Towards Certified Separate Compilation for Concurrent Programs

Extended Version

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

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

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

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

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

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

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

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

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

CompCert [16], the most well-known certified realistic compiler, establishes the semantics preservation property for compilation of sequential Clight programs, but with no explicit support of separate compilation. To support general separate compilation, Stewart et al. [29] develop Compositional CompCert, which allows the modules to call each other’s external functions. Like CompCert, Compositional CompCert only supports sequential programs too.

Stewart et al. [29] do argue that Compositional CompCert may be applied for data-race-free (DRF) concurrent programs, since they behave the same in the standard interleaving semantics as in certain non-preemptive semantics where switches between threads occur only at certain designated program points. The sequential compilation is sound as long as the switch points are viewed as external function calls so that optimizations do not go beyond them.

Although the argument is plausible, there are still significant challenges to actually implement it. We need proper formulation of the non-preemptive semantics and the notion of DRF. Then we need to indeed prove that DRF programs have the same behaviors (including termination) in the interleaving semantics as in the non-preemptive semantics, and verify that the compilation preserves DRF. More importantly, the formulation and the proofs should be done in a compositional and language-independent way, to allow separate compilation where the modules can be implemented in different languages. Reusing the proofs of CompCert is challenging too, as memory allocation for multiple threads may require a different memory model from that of CompCert, and the non-deterministic semantics also makes it difficult to reuse the downward simulation in CompCert.

In addition, the requirement of DRF could be sometimes overly restrictive. Although Boehm [3] points out there are essentially no “benign” races in source programs, and languages like C/C++ essentially give no semantics to racy programs [4], it is still meaningful to allow benign races in

∗The first two authors contributed equally and are listed alphabetically.
†Corresponding author.
‡Also with State Key Laboratory for Novel Software Technology.
We also extend the framework to allow based on previous work on certified compilation, but makes
itives like spin-locks in Linux can be supported. Our work is
itives that bridge the gap between compilation for
as separate compilation of DRF programs to x86 assembly.
We apply our framework to verify the correctness of CompCert
for separate compilation of DRF programs to x86 assembly. We also extend the framework to allow confined benign races in x86-TSO (confined in that the racy code must execute in a separate region of memory and have race-free abstraction), so that optimized implementations of synchronization primitives like spin-locks in Linux can be supported. Our work is based on previous work on certified compilation, but makes the following new contributions:

- We design a compositional footprint-preserving simulation as the correctness formulation of separate compilation for sequential programs (Sec. 4). It considers module interactions at both external function calls and synchronization points, thus is compositional with respect to both module linking and non-preemptive concurrency. It also requires the footprints (i.e., the set of memory locations accessed during the execution) of the source and target modules to be related. This way we effectively reduce the proof of the compiler’s preservation of DRF, a whole program property, to the proof of local footprint preservation.

- We work with an abstract programming language (Sec. 3), which is not tied to any specific synchronization constructs such as locks but uses abstract labels to model how such constructs interact with other modules. It also abstracts away the concrete primitives that access memory. We introduce the notion of well-defined languages to enforce a set of constraints over the state transitions and the related footprints, which actually give an extensional interpretation of footprints. These constraints are satisfied by various real languages such as Clight and x86 assembly. With the abstract language, we study the equivalence between preemptive and non-preemptive semantics (Sec. 3.3), the equivalence between DRF and NPDRF (the notion of race-freedom defined in the non-preemptive setting, shown in Sec. 5), and the properties of our new simulation. As a result, the lemmas in our proof framework are re-usable when instantiating to real languages.

- We prove a strengthened DRF-guarantee theorem for the x86-TSO model (Sec. 7.3). It allows an x86-TSO program to call a (x86-TSO) module with benign races. If one can replace the racy module with a more abstract version so that the resulting program is DRF in the sequentially consistent (SC) semantics, then the original racy x86-TSO program would behave the same with the more abstract program in the SC semantics. This way we allow the target programs to have confined benign races in the relaxed x86-TSO model. We use this approach to support efficient x86-TSO implementations of locks.

- Putting all these together, our framework (see Fig. 2 and Fig. 3) successfully builds certified separate compilation for concurrent programs from sequential compilation. It highlights the importance of DRF preservation for correct compilation. It also shows a possible way to adapt the existing work of CompCert and Compositional CompCert to interleaving concurrency.

- As an instantiation of our language-independent compilation verification framework, we develop the certified compiler CASCompCert1 (Sec. 7). We instantiate the source and target languages as Clight and x86 assembly. We also provide an efficient x86-TSO implementation of locks as a synchronization library. CASCompCert reuses the compilation passes of CompCert-3.0.1 [6] (including all the translation passes and four optimization passes). We prove that they satisfy our new compilation correctness criterion, where we reuse a considerable amount of the original CompCert proofs, with minor adjustment for footprint-preservation. The proofs for each pass take less than one person week on average.

This technical report (TR) is an extended version of our PLDI’19 paper. The Coq development and the paper are publicly available at https://plax-lab.github.io/publications/ccc/.

2 Informal Development

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

2.1 CompCert

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

Correct(Comp) \iff \forall S, C. \ Comp(S) = C \implies S \approx C.

The semantics preservation S ≈ C requires S and C have the same sets of observable event traces:

S \approx C \iff \forall B. Etr(S, B) \iff Etr(C, B).

Here we write Etr(S, B) to mean that an execution of S produces the observable event trace B, and likewise for C.

To verify S ≈ C, CompCert relies on the deterministic of the target language (written as Det(C) in Fig. 1(a)) and proves only the downward direction S ⊆ C, i.e., S refines C:

1 It is short for an extended CompCert with the support of Concurrency, Abstraction and Separate compilation.
Towards Certified Separate Compilation for Concurrent Programs

\[
S \approx C \quad S_1 \circ \ldots \circ S_n \approx C_1 \circ \ldots \circ C_n
\]

\[
\uparrow \quad \text{Det}(C) \quad S_1 \circ \ldots \circ S_n \subseteq C_1 \circ \ldots \circ C_n
\]

\[
\uparrow \quad S_1 \circ \ldots \circ S_n \not\subseteq C_1 \circ \ldots \circ C_n
\]

\[
S \not\sim C \quad \forall R, G \vdash S_1 \not\sim' C_i
\]

(a) CompCert \hspace{1cm} (b) Compositional CompCert

\[
S \rightarrow S' \quad S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4
\]

\[
G \not\sim' R \quad G \not\sim' C
\]

\[
C \rightarrow C' \quad C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4
\]

(c) \hspace{1cm} (d) R, G \vdash S \not\sim' C

Figure 1. Proof structures of certified compilation

\[ S \subseteq C \text{ iff } \forall B. Etr(S, B) = Etr(C, B). \]

The deterministic \( \text{Det}(C) \) ensures that \( C \) admits only one event trace, so we can derive the upward refinement \( S \subseteq C \) from \( S \subseteq C \). The latter is then proved by constructing a (downward) simulation relation \( S \not\sim C \), depicted in Fig. 1(c). However, the simulation \( \not\sim \) relates whole programs only and it does not take into account the interactions with other modules which may update the shared resource. So it is not compositional and does not support separate compilation.

2.2 Compositional CompCert

Compositional CompCert supports separate compilation by re-defining the simulation relation for modules. Figure 1(b) shows its proof structure. Suppose we make separate compilation \( \text{Comp}_1, \ldots, \text{Comp}_n \). Each \( \text{Comp} \) transforms a source module \( S_i \) to a target module \( C_i \). The overall compilation is correct if, when linked together, the target modules \( C_1 \circ \ldots \circ C_n \) preserve the semantics of the source modules \( S_1 \circ \ldots \circ S_n \) (here we write \( \circ \) as the module-linking operator). For example, the following program consists of two modules. The function \( f \) in module \( S_1 \) calls the external function \( g \). The external module \( S_2 \) may access the variable \( b \) in \( S_1 \).

```c
// Module S1
extern void g(int *x); // Module S2
int f(){
    int g(int *x){(2.1)
        int a = 0, b = 0;
        *x = 3;
        g(&b);
    }
    return a + b;
}
```

Suppose the two modules \( S_1 \) and \( S_2 \) are independently compiled to the target modules \( C_1 \) and \( C_2 \). The correctness of the overall compilation requires \( (S_1 \circ S_2) \approx (C_1 \circ C_2) \).

With the deterministic of the target modules, we only need to prove the downward refinement \( (S_1 \circ S_2) \subseteq (C_1 \circ C_2) \), which is reduced to proving \( (S_1 \circ S_2) \not\subseteq (C_1 \circ C_2) \), just as in CompCert. Ideally we hope to know \( (S_1 \circ S_2) \not\subseteq (C_1 \circ C_2) \) from \( S_1 \not\subseteq C_1 \) and \( S_2 \not\subseteq C_2 \), and ensure the latter two by correctness of \( \text{Comp}_1, \ldots, \text{Comp}_n \). However, the CompCert simulation \( \not\sim \) is not compositional.

To achieve compositionality, Compositional CompCert defines the simulation relation \( \not\sim' \) shown in Fig 1(d). It is parameterized with the interactions between modules, formulated as the rely/guarantee conditions \( R \) and \( G \) \[14\]. The rely condition \( R \) of a module specifies the general callee behaviors happen at the external function calls of the current module (shown as the thick arrows in Fig. 1(d)). The guarantee \( G \) specifies the possible transitions made by the module itself (the thin arrows). The simulation is compositional as long as the \( R \) and \( G \) of linked modules are compatible.

Compositional CompCert proves that the CompCert compilation passes satisfy the new simulation \( \not\sim' \). The intuition is that the compiler optimizations do not go beyond external calls (unless only local variables get involved). That is, for the example (2.1), the compiler cannot do optimizations based on the (wrong) assumption that \( b \) is 0 when \( f \) returns.

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

\[
\begin{align*}
1 \quad & r1 = 1; \\
2 \quad & r2 = r2 + 1; \\
3 \quad & \text{lock}(); \\
4 \quad & x = 1; \\
5 \quad & y = x + 1; \\
6 \quad & \text{unlock}(); \\
\end{align*}
\]

Intuitively, we can use Compositional CompCert to compile the program, where the code segment between two consecutive switch points is compiled as sequential code. By viewing the switch points as special external function calls, the simulation \( \not\sim' \) can be applied to relate the non-preemptive executions of the source and target modules.

2.3 Challenges and Our Approaches

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

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

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

Besides, we also allow module-local steps to generate messages EntAtom and ExtAtom to indicate the boundary of the atomic operations inside the module. The concrete commands that generate these messages are unspecified, which can be different in different modules.

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

A: We decide to use a different memory model. We reserve separate address spaces $F$ for memory allocation in different threads (see Sec. 3). Therefore allocation of one thread would not affect behaviors of others. This greatly simplifies the semantics and the proofs, but also makes it almost impossible to reuse CompCert proofs (see Sec. 7.2). We address this problem by establishing some semantics equivalence between our memory model and the CompCert memory model (shown in Sec. 7.2).

Q: How to compositionally prove DRF-preservation? The simulation $\preceq$ in Compositional CompCert cannot ensure DRF-preservation. As we have explained, DRF is a whole-program property, and so is DRF-preservation. To support separate compilation, we need to reduce DRF-preservation to some thread-local property.

A: We propose a new compositional simulation $\preceq$ (see Sec. 4). Based on the simulation in Fig. 1(d), we additionally require footprint consistency saying that the target $C$ should have the same or smaller footprints than the source $S$ during related transitions. For instance, when compiling lines 4–5 of the left thread in (2.2), the target is only allowed to read $x$ and write to $x$ and $y$. Note that we check footprint consistency at switch points only. This way we allow compiler optimizations as long as they do not go beyond the switch points. For lines 4–5 of the left thread in (2.2), the target could be $y=2; x=1$ where the writes to $x$ and $y$ are swapped.

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

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

Q: How to support benign races and relaxed machine models? It is difficult to write useful DRF programs within sequential languages (e.g., CompCert Clight) because they do not provide synchronization primitives (e.g., locks). Nevertheless, we can implement synchronization primitives as external modules so that the threads written in the sequential languages can call functions (e.g., lock-acquire and lock-release) in the external modules. In practice, the efficient implementation $\pi_o$ of the synchronization primitives may be written directly in assembly languages, and may introduce benign races when used by multiple threads simultaneously (e.g., see Fig. 10 for such an efficient implementation of spin locks). We may view that there is a separate compiler transforming the abstract primitives $y_o$ to their implementations $\pi_o$. But how do we prove the compilation correctness in the case when the target code may contain races?

In addition, our previous discussions all assumed that the source and target programs have sequentially consistent (SC) behaviors. But real-world target machines commonly use relaxed semantics. Although most relaxed models have DRF guarantees saying that DRF programs have SC behaviors in
Towards Certified Separate Compilation for Concurrent Programs

the relaxed semantics, there are no such guarantees for racy programs. It is unclear whether the compilation is still correct when the target code (which may have benign races due to the use of efficient implementations of synchronization primitives) uses relaxed semantics.

A: Although the target assembly program using πo may contain races, we do know that the assembly program using Yo is DRF if the source program using Yo is DRF and the compilation is DRF-preserving. We propose a compositional simulation πo ≈o Yo, which ensures a strengthened DRF-guarantee theorem for the x86-TSO model. It says, the x86-TSO program using πo behaves the same as the program using Yo in the SC semantics if the latter is DRF.

