, it may be more natural to relate the footprints of *multiple* source steps to the corresponding target footprints. Our simulation allows one to accumulate the footprints using the history footprints (Δ_0, δ_0), and check $\mathsf{FPmatch}$ (the footprint matching condition, defined in Fig. 19) later. Note that one may choose to check $\mathsf{FPmatch}$ at every source step. Whether to accumulate footprints or check $\mathsf{FPmatch}$ depends on the compilation applications, and we leave the choices to verifiers. However, for infinite executions, it is not allowed to continuously accumulate the footprints and never check $\mathsf{FPmatch}$. Therefore our simulation is parameterized with a metric i from a well-founded set $\mathtt{index}$. The metric should decrease at steps that accumulate the footprints and could be reset at steps that check $\mathsf{FPmatch}$.

We also introduce a bit β to indicate the synchronization points (i.e., the program points when the control may switch to the environment). It takes two values $\bullet$ and $\circ$. Initially it is $\bullet$, indicating a possible switch that allows the environment threads to make steps before the current thread starts. Internal τ-steps of the current thread keeps β to be $\circ$ (see condition 1 in Def. 28). When the current thread makes a transition labeled with e, entA or extA (see condition 2 in Def. 28), we set β from $\circ$ to $\bullet$. The environment can interfere with the current thread when β is $\bullet$ (see condition 3 in Def. 28). After the environment interference, β should be reset to $\circ$.

Our simulation definition follows the diagram in Fig. 2(b). Every source τ-step should correspond to zero-or-multiple target τ-steps (see condition 1 in Def. 28). One may check $\mathsf{FPmatch}$ for footprints accumulated till the current steps and reset the metric (see 1(a)), or choose to continue accumulating the footprints and decrease the metric (see 1(b)). To simplify the presentation, [[[??]]] the footprints should be accumulated when the source step correspond to zero target steps.

At switch points (see condition 2 in Def. 28), we check $\mathsf{FPmatch}$ for the accumulated footprints and reset the metric. We also evolve μ to μ', which satisfies $\mathsf{EvolveG}$ (defined in Fig. 19).

We require the simulation relation is preserved after the environment interference (see condition 3 in Def. 28). The environment steps should not change the current thread's local memory. The condition 3(a) disallow the environment threads to update locations in the current thread's $\mathbb{F}$ (or F at the target level) except in the shared parts $\mathbb{S}$ (or corresponds to $\mathbb{S}$). Here $\doteq$ is defined at the bottom of Fig. 4. After the environment step, μ is evolved to μ' satisfying $\mathsf{EvolveR}$ (defined in Fig. 19).