2.4 Frameworks and Key Semantics Components

Figure 2 shows the semantics components and proof steps in our basic framework for DRF and SC target code. The goal of our compilation correctness proof is to show the semantics preservation (i.e., S1 || ... || Sn ≡ C1 || ... || Cn at the top of the figure). This follows the correctness of separate compilation of each module, formulated as R, G ⊨ S1 ≈ C1 (the bottom left), which is our new footprint-preserving module-local simulation. We do the proofs in the following steps. Double arrows in the figure are logical implications.

First, we restrict the compilation to source preemptive code that is race-free (i.e., DRF(S1 || ... || Sn) at the right bottom of the figure). Then from the equivalence between preemptive and non-preemptive semantics, we derive 3, the equivalence between S1 || ... || Sn and S1 * ... * Sn, the latter representing non-preemptive execution of the threads. Similarly, if we have DRF(C1 || ... || Cn) (at the top right), we can derive 2. With 1 and 2, we can derive the semantics preservation ≡ between preemptive programs from ≡ between their non-preemptive counterparts.

Second, DRF of the target programs is obtained through the path 3, 7, and 8. We define a notion of DRF for non-preemptive programs (called NPDRF), making it equivalent to DRF, from which we derive 3 and 7. To know NPDRF(C1 || ... || Cn) from NPDRF(S1 || ... || Sn), we need a DRF-preserving simulation ≃ between non-preemptive programs (see 7).

Third, by composing our footprint-preserving local simulation, the DRF-preserving simulation ≃ for non-preemptive whole programs can be derived (step 8). Given the downward whole-program simulation, we flip it to get an upward one (step 6), with the extra premise that the local execution in each target module is deterministic. Using the simulation in both directions we derive the equivalence (step 3).

Figure 3 shows our ideas for adapting the results of Fig. 2 to relaxed target models and to allow the target programs to contain confined benign races. The goal of the compilation correctness proof is still to show the refinement P ⊑ pnm. Here the source P contains a library module Yo of the abstract synchronization primitives, as well as normal client modules γ1, . . . , γm. Threads in P can call functions in these modules. In the target code pnm, each client module γi is compiled to πi using a sequential compiler SeqCompγi. The library module Yo is implemented as πo, which may contain benign races. The superscript pnm indicates the use of relaxed memory models.

To prove the refinement, we first apply the results of Fig. 2 and prove P ⊑ psc assuming DRF(P) and correctness of each SeqCompγi (step 1 in Fig. 3). Here the superscript sc means the use of SC assembly semantics. Each client module πi in psc has the same code as in pnm but uses the SC semantics. Note that psc still uses the abstract library Yo rather than its racy implementation πo. We can view that at this step the library is compiled by an identity transformation.

Figure 2 also shows DRF preservation, so we know DRF(psc) given DRF(P) (step 2 in Fig. 3). The last step 3 is our strengthened DRF-guarantee theorem. We require a compositional simulation relation πo pnm ≃o Yo holds, saying that πo in the relaxed semantics indeed implements Yo.

The approach not only supports racy implementations of synchronization primitives, but also applies in more general cases when πo is a racy implementation of a general concurrent object such as a stack or a queue. For instance, πo could be the Treiber stack implementation [30], and then Yo could...
Figure 4. The abstract concurrent language

be an atomic abstract stack. Then $\pi_n^{\ell m} \preceq^n y_0$ can be viewed as a correctness criterion of the object implementation in a relaxed model. It ensures a kind of "contextual refinement", i.e., any client threads using $\pi_n$ (in relaxed semantics) generate no more observable behaviors than using $y_0$ instead (in SC semantics), as long as the clients using $y_0$ is DRF.

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

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

3 The Language and Semantics

3.1 The Abstract Language

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

The abstract module language $tl$ is defined as a tuple $(Module, Core, InitCore, \rightarrow)$, whose components can be instantiated for different concrete languages. $Module$ describes the syntax of programs. As in Compositional CompCert [29], $Core$ is the set of internal "core" states $\kappa$, such as control continuations or register files. The function $InitCore$ returns an initial "core" state $\kappa$ whenever a thread is created or an external function call is made. In this paper we mainly focus on threads as different modules, and omit the external calls to simplify the presentation. We do support external calls in our Coq implementation in the same way as in Compositional CompCert. The labelled transition "$\rightarrow^*$" models the local execution of a module, which we explain below. We use $P(S)$ to represent the powerset of the set $S$.

**Module-local semantics.** The local execution step inside a module is in the form of $F \vdash (\kappa, \sigma) \rightarrow^* (\kappa', \sigma')$. The *global* memory state $\sigma$ is a finite partial mapping from memory addresses to values. Each step is also labeled with a message $r$ and a footprint $\delta$.

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

The messages $r$ contain information about the module-local steps. Here we only consider externally observable events $e$ (such as outputs), termination of threads ($\text{ret}$), and the beginning and the end of atomic blocks ($\text{EntAtom}$ and $\text{ExtAtom}$). Any other steps are silent, labeled with a special symbol $\tau$. The label $\tau$ is often omitted for cleaner presentation. Atomic blocks are the language constructs to ensure sequential execution of code blocks inside them. They can be implemented differently in different module languages. The messages define the protocols of communications with the global whole-program semantics (presented below).
Towards Certified Separate Compilation for Concurrent Programs

The footprint $\delta$ is a pair $\langle rs, ws \rangle$ of the read set and write set of memory locations in this step.\(^5\) We write $\mathit{emp}$ for the footprint where both sets are empty.

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

**Well-defined languages.** Although the abstract module language $\mathcal{tl}$ can be instantiated with different real languages, the instantiation needs to satisfy certain basic requirements. We formulate these requirements in Def. 1. It gives us an extensional interpretation of footprints.

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

1. $\mathit{forward}(\sigma, \sigma')$;
2. $\mathit{LEffect}(\sigma, \sigma', \delta, F)$;
3. For any $\sigma_1$, if $\mathit{LEqPre}(\sigma, \sigma_1, \delta, F)$, then there exists $\sigma'_1$ such that $F \vdash (\kappa, \sigma_1) \xrightarrow{\delta} \langle \kappa', \sigma'_1 \rangle$ and $\mathit{LEqPost}(\sigma', \sigma'_1, \delta, F)$.
4. Let $\delta_0 = \langle \delta \mid \exists \kappa', \sigma. F \vdash (\kappa, \sigma) \xrightarrow{\delta} \langle \kappa', \sigma' \rangle \rangle$. For any $\sigma_1$, if $\mathit{LEqPre}(\sigma, \sigma_1, \delta_0, F)$, then for any $\kappa'_1, \sigma'_1, \delta_1$, $F \vdash (\kappa, \sigma_1) \xrightarrow{\delta_1} \langle \kappa'_1, \sigma'_1 \rangle \implies \exists \sigma'_1. F \vdash (\kappa, \sigma) \xrightarrow{\delta_1} \langle \kappa'_1, \sigma'_1 \rangle$.

It requires that a step may enlarge the memory domain but cannot reduce it (Item (1)), and the additional memory should be allocated from $F$ and included in the write set (Item (2)). Item (2) also requires that the memory out of the write set should keep unchanged. Item (3) says that the memory updates and allocation only depend on the memory content in the read set, the availability of the memory cells in the write set, and the set of memory locations already allocated from $F$. Item (4) requires that the non-determinism of each step is not affected by memory contents outside of all the possible read sets in $\delta_0$. Here we do not relate $\sigma'_1$ and $\sigma'$, which can be derived from Item (3).

We have proved in Coq that some real languages satisfy $\mathrm{wd}$, including Cght, Cminor, and x86 assembly [13].

3.2 The Global Preemptive Semantics

Figure 7 shows the global states and selected global semantics rules to model the interaction between modules. The world $\mathcal{W}$ consists of the thread pool $\mathcal{T}$, the ID $\mathcal{t}$ of the current thread, a bit $d$ indicating whether the current thread is in an atomic block or not, and the memory state $\mathcal{S}$. The thread pool $\mathcal{T}$ maps a thread ID to a triple recording the module language $\mathcal{tl}$, the free list $\mathcal{F}$, and the current core state $\mathcal{K}$.

\(^5\)In our Coq code, a footprint contains two additional fields $\mathit{cmps}$ and $\mathit{frees}$ for operations that observe and modify memory permissions. They allow footprints to capture memory operations more precisely, but they are orthogonal to our main ideas for supporting concurrency and hence merged into $rs$ and $ws$ in the paper to simplify the presentation.

\(^6\)As we mentioned, our Coq implementation supports external function calls, so $T$ actually maps a thread ID to a stack of triples $\langle tl, F, k \rangle$. The Coq code also contains additional semantics rules for external calls. The formalization reuses the definitions in Compositional CompCert.
We need to record the atomic bits of all threads because world There is no rule like Preemptive and non-preemptive global semantics Figure 7. Preemptive and non-preemptive global semantics

The global world \( W \) is defined similarly as the preemptive world \( W \), except that \( W \) keeps an atomic bit map \( d \) recording whether each thread’s next step is inside an atomic block. We need to record the atomic bits of all threads because the context switch may occur when a thread just enters an atomic block.

The last two rules in Fig. 7 define the non-preemptive global steps \( \delta W \). More rules are shown in Appendix A. There is no rule like Switch of the preemptive semantics, since context-switch occurs only at synchronization points. \( \text{EntAt}_{np} \) and \( \text{ExtAt}_{np} \) execute one step of the current thread \( t \), and then non-deterministically switch to a thread \( t' \). The corresponding global steps produce the \( sw \) events.

4 The Footprint-Preserving Simulation

In this section, we define a module-local simulation as the correctness obligation of each module’s compilation, which is compositional and preserves footprints, allowing us to derive a whole-program simulation that preserves DRF.

Footprint consistency. As in CompCert, the simulation requires that the source and the target generate the same external events. In addition, it also requires that the target has the same or smaller footprints than the source, which is important to ensure DRF-preservation. Recall that the memory accessible by a thread \( t_i \) consists of two parts, the shared memory \( S \) and the local memory allocated from \( F_i \), as shown in Fig. 5. DRF informally requires that the threads never have conflicting accesses to the memory in \( S \) at the same time.

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

\[
\mu = (S, S, f), \text{ where } S, S \in \mathcal{P}(\text{Addr}) \text{ and } f \in \text{Addr} \rightarrow \text{Addr}. 
\]

Here \( S \) and \( S \) specify the shared memory locations at the source and the target respectively. The partial mapping \( f \) maps locations at the source level to those at the target. We require \( \mu \) to be well-formed, defined as \( \text{wf}(\mu) \) in Fig. 8.

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

Rely/guarantee conditions. We use rely and guarantee conditions to specify interaction between modules. They need to enforce the view of accessibility of shared and local memory in Fig. 5. More specifically, a module expects others to keep its local memory (in \( F \)) intact. In addition, although other modules may update the shared memory \( S \), they must preserve certain properties of \( S \). One such property is \( \text{closed}(S, \Sigma) \) (defined in Fig. 7), which ensures \( S \) cannot contain memory pointers pointing to local memory cells in any \( F_i \). Otherwise a thread \( t_i \) can update the memory in \( F_i \) by tracing these pointers.\(^6\) Also the invariant \( \text{Inv} \) should be preserved, which relates the contents of the corresponding memory locations

\(^6\)We disallow cross-module escape of pointers pointing to stack-allocated variables. (We still allow the escape within a module and the transfer of dynamically allocated heap data structures.) Appendix E presents the framework with the support of stack pointer escape. Adding the support looks relatively orthogonal to our main ideas for supporting concurrency, and we can follow the approach of Compositional CompCert (the part for stack pointer escape).
Towards Certified Separate Compilation for Concurrent Programs

Definition 3. \( F \) is the largest relation such that, whenever \( F \) is compatible with \( \varphi \) through \( \mathit{initM} \).

1. for all \( k', \Sigma' \) and \( \Delta \), if \( \mathcal{F} \triangleright (k, \Sigma) \triangleleft (k', \Sigma') \) and \( (\Delta_0 \cup \Delta) \subseteq (\Sigma \cup \mu, S) \), then one of the following holds:
   a. \( \exists j < i. (\mathcal{F}, (k, \Sigma'), \Delta_0 \cup \Delta) \leq^j (F, (k, \sigma), \delta_0) \), or
   b. there exist \( \kappa', \sigma', \delta \) and \( j \) such that:
      i. \( F \triangleright (k, \sigma) \triangleright^j (k', \sigma') \);
      ii. \( (\delta_0 \cup \delta) \leq^j (F, (k', \sigma'), \delta_0 \cup \delta) \); and
      iii. \( (\mathcal{F}, (k', \Sigma'), \Delta_0 \cup \Delta) \leq^j (F, (k', \sigma'), \delta_0 \cup \delta) \).

2. for all \( k' \) and \( i \), if \( \mathcal{F} \triangleright (k, \Sigma) \triangleleft^i (k', \Sigma') \), \( i \neq \tau \), and
   HG\((\Delta_0, \Sigma, \mathcal{F}, \Sigma')\), there exist \( \kappa', \delta, \sigma' \) and \( \kappa'' \) such that:
   a. \( F \triangleright (k, \sigma) \triangleright^\tau (k', \sigma') \), and \( F \triangleright (k', \sigma') \triangleright^{\tau \text{ em}} (k'', \sigma'') \), and
   b. \( \mathcal{L}(\mu, (\delta_0 \cup \delta), (\delta', F), (\Delta_0, \Sigma)) \), and
   c. for all \( \kappa'' \) and \( \Sigma', \) if \( \mathcal{R}(\mu, \Sigma, \Sigma', \mathcal{F}), (\sigma', \sigma'', F) \), then there exists \( j \) such that:
      \( (\mathcal{F}, (k', \Sigma'), \mathcal{F}), \leq^j (F, (k'', \sigma'')) \).

The simulation \( (\mathcal{F}, (k, \Sigma), \Delta_0) \leq^i (F, (k, \sigma), \delta_0) \) carries \( \Delta_0 \) and \( \delta_0 \), the footprints accumulated at the source and the target, respectively. The definition follows the diagram in Fig. 1(d). For every \( \tau \)-step in the source (case 1), if the newly generated footprints and the accumulated \( \Delta_0 \) are in scope (i.e. every location must either be from the freelist space \( \mathcal{F} \) of current thread, or from the shared memory \( \mu, S \)), then the step corresponds to zero or multiple \( \tau \)-steps in the target, and the simulation holds over the resulting states with the accumulated footprints and a new index \( j \). We carry the well-founded index to ensure the simulation preserves termination.

If the source step corresponds to at least one target steps (case 1-b), the footprints at the target must also be in scope and be consistent with the source level footprints. The accumulation of footprints allows us to establish \( \mathcal{F} \)match for compiler optimizations that reorder the instructions.

At the switch points when the source generates a nonsilent message \( i \) (case 2), if the footprints and states satisfy the high-level guarantee HG, the target must be able to generate the same \( i \), and the accumulated footprints and the state satisfy the low-level guarantee LG. We also need to consider the interaction with other modules or threads. For any environment steps satisfying \( \mathcal{R} \), the simulation must hold over the new states, with some index \( j \) and \( \emptyset \) footprints — Since the effects of the current thread have been made visible to the environments at the switch point, we can clear the accumulated footprints. Rely and the guarantees HG and LG are defined in Fig. 8, as explained before.

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

**Definition 4** (Reach Closed Module). ReachClose(sl, ge, γ) iff., for all f, k, Σ, F and S, if sl.InitCore(γ, f) = k, S = dom(Σ), ge ⊆ Σ, F ∩ S = Ø, and closed(S, Σ), then RC(F, S, (k, Σ)).

Here RC is defined as the largest relation such that, whenever RC(F, S, (k, Σ)), then for all Σ’ such that R(S, Σ’, S, (k, Σ)), and for all k’, Σ’, Σ’’, k, and Σ such that F ⊢ (k, Σ’), we have HG(Δ, Σ’’, F, S, (k, Σ’’)).

The relation RC(F, S, (k, Σ)) essentially says during every step of the execution of (k, Σ), HG always holds over the resulting footprints Δ and states, even with possible interference from the environment, as long as the environment steps satisfy the reify condition R defined in Fig. 8.

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

**Lemma 5** (Transitivity). ∀ sl, sl’, tl, γ, γ’, π.
if (sl, ge, γ) ≺_{φ} (sl’, ge’, γ’) and (sl’, ge’, γ’) ≺_{φ'} (tl, ge”, π), then (sl, ge, γ) ≺_{φφ'} (tl, ge”, π).

The proof of Lem. 5 relies on the auxiliary lemmas similar to “memory interpolations” in Compositional CompCert.

**Lemma 6** (Compositionality, ⊕ in Fig. 2).
For any f1, ..., fn, φ, = {(sl1, ge1, γ1), ..., (slm, gem, γm)}, and Π = {(tl1, ge1, γ1), ..., (tlm, gem, γm)}, if
∀ i ∈ [1, m], wd(sl1) ∧ wd(tl1) ∧ ReachClose(sl1, ge1, γ1)
∧ (sl1, ge1, γ1) ≺_{φ} (tl1, ge1, γ1),
then let Γ in f1 | ... | fn ↦ let Γ in f1 | ... | fn .

Lemma 6 shows the compositionality of our simulation. The whole program downward simulation ˆP ⊆ ˆP relates the non-preemptive execution of the whole source program P and target program P. The definition is given in Appendix B.

With determinism of the target module language (written as det(tl)), we can flip ˆP ⊆ ˆP to derive the upward simulation ˆP ⊆ ˆP. We give the definition of det(tl) and the Flip Lemma (⊕ in Fig. 2) in Appendix B. Lemma 7 shows the non-preemptive global simulation ensures the refinement.

**Lemma 7** (Soundness, ⊕ in Fig. 2). If ˆP ⊆ ˆP, then ˆP ⊆ ˆP.

### 5 Data-Race-Freedom

Below we first define the conflict of footprints.

δ1 ∼ δ2 iff (δ1.ws ∩ δ2.ws = Ø) ∨ (δ1.ws ∩ δ2.ws = Ø)

(δ1, d1) ∼ (δ2, d2) iff (δ1 ∼ δ2) ∧ (d1 = 0 ∨ d2 = 0)

Recall that, when used as a set, δ represents δ.rs ∪ δ.ws. Since we do not treat accesses of the same memory location inside atomic blocks as a race, we instrument a footprint δ with the atomic bit d to record whether the footprint is generated inside an atomic block (d = 1) or not (d = 0). Two

<table>
<thead>
<tr>
<th>P load</th>
<th>W</th>
<th>W’</th>
<th>W’ ≡ Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>predict(W, t1, (δ1, d1))</td>
<td>predict(W, t2, (δ2, d2))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1 ≠ t2</td>
<td>(δ1, d1) ∼ (δ2, d2)</td>
<td>Race</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W ≡ Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>W = (T, 0, σ)</td>
</tr>
<tr>
<td>T(t) = (F, κ)</td>
</tr>
<tr>
<td>F + (κ, σ) δ→ (κ’, σ’’)</td>
</tr>
</tbody>
</table>

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

instrumented footsteps (δ1, d1) and (δ2, d2) are conflicting if δ1 and δ2 are conflicting and at least one of d1 or d2 is 0.

We define data races in Fig. 9 for preemptive semantics. In the race rule, W steps to Race if there are conflicting footprints of two threads predicted from the current configuration through predict(W, t, (δ, d)). Then we define DRF(P): DRF(P) iff ¬P ≡ Race

NPDRF( ˆP ) is defined similarly. We can prove NPDRF is equivalent to DRF (⊕ and ♦ in Fig. 2). The following Lem. 8 shows the simulation preserves NPDRF. Given the equivalence, we know it also preserves DRF.

**Lemma 8** (NPDRF Preservation, ♦ in Fig. 2).
For any ˆP, ˆP, if ˆP ⊆ ˆP, and NPDRF( ˆP ), then NPDRF( ˆP ).

**Lemma 9** (Semantics Equivalence, ⊕ and ♦ in Fig. 2).
For any Π, f1, ..., fm, if DRF(let Π in f1 | ... | fm), then let Π in f1 | ... | fm ≡ let Π in f1 | ... | fm.

### 6 The Final Theorem

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

SeqComp ::= (CodeT, φ), where CodeT ∈ Module → Module

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

**Definition 10** (Sequential Compiler Correctness).
Correct(SeqComp, sl, tl) iff
∀ γ, π, ge, ge’. SeqComp.CodeT(γ) = π ∧ [SeqComp, φ](ge) = ge’
⇒ (sl, ge, γ) ≺_{SeqComp, φ} (tl, ge’, π).

The desired correctness GCorrect (Def. 11) of concurrent program compilation is the semantics preservation of whole programs, i.e., every target concurrent program is a refinement of the source. Here all the SeqComp must agree on the transformation φ of global environments (see item 1 below).
Our final theorem is then formulated as Thm. 12. It says if a set of sequential compilers are certified to satisfy our correctness obligation Correct, the source and target languages sl_i and tl_i are well-defined, and the target languages are deterministic, then the compiled Clight programs serve as a whole is GCorrect for compiling concurrent programs. The proof simply applies the lemmas that correspond to Ω-Θ in Fig. 2.

Theorem 12 (Final Theorem). For any SeqComp_1, . . . , SeqComp_m, sl_1, . . . , sl_m, tl_1, . . . , tl_m such that for any i ∈ {1, . . . , m} we have wd(sl_i), wd(tl_i), det(tl_i), and Correct(SeqComp_i, sl_i, tl_i), then GCorrect((SeqComp_1, sl_1, tl_1), . . . , (SeqComp_m, sl_m, tl_m)).

7 The CASCompCert Compiler

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

As a tiny example, Fig. 10(a) shows a spin-lock implementation in a simple imperative language which we call Clmp. (C) is an atomic block, which cannot be interrupted by other threads. The EntAtom and ExtAtom events are generated at the beginning and the end of the atomic block respectively. The command assert(t(B) aborts if B is false. This source implementation of locks serves as an abstract specification, which is manually translated to x86-SC_TSO, as shown in Fig. 10(b). The implementation is similar to the Linux spin-lock implementation (a.k.a. the TTAS lock [11]). To acquire the lock, the _l_acq block reads and resets the lock bit to 0 atomically by the lock-prefixed cmpxchg1, which ensures mutual exclusion. The spin loop reads the lock bit until it appears to be available (i.e., not 0). Note that the load and store operations in the spin and unlock blocks are not lock-prefixed. This optimization introduces benign races. With the external lock module, we can implement a DRF counter inc in Clmp, as shown in Fig. 10(c). An example whole program P is let {Y_C, Y_lock} in inc () || inc ().

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

(a) lock specification y_lock

lock: movl $l, %ecx 
movl $0, %edx 
1_acq: movl $1, %eax 
lock cmpxchg1 %edx, %ecx void inc(){ 
je enter 
spin: movl (%ecx), %ebx 
tmp = x; 
cmp $0, %ebx 
je spin 
unlock(); 
jmp _l_acq 
print(tmp); 
enter: retl 
unlock: movl $l, %eax 
movl $1, (%eax) 
(c) a Clight client y_C 
retl 
(b) lock implementation n_lock

Figure 10. Lock as an external module of Clight programs

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

7.1 Language Instantiations

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

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

(FList) F ::= b_1::b_2:: ... (Block) b ∈ N^* 
(Bindex) N ∈ N (Core) κ ::= (c, N) 
(Mem) σ ∈ Block →fin (N → val)

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

7.2 Adapting CompCert

Given the program \( \text{let } \{ y_1, \ldots, y_t, y_o \} \in f_1 \parallel \ldots \parallel f_n \) consisting of Clight modules \( y_i \) and the module \( y_o \) (we omit \( s \) and \( g \) in the modules to simplify the presentation), the compilation \( \text{Comp} \) is defined as

\[
\text{Comp}(\text{let } \{ y_1, \ldots, y_t, y_o \} \in f_1 \parallel \ldots \parallel f_n) = \text{CompCert}(y_1, \ldots, \text{CompCert}(y_t), \text{IdTrans}(y_o)) \in f_1 \parallel \ldots \parallel f_n
\]

where \( \text{CompCert} \) is the adapted compilation consisting of the original CompCert passes, and \( \text{IdTrans} \) is the identity translation which returns the object module unchanged.

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

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

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

**Theorem 14 (Correctness with x86-SC backend and obj.).**

\( \text{GCorrect}((\text{CompCert}, \text{Clight}, \text{x86-SC}), (\text{IdTrans}, \text{Clmp}, \text{Clmp})) \).

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

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

Our solution is to define a bijection between memories under the two models. As a result, the behaviors of a thread under our model are equivalent to its behaviors under CompCert model, and our module-local simulation can be derived

\[ \forall e \in \{1, \ldots, m\}, \text{CodeT}(\gamma_i) = \pi_e \land \text{injective}(\phi) \land (\text{CompCert}.\phi = \phi) \land (\text{CompCert}.\pi_e = \pi_e) \]
Towards Certified Separate Compilation for Concurrent Programs

<table>
<thead>
<tr>
<th>Compilation passes and framework</th>
<th>Spec</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CompCert</td>
<td>Ours</td>
</tr>
<tr>
<td>Cpingen</td>
<td>513</td>
<td>1021</td>
</tr>
<tr>
<td>Cminorgen</td>
<td>753</td>
<td>1556</td>
</tr>
<tr>
<td>Selection</td>
<td>336</td>
<td>500</td>
</tr>
<tr>
<td>RTLgen</td>
<td>428</td>
<td>543</td>
</tr>
<tr>
<td>Tailcall</td>
<td>173</td>
<td>328</td>
</tr>
<tr>
<td>Renumber</td>
<td>86</td>
<td>245</td>
</tr>
<tr>
<td>Allocation</td>
<td>704</td>
<td>785</td>
</tr>
<tr>
<td>Tunneling</td>
<td>131</td>
<td>339</td>
</tr>
<tr>
<td>Linearize</td>
<td>236</td>
<td>371</td>
</tr>
<tr>
<td>CleanupLabels</td>
<td>126</td>
<td>387</td>
</tr>
<tr>
<td>Stacking</td>
<td>730</td>
<td>1038</td>
</tr>
<tr>
<td>Assmigen</td>
<td>208</td>
<td>338</td>
</tr>
<tr>
<td>Compositionality (Lem. 6)</td>
<td>580</td>
<td></td>
</tr>
<tr>
<td>DRF preservation (Lem. 8)</td>
<td>358</td>
<td></td>
</tr>
<tr>
<td>Semantics equiv. (Lem. 9)</td>
<td>1540</td>
<td></td>
</tr>
<tr>
<td>Lifting</td>
<td>813</td>
<td></td>
</tr>
</tbody>
</table>

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

object code does not preserve termination for now, which we leave as future work.

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

**Lemma 16** (Restore SC semantics for DRF x86 programs). Let \(\Pi_{sc} = \{(t_{sc}, g_{sc}, \pi_1), \ldots, (t_{sc}, g_{sc}, \pi_m), (s_{lsc}, g_{sr}, o_{ro})\}\), and \(\Pi_{tsos} = \{(t_{tsos}, g_{tsos}, \pi_1), \ldots, (t_{tsos}, g_{tsos}, \pi_m), (t_{tsos}, g_{tsos}, o_{ro})\}\).

For any \(f_1, \ldots, f_n\), if

1. Safe(\(\Pi_{sc} \text{ in } f_1, \ldots, f_n\)) and DRF(\(\Pi_{sc} \text{ in } f_1, \ldots, f_n\)),
2. \((t_{tsos}, g_{tsos}, o_{ro}) \preccurlyeq \circ (s_{lsc}, g_{sr}, o_{ro})\),

then \(\Pi_{tsos} \text{ in } f_1, \ldots, f_n \preccurlyeq \Pi_{sc} \text{ in } f_1, \ldots, f_n\).

As explained before, Lem. 16 can be viewed as a strengthened DRF-guarantee theorem for x86-TSO in that, if we let \(o_{ro}\) contain only skip and \(g_{sr} = \emptyset\), Lem. 16 implies the DRF-guarantee of x86-TSO.

7.4 Proof Efforts in Coq

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

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

By contrast, implementing our framework is more challenging, which took us about 1 person year. In particular, proving the equivalence between non-preemptive and preemptive semantics for DRF programs took us more time than expected, although it seems to be a well-known folklore theorem. The co-inductive proofs there involve a large number of non-trivial cases of reordering threads’ executions.

8 Related Work and Conclusion

**Compiler verification.** Various work extends CompCert [16] to support separate compilation or concurrency. We have discussed Compositional CompCert [2, 29] in Sec. 1 and 2. SepCompCert [15] extends CompCert with the support of syntactical linking. Their approach requires all the compilation units be compiled by CompCert. They do not support cross-language linking or concurrency as we do.

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

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

Podkopaev et al. [24] prove correctness of the compilation from the promising semantics (which is a high-level operational relaxed model) to the operational ARMv8-POP machine. They develop whole-program simulations to deal with the complicated relaxed behaviors. Later on they verify compilations from the promising semantics to declarative hardware models such as POWER, ARMv7 and ARMv8 [25]. The simulations are tied to the promising semantics. They do not aim for reusing existing sequential compilations but handle a lot of complicated issues in relaxed models.

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

Vellvm [35, 36] proves correctness of several optimization passes for sequential LLVM programs. Wang et al. [32] verify a separate compiler from Cito to Bedrock, which relies on axiomatic specifications for cross-language external calls. It is unclear how to adapt their work to concurrency. Perconti and Ahmed [23] verify separate compilation by embedding languages in a combined language. They do not support concurrency either. Ševčík [26] studies safety of a class of optimizations in concurrent settings using an abstract trace semantics. It is unclear if his approach can be applied to verify general compilation. Lochbihler [20] verifies a compiler for concurrent Java programs. His simulation has similar restrictions as CompCertTSO.

Non-preemptive semantics and data-race-freedom. Non-preemptive (or cooperative) semantics has developed in various settings for various purposes (e.g., [1, 5, 18, 31, 34]). Both Ferreira et al. [9] and Xiao et al.[33] study the relationships between non-preemptive semantics and DRF, but they do not give any mechanized proofs of termination-preserving semantics equivalence as in our work. DRFx [21] proposes a concept called Region-Conflict-Freedom, which looks similar to our NPDRF, but there is no formal operational formulation as we do. Owens [22] proposes Triangular-Race-Freedom (TRF) and proves that TRF programs behaves the same in x86-SC and x86-TSO. TRF is weaker than DRF and can be satisfied by the efficient spin lock code in Fig. 10.

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

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

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

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

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

Acknowledgments

We thank our shepherd John Reppy, Zhong Shao and anonymous referees for their suggestions and comments on earlier versions of this paper. We especially thank Weijie Fan and Jianwei Song for their helps on the Coq proofs of the optimization passes Tunnel1ng and CleanupLabels. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant No. 61632005, and from Huawei Innovation Research Program (HIRP).

References

Towards Certified Separate Compilation for Concurrent Programs

\[
T(t) = (t, F, \kappa) \quad F \triangleright (\kappa, \sigma) \xrightarrow{\epsilon}_{\text{emp}} (\kappa', \sigma)
\]

\[
T' = T[t \sim (t, F, \kappa)]
\]

\[
\frac{(T, t, 0, \sigma) \xrightarrow{\text{emp}} (T', t, 0, \sigma)}{\text{Term}}
\]

\[
T(t) = (t, F, \kappa) \quad t' \in \text{dom}(T\setminus t) \quad F \triangleright (\kappa, \sigma) \xrightarrow{\text{ret}_{\text{emp}}} (\kappa', \sigma)
\]

\[
\frac{(T, t, 0, \sigma) \xrightarrow{\text{sw}_{\text{emp}}} (T\setminus t, t', 0, \sigma)}{\text{Done}}
\]

\[
T(t) = (t, F, \kappa) \quad \text{dom}(T) = \{t\} \quad F \triangleright (\kappa, \sigma) \xrightarrow{\text{ret}_{\text{emp}}} (\kappa', \sigma)
\]

\[
\frac{(T, t, 0, \sigma) \xrightarrow{\epsilon}_{\text{emp}} \text{done}}{\text{Done}}
\]

\[
T(t) = (t, F, \kappa) \quad F \triangleright (\kappa, \sigma) \xrightarrow{\epsilon}_{\delta} \text{abort}
\]

\[
\frac{(T, t, 0, \sigma) \xrightarrow{\epsilon}_{\delta} \text{abort}}{\text{Abort}}
\]

Figure 14: More Rules of the Preemptive Global Semantics

In ECOOP. 504–528.


A More about the Language Settings

Global preemptive semantics. Fig. 14 gives the omitted rules of the global preemptive semantics.

The Print rule shows the step generating an external event. It requires that the flag d must be 0, i.e., external events can be generated only outside of atomic blocks. Also the step generating an external event does not access memory, so its footprint is empty (\text{emp}).

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

The Abort rule says the whole program aborts if a local module aborts.

We write \( F \triangleright \phi \xrightarrow{\epsilon}_{\delta} \phi' \) for zero or multiple silent steps, where \( \delta \) is the accumulation of the footprint of each step. \( F \triangleright \phi \xrightarrow{\epsilon}_{\delta} \phi' \) represents at least one step. Similar notations are used for global steps. We also write \( W \xrightarrow{\tau} W' \) for multiple steps that either are silent or produce sw events. It must contain at least one silent step. \( W \xrightarrow{\epsilon} W' \) represents multiple steps with exactly one \( e \) event produced (while other steps either are silent or produce sw events).
for all $i$ and $j$ in $\{1, \ldots, n\}$, and $i \neq j$:

\begin{align*}
F_i \cap F_j &= \emptyset, \\
\tau_i, \text{InitCore}(\pi_i, t_j) &= \kappa_j, \text{ where } (\tau_i, \text{get}_i, \pi_i) \in \Pi \\
T &= \{1 \sim (\tau_i, F_i, k_j), \ldots, n \sim (\tau_n, F_n, k_n)\} \\
\sigma &= \text{GE}(\Pi) \; \text{closed}(\sigma) \\
t &\in \text{dom}(T) \quad d = \{t_1 \sim 0, \ldots, t_n \sim 0\}
\end{align*}

**Definition 17 (Event-Trace Refinement and Equivalence).**

$P \subseteq \mathbb{P}$ iff $\forall B. \; \text{PETr}(P, B) \implies \text{PETr}(\mathbb{P}, B)$.

$P \approx \mathbb{P}$ iff $\forall B. \; \text{PETr}(P, B) \iff \text{PETr}(\mathbb{P}, B)$.

**Safety.** Below we use the event traces to define $\text{Safe}(\mathcal{W})$ and $\text{Safe}(\mathbb{P})$.

**Definition 18 (Safety).**

$\text{Safe}(\mathbb{P})$ iff $\exists \mathcal{W}. \; P \Rightarrow \mathcal{W}$ and $\forall \mathcal{W}. \; (P \Rightarrow \mathcal{W}) \Rightarrow \text{Safe}(\mathcal{W})$.

**Non-preemptive semantics.** Fig. 15 gives the omitted non-preemptive semantics rules. Like $\text{EntAt}_{np}$ and $\text{ExtAt}_{np}$, the rules $\text{Print}_{np}$ and $\text{Term}_{np}$ execute one step of the current thread $t$, and then non-deterministically switch to a thread $t'$ (which could just be $t$). The corresponding global steps produce the $\text{sw}$ events (or the external event $e$ in the $\text{Print}_{np}$ rule). Other rules are very similar to their counterparts in the preemptive semantics.

**B More about the Simulation**

**Proof of Lemma 6.** Proof of Lemma 6 relies on the following non-interference lemma.

**Lemma 19 (Non-Interference).** For any $k, k'$ of some well-defined language $sl$ and any $\kappa, \kappa'$ of some well-defined language $tl$, if $\mathbb{F}_{1} \vdash (k, \Sigma_0) \xrightarrow{\lambda_0} (k', \Sigma)$, and $F_1 \vdash (k, \Sigma_0) \xrightarrow{\delta_0} (k', \Sigma)$, then for any $\mu, \mathbb{F}_2$ and $\mathbb{F}_2$ such that $(\mu, \Sigma \cup F_1) \cap \mathbb{F}_2 = \emptyset$, and $(\mu, \Sigma \cup F_1) \cap \mathbb{F}_2 = \emptyset$, we have

\begin{align*}
\text{HG}(\lambda_0, \Sigma, F_1, \mu, \Sigma) \land \text{LG}(\mu, \delta_0, \Sigma, F_1, (\lambda_0, \Sigma)) \\
\implies Rely(\mu, (\lambda_0, \Sigma, \mathbb{F}_2), (\delta_0, \Sigma, \mathbb{F}_2)).
\end{align*}

Here we use $\mathbb{F} \vdash (k, \Sigma_0) \xrightarrow{\lambda_0} (k', \Sigma)$ to say:

- either $\mathbb{F} \vdash (k, \Sigma_0) \xrightarrow{\lambda_0} (k', \Sigma)$,
- or $\mathbb{F} \vdash (k, \Sigma_0) \xrightarrow{\delta_0} (k'', \Sigma)$ and $\mathbb{F} \vdash (k'', \Sigma) \xrightarrow{\delta_0} (k', \Sigma)$ for some $k''$.

The non-interference lemma says, after the execution of $(k, \Sigma_0)$ and $(k, \Sigma_0)$ until some interaction point with resulting states $\Sigma$ and $\sigma$, we are able to establish the Rely condition over the initial and resulting states with respect to $\mu$ and some freelists $\mathbb{F}_2$ and $\mathbb{F}_2$, if the freelists $\mathbb{F}_2$ and $\mathbb{F}_2$ contains no shared location and are disjoint with the freelists $\mathbb{F}_1$ and $F_1$, respectively, and the execution guarantees $\text{HG}$ and $\text{LG}$. Here $\text{LG}$ is established by our module-local simulation if $\text{HG}$ is guaranteed, and $\text{HG}$ is established by $\text{RC}$ by the following lemma, which says we can establish $\text{HG}$ for multiple steps if the module and corresponding state satisfies $\text{RC}$.

**Lemma 20 (RC-guarantee).** If $\text{RC}(\Sigma, \mathbb{F}, (k, \Sigma_0))$ and $\mathbb{F} \vdash (k, \Sigma_0) \xrightarrow{\delta_0} (k', \Sigma)$ then $\text{HG}(\lambda_0, \Sigma, F_1, \Sigma)$.
The non-preemptive global downward simulation. The definition of the whole program simulation is similar to the module local simulation, except for the case for environment interference (Rely steps), which is unnecessary for whole program simulation. Every source step should correspond to multiple target steps. As in module local simulation, we always require the target footprints matches those of the source, defined as FPmatch. Also the footprint should always be in scope. As a special requirement for the whole program simulation, we always require the source and the target do lock-step context switch and they always switch to the same thread.

Below we use \( \text{curF} \) for the freelist of the current thread, i.e., \( \text{curF}(\tau, t_{\_\_\_}) \) def \( \equiv \text{ curF}(t) \).

**Definition 21** (Whole-Program Downward Simulation).
Let \( \hat{P} = \text{let } \Gamma \text{ in } f_1 | \ldots | f_m, \hat{P} = \text{let } \Pi \text{ in } f_1 | \ldots | f_m, \Gamma = \{(sl_1, ge_{r_1}, \gamma_1), \ldots, (sl_m, ge_{r_m}, Y_m)\} \) and \( \Pi = \{(tl_1, ge_{l_1}, \pi_1), \ldots, (tl_m, ge_{l_m}, \pi_m)\} \).
We say \( \hat{P} \leq \hat{P} \) iff

- \( \exists \phi. (\forall 1 \leq i \leq m. [\phi] e_i = ge_{c_i}) \), and
- for any \( \hat{W} \), if \( \hat{P} : \text{load } \hat{W} \), then there exists \( W, i \in \text{index}, \mu \), such that \( P : \text{load } W \) and \( \hat{W} \text{, emp} \leq \mu (W, \text{emp}) \).

Here we define \( \leq \mu \) \((\hat{W}, \Delta_0) \leq \mu (\hat{W}, \Delta_0)\) as the largest relation such that whenever \((\hat{W}, \Delta_0) \leq \mu (\hat{W}, \Delta_0)\), then the following are true:

1. \( \hat{W} \, t = \hat{W} \, t, \text{ and } \hat{W} \, d = \hat{W} \, d \);
2. \( \forall \hat{W}', \Delta, \text{ if } \hat{W} \rightarrow (\hat{W}', \Delta) \text{ then one of the following holds:} \)
   a. \( \exists j. j < i, \text{ and } (\hat{W}', \Delta_0 \cup \Delta) \leq \mu (\hat{W}, \Delta_0) \); or
   b. \( \exists \hat{W}', \Delta, j. \)
      i. \( W \rightarrow (\hat{W}', \Delta) \), and
      ii. \( (\Delta_0 \cup \Delta) \subseteq (\text{curF}(W) \cup \mu.S), \text{ and } \)
      iii. \( \hat{W}, \Delta_0 \cup \Delta \leq \mu \)
3. \( \forall \hat{T}', \hat{d}', \Sigma, o, t'. \text{ if } o \neq \tau \text{ and } \hat{W} : \text{post } (\hat{T}', \hat{d}', \Sigma) \), then
   a. \( \hat{W} \rightarrow (\hat{W}') \), and
   b. \((\Delta_0 \cup \Delta) \subseteq (\text{curF}(W) \cup \mu.S), \text{ and } \)
   c. \( ((\hat{T}', \hat{d}', \Sigma), \text{ emp}) \leq \mu \)
4. if \( \hat{W} \rightarrow \text{ done } \), then \( \forall \hat{W}', \delta \).
   a. \( \hat{W} \rightarrow (\hat{W}') \), and
   b. \((\Delta_0 \cup \Delta) \subseteq (\text{curF}(W) \cup \mu.S), \text{ and } \)

The condition on context switches (lock step, and to the same thread) in our whole-program simulations is not so strong as it sounds. It can be naturally derived from composing our module-local simulations. In our module-local simulation, we already have lock-step EntAtom/ExtAtom steps (see case 2 in Def. 3), which will be switch points in whole programs. Besides, since the local simulation is supposed to relate the source and target of the same thread, it seems natural to require the source and target to always switch to the same thread in the whole-program simulation. The switching-to-the-same-thread condition makes it easy to flip the whole-program simulation (i.e. derive the whole program upward simulation from the downward one). The upward and downward simulations require us to map each target thread to some source thread and also map each source thread to a target one. Since the number of target threads is the same as the number of source threads, we would need to require the mapping between the source and target threads to be a bijection. The easiest way to ensure the bijection is to let the thread identifiers to be the same.

**The non-preemptive global upward simulation.** We define the whole program upward simulation as \( \hat{P} \leq \hat{P} \) in Def. 22. It is similar to the downward simulation \( \hat{P} \leq \hat{P} \), with the positions of source and target swapped. Note that we do not flip the FPmatch condition, since we always require footprint of target program being a refinement of the footprint of the source program, in order to prove DRF of the target program. Correspondingly, the address mapping \( \varphi \) is not flipped either.

**Definition 22** (Whole-Program Upward Simulation).
Let \( \hat{P} = \text{let } \Gamma \text{ in } f_1 | \ldots | f_m, \hat{P} = \text{let } \Pi \text{ in } f_1 | \ldots | f_m, \Gamma = \{(sl_1, ge_{r_1}, \gamma_1), \ldots, (sl_m, ge_{r_m}, Y_m)\} \) and \( \Pi = \{(tl_1, ge_{l_1}, \pi_1), \ldots, (tl_m, ge_{l_m}, \pi_m)\} \).
We say \( \hat{P} \leq \hat{P} \) iff

- \( \exists \phi. (\forall 1 \leq i \leq m. [\phi] e_i = ge_{c_i}) \), and
- for any \( \hat{W} \), if \( \hat{P} : \text{load } \hat{W} \), then there exists \( W, i \in \text{index}, \mu \), such that \( P : \text{load } W \) and \( \hat{W} \text{, emp} \leq \mu (W, \text{emp}) \).

Here we define \( \leq \mu \) \((\hat{W}, \Delta_0) \leq \mu (\hat{W}, \Delta_0)\) as the largest relation such that whenever \((\hat{W}, \Delta_0) \leq \mu (\hat{W}, \Delta_0)\), then the following are true:

1. \( \hat{W} \, t = \hat{W} \, t, \text{ and } \hat{W} \, d = \hat{W} \, d \);
2. \( \forall \hat{W}', \Delta, \text{ if } \hat{W} \rightarrow (\hat{W}', \Delta) \text{ then one of the following holds:} \)
   a. \( \exists j. j < i, \text{ and } (\hat{W}', \Delta_0 \cup \Delta) \leq \mu (\hat{W}, \Delta_0) \); or
   b. \( \exists \hat{W}', \Delta, j. \)
      i. \( W \rightarrow (\hat{W}', \Delta) \), and
      ii. \( (\Delta_0 \cup \Delta) \subseteq (\text{curF}(W) \cup \mu.S), \text{ and } \)
      iii. \( \hat{W}, \Delta_0 \cup \Delta \leq \mu \)
3. \( \forall \hat{T}', \hat{d}', \Sigma, o, t'. \text{ if } o \neq \tau \text{ and } \hat{W} : \text{post } (\hat{T}', \hat{d}', \Sigma) \), then
Figure 16. Data Race in Non-Preemptive Semantics

a. \( \widetilde{W} \xrightarrow{\phi} \widetilde{W}' \), and \( \overset{\phi}{\widetilde{W}'} \xrightarrow{\text{emp}} (\tau', t', d', \Sigma') \), and
b. \( \delta_0 \subseteq \text{curF}(\widetilde{W}) \cup \mu.S, \text{ and FPMatch}(\mu, \delta_0, \delta_0) \); and
c. \( ((\tau', t', d', \Sigma'), \text{emp}) \xrightarrow{\delta_0} ((\tau', t', d', \Sigma'), \text{emp}); \)
d. if \( \overset{\text{done}}{\widetilde{W}} \), then \( \exists \widetilde{W}', \Delta \).
e. \( \overset{\text{done}}{\widetilde{W}} \xrightarrow{\text{emp}} \widetilde{W}' \), and \( \overset{\text{done}}{\text{emp}} \)
f. \( \delta_0 \subseteq \text{curF}(\widetilde{W}) \cup \mu.S, \text{ and FPMatch}(\mu, \delta_0, \delta_0) \).

Determinism of the target language, and flip of the global simulation.

Definition 23 (Deterministic Languages). \( \text{det} (tl) \iff \) for all configuration \( \phi \) in \( tl \) (see the definition of \( \phi \) in Fig. 4), and for all \( F, F \vdash \phi \overset{\text{emp}}{\phi_1} \land F \vdash \phi \overset{\text{emp}}{\phi_2} \implies \phi_1 = \phi_2 \land t_1 = t_2 \land \delta_1 = \delta_2 \).

Lemma 24 (Flip, \( \otimes \) in Fig. 2).

For any \( f_1, \ldots, f_m, g_1, g_2, \Gamma = \{(s_{h_1}, g_{e_1}, Y_1), \ldots, (s_{h_m}, g_{e_m}, Y_m)\}, \)
\( \Pi = \{(t_{l_1}, g_{e_1}, \pi_1), \ldots, (t_{l_m}, g_{e_m}, \pi_m)\}, \) if \( \forall i. \text{det}(tl_i), \) and
let \( \Gamma \) in \( f_1 \mid \ldots \mid f_m \) \( \overset{\varphi}{\leq} \) let \( \Pi \) in \( f_1 \mid \ldots \mid f_m \),
then let \( \Pi \) in \( f_1 \mid \ldots \mid f_m \) \( \overset{\varphi}{\leq} \) let \( \Gamma \) in \( f_1 \mid \ldots \mid f_m \).

C More about DRF and NPDPDF

Data races in the non-preemptive semantics (\( \overset{\text{load}}{P} \implies \text{Race} \)) are defined in Fig. 16. Similar to the definition for the pre-emptive semantics, a non-preemptive program configuration \( \overset{\text{emp}}{W} \) steps to Race (\( \overset{\text{emp}}{W} \implies \text{Race} \)) if two threads are predicted having conflicting footprints, as shown in rule Race_{emp}. Predictions are made only at the switch points of non-preemptive semantics (\( \overset{\text{emp}}{W} \implies \overset{\delta}{\text{emp}} \) where \( \overset{\text{emp}}{W} \implies \overset{\delta}{\text{emp}} \), i.e., program points at atomic block boundaries, after an observable event, or after thread termination. The prediction rule Predict_{np} is a unified version of its counterpart Predict-0 and Predict-1 in Fig. 9, where \( \delta \) indicates whether the predicted steps are inside an atomic block. Similar to the Predict-1 rule, we do not insist on predicting the footprint generated by the execution reaching the next switch point, because the code segments between switch points could be nonterminating. In addition to the switch points, we also need to be able to perform a prediction at the initial state as well (the second rule in Fig. 16), because the first executing thread is non-deterministically picked at the beginning, which has similar effect as thread switching.

A program \( \overset{\text{load}}{P} \) is NPDRF if it never steps to Race:

\[
\text{NPDRF}(P) \iff \neg(\overset{\text{load}}{P} \implies \text{Race})
\]

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

Lemma 25 (Equivalence between DRF and NPDRF, \( \otimes \) and \( \otimes \) in Fig. 2). For any \( f_1, \ldots, f_m, \Pi = \{(t_{l_1}, \pi_1), \ldots, (t_{l_m}, \pi_m)\} \) such that \( \forall i. \text{wd}(tl_i), \) and \( P = \text{let } \Pi \text{ in } f_1 \parallel \ldots \parallel f_m, \)

\[
\text{DRF}(P) \iff \text{NPDRF}(P).
\]

D More about the Extended Framework

D.1 x86-TSO semantics

Our x86-TSO semantics follows the x86-TSO model presented in [22, 28] and CompCertTSO [27].

To make the refinement proof easy, we choose to present the x86-TSO global semantics in a similar way as our preemptive global semantics. The machine states and the global transition rules are shown in Fig. 17 and 18 respectively. Here

- All modules are x86-TSO modules.
- The whole program configuration \( \overset{\text{emp}}{W} \) is instrumented with write buffers B. Each thread \( t \) has its own write buffer \( B(t) \).
- We do not need the atomic bit and EntAtom, ExtAtom events in the x86-TSO model.

The Unbuffer rule in the global semantics in Fig. 18 says that the machine non-deterministically apply the writes in a thread’s buffer to the real memory.

The x86 instruction sets and the local semantics are shown in Fig. 19. For normal writes, a thread always writes to its buffer rather than the real memory. For lock-prefix instructions, the thread requires the buffer is empty and it updates the memory directly (see the LOCKDEP rule).
Towards Certified Separate Compilation for Concurrent Programs

\[
\begin{align*}
(\text{Prog}) \quad P &::= \text{let } \Pi \text{ in } f_1 \parallel \ldots \parallel f_n \\
(\text{MdSet}) \quad \Pi &::= \{(g_{e_1}, \pi_1), \ldots, (g_{e_m}, \pi_m)\} \\
(\text{Lang}) \quad t_{x86-\text{TSO}} &::= (\text{Module}, \text{Core}, \text{InitCore}, \text{tso}) \\
&\quad \text{Core } \kappa ::= (\pi, \ldots) \\
&\quad \text{ge} \in \text{Addr} \rightarrow_{\text{fin}} \text{Val} \\
&\quad \text{InitCore} \in \text{Module} \rightarrow \text{Entry} \rightarrow \text{list(Val)} \rightarrow \text{Core} \\
&\quad \text{tl} \in \text{N} \\
&\quad \text{Addr} \leftrightarrow_{\text{fin}} \text{Val} \\
&\quad \text{Val} v ::= l \mid \ldots \\
&\quad \text{Buf} b ::= e \mid (l, v) ::: b \\
&\quad \pi \in P_{\omega}(\text{Addr}) \\
&\quad \text{FinCore} \in \text{Module} \rightarrow \text{Entry} \rightarrow \text{Val} \\
&\quad \text{Abort} \in \text{Abort} \\
(\text{ThrdID}) \quad t \in \text{N} \\
&\quad \text{Addr} \leftrightarrow_{\text{fin}} \text{Val} \\
(\text{State}) \quad \sigma \in \text{Addr} \rightarrow_{\text{fin}} \text{Val} \\
&\quad \text{Val} v ::= l \mid \ldots \\
(\text{Buf}) \quad b ::= e \mid (l, v) ::: b \\
&\quad \text{FList, F} \in P_{\omega}(\text{Addr}) \\
(\text{Msg}) \quad \iota ::= \tau \mid e \mid \text{call}(f, \overline{v}) \mid \text{ret}(v) \\
(\text{Event}) \quad e ::= \text{print}(v) \\
(\text{Config}) \quad \phi, \Phi ::= (\kappa, \sigma) \mid \text{Abort} \\
(\text{World}) \quad W_{\text{tso}} ::= (\Pi, F, T, t, B, \sigma) \\
(\text{ThrdPool}) \quad T, T ::= \{t_1 \sim K_1, \ldots, t_n \sim K_n\} \\
(\text{RtMdStk}) \quad K, X ::= e \mid (F, \kappa) ::: K \\
(\text{Buffers}) \quad B ::= \{t_1 \sim b_1, \ldots, t_n \sim b_n\} \\
(\text{FSpace}) \quad F, F ::= F :: F \quad \text{(co-inductive)} \\
(\text{GMsg}) \quad o ::= \tau \mid e \mid \text{sw} \mid \text{ub} \quad (\text{ASET}) \quad S, S ::= P(\text{Addr})
\end{align*}
\]

Figure 17. x86TSO machine
\( T_0 = F_1 \cdots F_n : T \)  
\( T = F_{n+1} : F_{n+2} \cdots \)

for all \( i \neq j \): \( F_i \cap F_j = \emptyset \)  
\( \text{dom}(\sigma) \cap F_i = \emptyset \)  

for all \( i \in \{1, \ldots, n\}: \)  
\( (t_{\text{b86-D50}} \cdot \text{InitCore}(\pi_i, F_i)) = \kappa_i \)  
where \((t_{\text{b86-D50}} \cdot \text{Core}(\pi, \pi_i)) \in \Pi \)

\( T = \{ t \sim (t_{\text{b86-D50}} \cdot F_i, \kappa_i) : \epsilon \} \)
\( t \in \{1, \ldots, n\} \)  
\( \sigma = \text{GEM}(\Pi) \)  
\( \text{closed}(\sigma) \)

\[
\begin{array}{c}
\text{let } \Pi \text{ in } f_1 \text{ } \| \text{ } \| f_n \text{ } \Rightarrow_{tso} \text{ Load } \\
\text{T(t) = (F, \kappa) : K B(t) = b} \\
F \vdash (\kappa, (b, \sigma)) \xrightarrow{\delta_{tso}} (\kappa', (b', \sigma')) \\
T' = T \{ t \sim (\sigma, \kappa') : K \} \quad t = \tau \\
\text{r-step} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\delta_{tso}} (\Pi, T', t, B \{ t \sim (\sigma, b') \}) \\
\text{Print} \\
\text{T(t) = (F, \kappa) : K B(t) = \epsilon} \\
F \vdash (\kappa, (\epsilon, \sigma)) \xrightarrow{\text{emp}_{tso}} (\kappa', (\epsilon, \sigma)) \\
T' = T \{ t \sim (\sigma, \kappa') : K \} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\epsilon_{tso}} (\Pi, T', t, B, \sigma) \\
\text{EFCall} \\
\text{T(t) = (F_1, \kappa) : (F, \kappa) : K, F_1 \vdash (\kappa, (b, \sigma)) \xrightarrow{\text{emp}_{tso}} (\kappa', (b, \sigma))} \\
AftExn(\kappa, v) = \kappa' \\
T' = T \{ t \sim (\sigma, \kappa') : K \} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\epsilon_{tso}} (\Pi, T', t, B, \sigma) \\
\text{EFRet} \\
\text{T(t) = (F, \kappa) : \epsilon B(t) = \epsilon \text{ dom}(T) = \text{dom}(B) = \{ t \}} \\
F \vdash (\kappa, (\epsilon, \sigma)) \xrightarrow{\text{emp}_{tso}} (\kappa', (\epsilon, \sigma)) \\
(\Pi, T, t, B, \sigma) \xrightarrow{\epsilon_{tso}} (\Pi, T, t, B, \sigma) \\
\text{Term} \\
\text{T(t) = (F, \kappa) : \epsilon B(t) = \epsilon \quad \text{dom}(T) = \text{dom}(B) = \{ t \}} \\
F \vdash (\kappa, (\epsilon, \sigma)) \xrightarrow{\text{emp}_{tso}} (\kappa', (\epsilon, \sigma)) \\
(\Pi, T, t, B, \sigma) \xrightarrow{\epsilon_{tso}} (\Pi, T, t, B, \sigma) \\
\text{Done} \\
\text{T(t) = (F, \kappa) : \epsilon B(t) = \epsilon \text{ dom}(T) = \text{dom}(B) = \{ t \}} \\
F \vdash (\kappa, (\epsilon, \sigma)) \xrightarrow{\text{emp}_{tso}} (\kappa', (\epsilon, \sigma)) \\
(\Pi, T, t, B, \sigma) \xrightarrow{\epsilon_{tso}} (\Pi, T, t, B, \sigma) \\
\text{Unbuffer} \\
\text{t' \in dom(T)} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\text{sw}_{tso}} (\Pi, T, t', B, \sigma) \\
\text{Switch} \\
\text{B(t') = (l, v) : b} \quad \sigma' = \text{apply_buffer}((l, v), \sigma) \quad B' = B \{ t' \sim b \} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\text{ub}_{tso}} (\Pi, T, t, B', \sigma') \\
\text{Abort} \\
(\Pi, T, t, B, \sigma) \xrightarrow{\text{ab}_{tso}} \text{abort} \\
\end{array}
\]

\text{Figure 18. x86TSO global semantics}
Towards Certified Separate Compilation for Concurrent Programs

$$\begin{align*}
(\text{Instr}) & \quad c ::= \text{mov } r_d, r_s | \text{Iadd } r_d, r_s \mid \text{call } f \mid \text{ret} \mid \ldots \\
 & \quad \mid \text{lock } \text{xchg } l | \text{lock } \text{add } l r_s \mid \text{lock } \text{dec } l \mid \text{mfence}
(\text{Core}) & \quad \kappa ::= \text{Register} \rightarrow \text{Val} \\
(\text{Register}) & \quad r ::= \text{PC} \mid \text{SP} \mid \ldots
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = (\text{mov } r_d, [r_s]) & \quad \kappa(r_s) = l \\
& \quad l \notin \text{dom}(b) \quad \sigma(l) = v \quad \delta = ([l], \emptyset) \quad \text{READMEM}
\end{align*}$$

$$\begin{align*}
(\kappa, (b, \sigma)) & \rightarrow_{\delta}^{\tau} (\kappa[r \sim \nu, \text{PC} \sim \text{PC} + 1], (b, \sigma))
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = (\text{mov } r_d, [r_s]) & \quad \kappa(r_s) = l \\
& \quad b = b_1 + (l, v) \in b_0 \quad l \notin \text{dom}(b) \quad \delta = ([l], \emptyset) \quad \text{READBUF}
\end{align*}$$

$$\begin{align*}
(\kappa, (b, \sigma)) & \rightarrow_{\delta}^{\tau} (\kappa[r \sim \nu, \text{PC} \sim \text{PC} + 1], (b, \sigma))
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = (\text{mov } [r_d], r_s) & \quad \kappa(r_s) = \nu \quad \kappa(r_d) = l \\
& \quad \delta = (\emptyset, [l]) \quad \text{apply_buffer}(b + (l, v), \sigma) = \sigma' \quad \text{WRITEBUF}
\end{align*}$$

$$\begin{align*}
(\kappa, (b, \sigma)) & \rightarrow_{\delta}^{\tau} (\kappa[\text{PC} \sim \text{PC} + 1], (b + (l, v), \sigma))
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = \text{lock } \text{dec } l r & \quad R(\text{SF}) = \sigma(l) \leq 0 \\
& \quad \kappa' = \kappa[\text{PC} \sim \text{PC} + 1, r \sim \sigma(l)] \quad \sigma' = \sigma[l \sim \sigma(l) - 1] \quad \delta = ([l], [l]) \quad \text{LOCKDEC}
\end{align*}$$

$$\begin{align*}
(\kappa, (\epsilon, \sigma)) & \rightarrow_{\delta}^{\tau} (\kappa', (\epsilon, \sigma'))
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = \text{mfence} & \quad \text{BARRIER}
\end{align*}$$

$$\begin{align*}
(\kappa, (\epsilon, \sigma)) & \rightarrow_{\text{emp}}^{\tau} (\kappa[\text{PC} \sim \text{PC} + 1], (\epsilon, \sigma))
\end{align*}$$

$$\begin{align*}
\text{find_instr}(\kappa, \kappa(\text{PC})) = \text{call } \text{print}(l) & \quad \text{PRINT}
\end{align*}$$

$$\begin{align*}
\kappa' = \kappa[\text{PC} \sim \text{PC} + 1] \quad \sigma(\text{first argument}) = \nu
\end{align*}$$

$$\begin{align*}
(\kappa, (\epsilon, \sigma)) & \rightarrow_{\text{print}(\nu)}^{\tau} (\kappa', (\epsilon, \sigma))
\end{align*}$$

$$\begin{align*}
\text{apply_buffer}(l, v) & \in (\epsilon, \sigma) \\
\text{apply_buffer}(\epsilon, \sigma) & \in \sigma
\end{align*}$$

\textbf{Figure} 19. x86TSO thread local semantics
D.2 Object correctness definition

Definition 26 (Object correctness).

\( \langle ltsO, ge_r, \tau_o \rangle \models^\sigma (\text{slImp, ge}_r, \varnothing) \) if \( \exists \sigma, G_0, o_0 \) such that

1. For any \( t \), \( \langle \text{slImp, ge}_r, \varnothing \rangle \models^\sigma (ltsO, ge_r, \tau_o) \)
2. \( \text{ResPart}_o(R_0, G_0, o_0, \text{objM}) \)
3. \( \text{Sta}(o_0, R_0) \)
4. \( \forall t_1 \neq t_2, G_{t_1} \Rightarrow R_{t_2} \)

Where \( \text{ResPart}_o \) is defined in definition 27. \( \langle \text{slImp, ge}_r, \varnothing \rangle \models^\sigma (ltsO, ge_r, \tau_o) \) is an extension of Liang and Feng [19] with the support of TOSO semantics for the low-level code, and is defined in definition 28.

Definition 27 (Respect partition).

\( \text{ResPart}_o(R_0, G_0, o_0) \) iff

1. \( \forall \Sigma, B, \sigma, \Sigma', B', \sigma', t. \)
   \( \Sigma' = \{(p, I) | \Sigma \sigma \} \land \sigma' = \{(p, I) | \Sigma \sigma \} \)
   \( B' = B \lor B' = B(t' \sim B(t') + I) \)
2. \( \forall \Sigma, B, \sigma, \Sigma', B', \sigma', t. \)
   \( G_o'([B, \Sigma], \Sigma', B', \sigma') \Rightarrow \)
   \( R_o^\tau([B, \Sigma], \Sigma', B', \sigma') \)

Here we partition memory into object memory and client memory using assertion \( \rho_o \). And this definition says object rely/guarantee relations \( R_o, G_o \) respect this partition if and only if the object rely \( R_o \) holds for any change of object memory, and the object guarantee \( G_o \) guarantees non-object memory unchanged.

We instantiated the partition as \( \text{objM} \) in Fig. 20, which determines whether a location is in object memory by the location's permission.

\( \text{lifts} \) object global environment \( ge_r \) to memory with None permission:

\[ \langle ge_r \rangle(l) \triangleq \begin{cases} \text{undefined}, & \text{if } l \in \text{dom}(ge_r) \\ \text{None}, & \text{otherwise} \end{cases} \]

Definition 28 (Object module simulation).

\( \langle \text{slImp, ge}_r, \varnothing \rangle \models^\sigma (ltsO, ge_r, \tau_o) \) iff

1. \( \forall \Sigma, \sigma, \langle \text{ge}_r \rangle(l) \subseteq \Sigma \land \text{ge}_r \subseteq \Sigma \implies \)
   \( l_o((t_1 \sim \ldots \sim t_n \sim \epsilon), \tau_o) \), and
2. \( \forall \Sigma, B, \sigma, f, \tilde{v}, k. \text{if } \text{InitCore}(\pi, f, \tilde{v}) = k \land l_o((B, \pi)) \)
   \( \exists k. \text{InitCore}(y, f, \tilde{v}) = k \land (k, \Sigma) \models^\sigma \{R_{\text{tImp}}, G_0\} (k, B, \pi) \).

Here \( (k, \Sigma) \models^\sigma \{R_{\text{tImp}}, G_0\} (k, B, \pi) \) is defined as the largest relation such that whenever \( (k, \Sigma) \models^\sigma \{R_{\text{tImp}}, G_0\} (k, B, \pi) \), then all the following holds:

a. if \( (k, B(t), \sigma) \rightarrow_\delta (k', B(t'), \sigma') \) then
   \( \text{either } (k, \Sigma) \rightarrow_\delta \text{abort} \),
   or \( \exists k', \Sigma', B. \text{such that } \\
   i. (k, \Sigma) \rightarrow_\delta^* (k', B, \sigma') \\
   \text{and } \exists \Sigma' \text{. such that } \\
   ii. (k', \Sigma') \models^\sigma \{R_{\text{tImp}}, G_0\} (k', B(t \sim B'), \sigma') \).

b. if \( (k, B(t), \sigma) \rightarrow_\delta^* (k', B(t), \sigma) \) then
   \( \text{either } (k, \Sigma) \rightarrow_\delta \text{abort} \),
   or \( \exists k'. \text{ such that } \\
   i. (k, \Sigma) \rightarrow_\delta^* (k', B, \sigma) \\
   \text{and } \exists k'. \text{ such that } \\
   ii. (k', \Sigma') \models^\sigma \{R_{\text{tImp}}, G_0\} (k', B(t \sim B'), \sigma') \).

D.3 Proof of Theorem 15

Theorem 29 (Correctness with x86-TSO backend and obj).

For any \( f_1 \ldots f_n, \Gamma = \{(s_l, ge_l, \gamma_l), \ldots, (s_m, ge_m, \gamma_m)\} \) \( \langle \text{slImp, ge}_r, \varnothing \rangle \) and \( \Pi = \{(t_l, ge_l, \tau_l), \ldots, (t_m, ge_m, \tau_m)\} \), if

1. \( \forall i \in \{1, \ldots, m\}. \text{CompCert.tCode}(f_i, \gamma_i) \land \text{injective}(\varphi) \land \text{CompCert.}\varphi = \varphi \land [\varphi]_{ge_l} = ge_l \)
2. \( \text{SafeLet } \Gamma \rightarrow f_i \rightarrow \ldots \rightarrow f_n \)
3. DRF(\text{let } \Gamma \rightarrow f_i \rightarrow \ldots \rightarrow f_n)
4. \( \forall i \in \{1, \ldots, m\}. \text{ReachClose}(s_l, ge_l, \gamma_l), \) \( R_o \), and \( \text{ReachClose}(s_l, ge_l, \gamma_l) \),
5. \( \langle t_l, ge_l, \tau_l \rangle \models^0 \langle s_l, ge_l, \gamma_l \rangle \)
then \( \text{let } \Gamma \rightarrow f_i \rightarrow \ldots \rightarrow f_n \rightarrow \Pi \rightarrow f_i \rightarrow \ldots \rightarrow f_n \).

Proof. By transitivity of refinement, it suffices to prove

\( \text{let } \Gamma \rightarrow f_i \rightarrow \ldots \rightarrow f_n \rightarrow \Pi \rightarrow \Pi_{SC} \rightarrow f_i \rightarrow \ldots \rightarrow f_n \) \hspace{1cm} (D.1)
and
\( \text{let } \Pi_{SC} \rightarrow f_i \rightarrow \ldots \rightarrow f_n \rightarrow \Pi \rightarrow f_i \rightarrow \ldots \rightarrow f_n \) \hspace{1cm} (D.2)

Where \( \Pi_{SC} = \{(t_l, ge_l', \tau_l), \ldots, (t_m, ge_m', \tau_m, s_l, ge_l, \gamma_l)\} \).

(D.1) is proved by applying Theorem 19 and premise 1-4.
Towards Certified Separate Compilation for Concurrent Programs

To prove (D.2), by Lemma 30 below, it suffice to prove Safe(let \( f \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)) and DRF(let \( f \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)). The first is implied by premise 2 and (D.1); the second is implied by premise 3 and DRF preservation results as we discussed in section 5. \( \square \)

Lemma 30 (Restore SC semantics for DRF x86 programs). For any \( (\Pi_{\text{sc}} = ((t_{\text{sc}}, ge_x, \pi_x)), \ldots (t_{\text{sc}}, ge_m, \pi_m)), \Pi_{\text{ts}} = ((t_{\text{ts}}, ge_x, \pi_x)), \ldots (t_{\text{ts}}, ge_m, \pi_m)), \Pi_{\text{sc}} \) and for any \( f_1, \ldots, f_n \),

1. Safe(let \( \Pi_{\text{sc}} \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)), and
2. DRF(let \( \Pi_{\text{sc}} \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)), and
3. \( (t_{\text{ts}}, ge_x, \pi_x) \) \( \leq \) \( 0 \) \( (t_{\text{sc}}, ge_x, \pi_x) \) then

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \text{ } \Rightarrow \text{' let } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n. \]

Proof: By Lemma 31, it suffices to prove Safe(let \( \Pi_{\text{ts}} \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)), and DRF(let \( \Pi_{\text{ts}} \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)), and Safe(let \( \Pi_{\text{sc}} \) in \( f_1 \) \( \| \) \( \| \) \( f_n \)), which are exactly premises 1 and 2. \( \square \)

D.3.1 Whole-program simulation between TSO and SC

Definition 31 (Whole program simulation). For any \( f_1, \ldots, f_n, \Pi_{\text{sc}}, \Pi_{\text{ts}}, ge_x, \pi_x \),

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \text{ fit } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n \iff \forall t, B \text{ if} \]

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \text{ load } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n \text{ iff} \]

\( \exists \Sigma, T, t \text{ such that } \Pi_{\text{ts}} \subseteq \Pi_{\text{sc}}, \Sigma, T, t \text{, (B, \sigma)} \text{ then} \)

\( \Pi_{\text{ts}}, \Sigma, T, \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n \text{ load } \Pi_{\text{ts}}, \Sigma, T, t \text{, (B, \sigma)} \text{ and} \)

\( (\Pi_{\text{ts}}, \Sigma, T, t, (B, \sigma)) \text{ and} \)

\( (\Pi_{\text{sc}}, \Sigma, T, t, (B, \sigma)) \text{ \( \leq \) id } \Pi_{\text{sc}}, \Sigma, T, t, 0, 0, \Sigma). \)

Here \( W \leq \text{id } W \) is defined as the largest relation such that whenever \( W \leq \text{id } W \), then the following hold:

1. If \( W \) then there exists \( W' \) such that \( W \leq W' \) and \( W' \leq \text{id } W \).
2. If \( W \) then there exists \( W' \) such that \( W \leq W' \) and \( W' \leq \text{id } W \).
3. If \( W \) then \( W \leq \text{id } W \).
4. If \( W \) then \( W \text{ done } \).

Lemma 32 (whole program simulation implies ref.). For any \( f_1, \ldots, f_n, \Pi_{\text{sc}}, \Pi_{\text{ts}} \),

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \leq \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n \implies \text{let } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n \text{ load } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \]

D.3.2 Client simulation and composition

Definition 33 (Client module simulation).

\[ (t_{\text{sc}}, ge_x, \pi) \leq t_{\text{ts}}, ge_x, \pi \text{ iff} \]

\( \forall \Sigma, B, \sigma, f, \tilde{v}, \pi, k, \text{ if } \text{InitCore}(\pi, f, \tilde{v}, \pi) = k \text{ and } I_c(\Sigma, (B, \sigma)) \text{ then} \)

\( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ (} \sigma) \text{ is the largest relation such that whenever} \)

1. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)
2. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)
3. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \text{ fit } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n. \]

Here \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ is the largest relation such that whenever} \)

1. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)
2. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)
3. If \( (\Sigma, \sigma) \leq t_{\text{ts}}, ge_x, \pi \text{ (k, (B, \sigma))} \text{ then the following hold:} \)

\[ \text{let } \Pi_{\text{ts}} \text{ in } f_1 \| \ldots \| f_n \text{ fit } \Pi_{\text{sc}} \text{ in } f_1 \| \ldots \| f_n. \]
Proof. Trivial by definition. □

**Theorem 37** (TSO obj. compositionality).
For any $f_1, \ldots, f_n$,

$\Pi_{SC} = \{(t_{SC}, g_{E_i}, p_i), \ldots, (t_{SC}, g_{E_m}, p_m)\}, (sl_{tmp}, g_{E_{v_0}})\}$,

$\Pi_{TSO} = \{(t_{TSO}, g_{E_i}, p_i), \ldots, (t_{TSO}, g_{E_m}, p_m)\}, (t_{TSO}, g_{E_{v_0}})\}$,

$\forall i \in \{1, \ldots, m\}, (t_{SC}, g_{E_i}, p_i) \triangleright_{\Pi_{TSO}} (t_{TSO}, g_{E_i}, p_i)$,

$DRF(\Pi_{SC} \in f_1 \parallel \ldots \parallel f_n)$

$Safe(\Pi_{SC} \in f_1 \parallel \ldots \parallel f_n)$

$let \Pi_{TSO} in f_1 \parallel \ldots \parallel f_n$ 

Prove by contradiction. Assume $clientFP(\delta, \Sigma) \land conflict_c(\delta, W', t, W, B)$, we show we are able to construct SC execution with data race, which contradicts with premise 4.

By $clientFP(\delta, \Sigma) \land conflict_c(\delta, W', t, W, B)$, we know there exists $W_0, W_1, W_2, \ldots, W_N$, such that $\forall i \in \{0, \ldots, N-2\}$.

$W_i \overset{load}{\rightarrow} W_{i+1} \land \neg(clientFP(\delta, \Sigma) \land conflict_c(\delta, W_i, t, W_i, B))$

$W_{N-1} \overset{load}{\rightarrow} W_N$

$clientFP(\delta_{N-1}, \Sigma) \land conflict_c(\delta_{N-1}, W_{N-1}, t, W_{N-1}, B)$.

By $conflict_c(\delta_{N-1}, W_{N-1}, t, W_{N-1}, B)$, there exists one buffer item $(l, v)$ in buffer of some thread such that $l \in \delta_{N-1}, \Sigma \cup \delta_{N-1}, ws$. The buffer item $(l, v)$ must be inserted during the execution from $W_0$ to $W_N$, i.e., there exists $M < N$ such that $W_M.t \neq W_{N-1}.t$ and $\delta_M \sim \delta_{N-1}$ and $clientFP(\delta_{M}, \Sigma)$ and $l \in \delta_M, ws$, and $(l, v)$ not unbuffered during the execution from $W_M$ to $W_N$.

$W_0 \rightarrow \cdots \rightarrow W_M \overset{\delta_M}{\rightarrow} W_{M+1} \rightarrow \cdots \rightarrow W_{N-1} \overset{\delta_{N-1}}{\rightarrow} W_N$

Now we reorder the execution, to postpone the execution of thread $W_{M}.t$ after $W_M$, such that we could construct an execution trace as show below, such that $\forall i \in \{0, \ldots, N-2\}$.

$-(clientFP(\delta, \Sigma) \land conflict_c(\delta, W'_i, t, W'_i, B))$ and $\delta' \sim \delta''$ and $l \in \delta', ws$ and $conflict_c(\delta', t', W'_{N-1}, B)$.

Since thread local steps would not change TSO-memory (they only insert buffer items to buffer), we can safely reorder steps of different threads except unbuffer steps. To reorder normal (not unbuffer) step with unbuffer step, we apply lemma 39 to reorder non-unbuffer step with unbuffer of the same thread, and lemma 40 to reorder non-unbuffer step with unbuffer of another thread.

Since $(l, v)$ not unbuffered during the execution from $W_0$ to $W_N$, we could always apply lemma 39 to reorder step $M$ with unbuffer steps of the same thread.

To show premise of lemma 40 hold, it suffice to prove $\exists \Pi_{SC} \Pi_{TSO}$ $clientFP(\delta, \Sigma) \land conflict_c(\delta, W_i, t, W_i, B)$.

By definition of $R$, we could do induction on $N$ and apply client or object local simulations to construct SC execution $\Sigma_1, \ldots, \Sigma_N$ such that $\forall i \in \{0, \ldots, N-1\},$ $R(\Pi_{SC} \Pi_{TSO} \in f_1 \parallel \ldots \parallel f_n)$, we could do induction on $N$ and apply client or object local simulations to construct SC execution $\Sigma_1, \ldots, \Sigma_N$ such that $\forall i \in \{0, \ldots, N-1\},$ $R(\Pi_{SC} \Pi_{TSO} \in f_1 \parallel \ldots \parallel f_n)$, we could do induction on $N$ and apply client or object local simulations to construct SC execution $\Sigma_1, \ldots, \Sigma_N$ such that $\forall i \in \{0, \ldots, N-1\}$.

$W_0 \rightarrow \cdots \rightarrow W_M \rightarrow \cdots \rightarrow W_{N-1} \rightarrow W_N$

By definition of $R$ and $I$, $\forall i < N$, $\negbufConflict(W_i, B, \Sigma)$.

Now we have reordered the execution to $W'_0, \ldots, W'_{N-1}$.

We do the above induction again on $W'_0, \ldots, W'_{N-1}$ and construct SC execution $\Pi'_1, \ldots, \Pi'_{N-1}$ such that $\forall i \in \{0, \ldots, N-1\},$ $R(\Pi'_1 \Pi'_{N-1} \in f_1 \parallel \ldots \parallel f_n)$, as shown below.

$W'_0 \rightarrow \cdots \rightarrow W'_M \rightarrow \cdots \rightarrow W'_{N-1} \rightarrow W_N$

By definition of $R$ and $I$, $\forall i < N$, $\negbufConflict(W'_i, B, \Sigma)$.
Towards Certified Separate Compilation for Concurrent Programs

By $\mathcal{R}(\mathcal{W}_N, W', W_{N-1}, R_o, G_o, L_o)$, clientFP$((\delta', \Sigma))$, and clientFP$((\delta', \Sigma))$, we can apply client simulation and there exists $\mathcal{W}_N', \mathcal{W}_N''$, $\Lambda'$, $\Lambda''$ such that $\Lambda' = \delta'$ and $\Lambda''.rs \cup \delta''.ws \subseteq \Lambda''.rs \cup \Lambda'''.ws$ and

$$
\xymatrix{
\mathcal{W}_N' \ar[r] & \mathcal{W}_N'' \\
\Lambda' \ar[u]^t' \ar[r] & \Lambda'' \ar[u]_{t''}
}$$

Since $l \in \delta''.ws$ and $l \in \delta'''.ws \cup \delta'''.ws$, we have $\Lambda' \sim \Lambda''$, which indicates a racing execution that contradicts with premise 4.

\[\square\]

**Lemma 39** (Reordering unbuffer and write buffer step of the same thread).

\[\forall \mathcal{b}, \sigma, \delta, l, v, l', v'. \]
\[
\text{if } (\mathcal{b}, \sigma) \vdash_{\delta \tau_{\text{ts}}}(\mathcal{b} \ll(l', v'), \sigma), \]
\[
\text{then } (\mathcal{b}, \sigma) \vdash_{\delta \tau_{\text{ts}}}(\mathcal{b} \ll(l', v'), \sigma) \]
\[
\text{Proof:} \text{ Trivial by x86-TSO semantics.} \]

\[\square\]

**Lemma 40** (Reordering unbuffer and write buffer step of different threads).

\[\forall \mathcal{b}, \sigma, \delta, l, v, l', v'. \]
\[
\text{if } (\mathcal{b}, \sigma) \vdash_{\delta \tau_{\text{ts}}}(\mathcal{b} \ll(l', v'), \sigma) \]
\[
\text{and } l \neq l', \]
\[
\text{then } (\mathcal{b}, \sigma) \vdash_{\delta \tau_{\text{ts}}}(\mathcal{b} \ll(l', v'), \sigma) \]
\[
\text{Proof:} \text{ It suffice to guarantee } l \notin \delta.\mathcal{r} \cup \delta.\mathcal{w}. \text{By x86-TSO semantics, a step with non-empty read and write footprint must have the same read/write set, i.e. } \delta.\mathcal{r} = \delta.\mathcal{w} = \{l\}. \text{And we have } l \notin \delta.\mathcal{r} \cup \delta.\mathcal{w} \text{ by premise.} \]

\[\square\]

**Definition 41** (whole program simulation invariant).

$\mathcal{R}(\mathcal{G}, \mathcal{F}_s, T, t, d, \Sigma), (\Pi, \mathcal{T}, T, t, B, \sigma), R_o, G_o, l_o)$ iff

1. $t = t', d = 0$, and $\mathcal{F}_s = \mathcal{F}$, and $l_o(\Sigma, (B, \sigma))$, and $l_o(\Sigma, (B, \sigma))$,

2. $\forall t_1, (T(t_1), \Sigma) = (T(t_1), B, \sigma)$, and

3. $\neg T_1(\mathcal{F}_s, T, t, d, \Sigma) \Rightarrow \text{Race and}$

$\neg T_1(\mathcal{F}_s, T, t, d, \Sigma) \Rightarrow \text{abort.}$

Here $(T(t), \Sigma) \approx (T(t), B, \sigma)$ iff

if $T(t) = (F_1, k_1) :: \ldots :: (F_k, k_k) :: \varepsilon$

and $T(t) = (F_1, k_1) :: \ldots :: (F_k, k_k) :: \varepsilon$,

then

1. $k = k'$, and

2. $F_j = F_j$ for all $j = 1 \ldots k$, and

3. $(k_j, \Sigma) \varepsilon_{t_1, C_{\text{ts}}}(k_j, (B, \sigma))$ for all $j > 1$, and

4. $(k_1, \Sigma) \varepsilon_{t_1, C_{\text{ts}}}(k_1, (B, \sigma))$ or

$(k_1, \Sigma) \varepsilon_{t_1, C_{\text{ts}}}(k_1, (B, \sigma)).$

\[\text{D.3.4 x86 assembly module client simulation}

\[\text{Lemma 42 (Client module simulation).} \]

For any $ge, \pi, t, (H_{\text{tsso}}.sc, ge, \pi) \varepsilon_{t_1, C_{\text{ts}}}(t_{\text{tsso}}, ge, \pi)$

\[\text{Proof.} \text{ Since client code are identical, and CompCert memory model guaranteed x86-SC semantics to abort when accessing memory location with None permission (i.e., objm), it suffice to prove if } l_c(\Sigma, (B, \sigma)) \text{ then reading/writing the same location } l \text{ from/to } \Sigma \text{ or } (B, \sigma) \text{ yields results related by } l_c \text{ and } G_c, \text{ if } l \text{ does not conflict with } B. \text{ i.e.,}$

1. $\forall l, t, \neg \text{obj}(l, \Sigma) \Rightarrow \Sigma(l) = \text{apply buffer}(B(t), \sigma)(l)$, and

2. $\forall l, t, \neg \text{obj}(l, \Sigma) \wedge \neg \text{conflict}_c((0, \{l\}), t, B)$

$\Rightarrow l_c(\Sigma(\nu \sim v), (B(t \sim B(t) ++(l, v)), \sigma))$

\[
\wedge C_1^c(\Sigma, (B, \sigma)), \Sigma(\nu \sim v), (B(t \sim B(t) ++(l, v)), \sigma))
\]

1 is obvious by definition of $l_c$.

For 2, to show $l_c(\Sigma(\nu \sim v), (B(t \sim B(t) ++(l, v)), \sigma))$, it suffices to prove $\neg \text{buffer}(B(t \sim B(t) ++(l, v)), \sigma))$, which is obvious by $\neg \text{conflict}_c((0, \{l\}), t, B)$.

$C_1^c(\Sigma, (B, \sigma)), (\nu \sim v), (B(t \sim B(t) ++(l, v)), \sigma))$. The latter is also obvious by $\neg \text{obj}(l, \Sigma)$. \[\square\]

\[\text{D.4 Spinlock spec. and impl. satisfies our obj. correctness}

We prove the lock specification and implementation in Fig. 10 satisfies our object correctness, i.e., $(H_{\text{tsso}}.g_{l_{\text{lock}}}.\pi_{\text{lock}}) \varepsilon_{t_1, C_{\text{ts}}}(s_{k_{\text{imp}}}, g_{l_{\text{lock}}}.\pi_{\text{lock}})$.

\[\text{Lemma 43 (Lock impl. correctness).} \]

$(H_{\text{tsso}}.g_{l_{\text{lock}}}.\pi_{\text{lock}}) \varepsilon_{t_1, C_{\text{ts}}}(s_{k_{\text{imp}}}, g_{l_{\text{lock}}}.\pi_{\text{lock}})$

\[\text{Proof.} \text{ To prove object correctness, the key is to find correct } R_o, G_o, l_o. \text{ For the spinlock spec. and impl., we define } R_o, G_o, l_o \text{ in Fig. 21, where } L \text{ is the lock variable.} \]

The invariant $l_o$ says the lock variable has three possible states, which is

1. $l_{\text{available}}: L \text{ in both SC and TSO memory is available, and there is no buffered write to } L$

2. $l_{\text{unavailable}}: L \text{ in both SC and TSO memory is unavailable, and there is no buffered write to } L$

3. $l_{\text{buffered}}: L \text{ is available in SC memory, but unavailable in TSO memory and there is no buffered write to } L$

It is obvious that RespectPartition$_{\text{v}}(R_o, G_o, \text{objm})$ and Sta($l_o, R^o$) and $\forall t_2, G^o \Rightarrow R^o$.

The proof of lock spec./impl. has the simulation $(s_{k_{\text{imp}}}, g_{l_{\text{lock}}}.\pi_{\text{v}}) \varepsilon_{t_1, C_{\text{ts}}}(H_{\text{tsso}}.g_{l_{\text{lock}}}.\pi_{\text{v}})$ is also straightforward, and is mechanized in Coq. \[\square\]
\( I_o \) \( \overset{\text{def}}{=} \) \( I_{\text{available}} \lor I_{\text{buffered}} \lor I_{\text{unavailable}} \)

\( G_o^l \) \( \overset{\text{def}}{=} \) \( G_{\text{lock}}^l \lor G_{\text{unlock}}^l \lor \text{Id} \)

\( R_o^l \) \( \overset{\text{def}}{=} \) \( \left( \lor \forall t' L_{t'} G_{t'}^l \right) \lor R_{\text{buffer}} \)

\( I_{\text{available}}(\Sigma, (B, \sigma)) \) \( \overset{\text{def}}{=} \) \( \Sigma(L) = (1, \text{None}) \land \sigma(L) = 1 \land (\forall t . L \notin \text{dom}(B(t))) \)

\( I_{\text{buffered}}(\Sigma, (B, \sigma)) \) \( \overset{\text{def}}{=} \) \( \Sigma(L) = (1, \text{None}) \land \sigma(L) = 0 \land \exists t, \text{buffered_unlock}(t, B) \)

\( I_{\text{unavailable}}(\Sigma, (B, \sigma)) \) \( \overset{\text{def}}{=} \) \( \Sigma(L) = (0, \text{None}) \land \sigma(L) = 0 \land (\forall t . L \notin \text{dom}(B(t))) \)

\( \text{buffered_unlock}(t, B) \) \( \overset{\text{def}}{=} \) \( B(t) = b \leftrightarrow (L, 1): b' \land L \notin \text{dom}(b) \cup \text{dom}(b') \land \forall t' \neq t. L \notin \text{dom}(B(t')) \)

\( \text{Id}((\Sigma, (B, \sigma)), (\Sigma', (B', \sigma'))) \) \( \overset{\text{def}}{=} \) \( \Sigma = \Sigma' \land \sigma = \sigma' \land B = B' \)

\( G_{\text{lock}}^l((\Sigma, (B, \sigma)), (\Sigma', (B', \sigma'))) \) \( \overset{\text{def}}{=} \) \( \Sigma(L) = (1, \text{None}) \land \sigma(L) = 1 \land \Sigma' = \Sigma(L \leftrightarrow (0, \text{None})) \land \sigma' = \sigma[L \leftrightarrow 0] \land B = B' \)

\( G_{\text{unlock}}^l((\Sigma, (B, \sigma)), (\Sigma', (B', \sigma'))) \) \( \overset{\text{def}}{=} \) \( \Sigma(L) = (0, \text{None}) \land \sigma(L) = 0 \land \Sigma' = \Sigma(L \leftrightarrow (1, \text{None})) \land \sigma = \sigma' \land B' = B[t \leftrightarrow B(t) \leftrightarrow (L, 1)] \)

\( R_{\text{buffer}}((\Sigma, (B, \sigma)), (\Sigma', (B', \sigma'))) \) \( \overset{\text{def}}{=} \) \( I_{\text{buffered}} \land I_{\text{available}} \land I_{\text{available}} \land I_{\text{buffered}} \land I_{\text{buffered}} \land I_{\text{buffered}} \land I_{\text{unavailable}} \land I_{\text{unavailable}} \)

**Figure 21.** \( R_o, G_o, I_o \) instantiations for spinlock
Towards Certified Separate Compilation for Concurrent Programs

defines the module-local downward simulation which supports escape of stack pointers. stack pointers to the corresponding target-level escaped stack pointers. may change during the executions of source and target code. The function \( f \) may change the preemptive and the non-preemptive global semantics is not changed.

Our way to support stack pointer escape is actually very similar to the approach in Compositional CompCert, which is a function mapping block IDs to addresses.

E.1 Revised Definitions and Key Lemmas

In this section, we give the revised definitions of simulations, and the on-paper proofs of the key lemmas. The revised simulation is still parameterized with \( \mu \). We give the useful auxiliary definitions in Fig. 23. Most definitions are similar to their counterparts in Fig. 8. Definition 44 defines the module-local downward simulation which supports escape of stack pointers.
\[ \text{wf}(\mu, \Sigma, \sigma) \iff \text{injective}(\mu, f, \Sigma) \land (\mu. S = \text{dom}(\mu.f)) \land (\mu. S \subseteq \text{dom}(\Sigma)) \land (\mu. f \{ \mu. S \} \subseteq \mu. S) \land (\mu. f \{ \text{locs}(\Sigma) \} \subseteq \text{locs}(\sigma)) \]

\[ \text{fpmatch}(\mu, (\Lambda, \Sigma), (\delta, F)) \iff \text{wf}(\mu, \Sigma, \sigma) \land (\mu.f \{ \mu. S \cap F \} \subseteq F) \land (\mu.f \{ \mu. S - F \} \cap F = \emptyset) \]

\[ \text{LShare}(\mu, \Sigma) \overset{\text{def}}{=} \mu.f \{ \text{locs}(\Sigma) \cap ||\mu. S|| \}
\]

\[ \text{HGuar}(\mu, \Lambda, \Sigma) \iff G(\Lambda, \Sigma, ||\mu. S||)
\]

\[ G(\Lambda, \Sigma, F) \iff \text{dom}(\Sigma) \subseteq \text{LShare}(\mu, \Sigma) \]

\[ \text{Rely}(\mu, \mu', (\Sigma, \Sigma', F), (\sigma, \sigma', F)) \iff R(\Sigma, \Sigma', F, ||\mu. S||) \land R(\sigma, \sigma', F, \text{LShare}(\mu, \Sigma)) \land \text{Evolve}(\mu, \mu', \Sigma', \sigma', F, F)
\]

\[ \text{Evolve}(\mu, \mu', \Sigma', \sigma', F, F) \iff \text{Evolve}(\mu, \mu', \Sigma', \sigma, F) \land (\mu.S - \mu. S \subseteq \Sigma)
\]

\[ \text{Inv}(f, \Sigma, \sigma) \iff \forall b, n, b', n', ((b, n) \in \text{locs}(\Sigma)) \land (f \{ (b, n) \} = (b', n')) \implies ((b', n') \in \text{locs}(\sigma)) \land (\Sigma(b(n)) \overset{f}{\to} \sigma(b')(n'))
\]

\[ \forall v_1 \overset{f}{\to} v_2 \iff (v_1 \notin \Sigma) \lor (v_1 = v_2) \lor v_1 \in \Sigma \land v_2 \in \Sigma \land f(v_1) = v_2
\]

\[ \text{injective}(f, \Sigma) \iff \forall l_1, l_2, l_1' \neq l_2 \land l_1, l_2 \in \text{locs}(\Sigma) \land l_1 + l_2 \in \text{locs}(\Sigma) \land f \{ l_1 \} = l_2' \wedge f \{ l_2 \} = l_2' \implies l_1' \neq l_2'
\]

\[ \text{injective}(f) \iff \forall l_1, l_2, l_1' \neq l_2 \land l_1 + l_2 \in \text{locs}(\Sigma) \land f \{ l_1 \} = l_2' \wedge f \{ l_2 \} = l_2' \implies l_1' \neq l_2'
\]

\[ \text{initM}(\varphi, \text{ge}, \Sigma, \sigma) \iff \text{ge} \subseteq \text{locs}(\Sigma) \land \text{closed}((\text{dom}(\Sigma), \Sigma) \land \text{locs}(\sigma) = \varphi \{ \text{locs}(\Sigma) \} \land \text{dom}(\sigma) = \varphi \{ \text{dom}(\Sigma) \} \land \text{Inv}(\varphi, \Sigma, \sigma)
\]

\[ f \{ s \} \overset{\text{def}}{=} \{ b' \mid \exists h. (h \in S) \land f(h) = (b', \_))
\]

\[ f \{ l \} \overset{\text{def}}{=} (b', n + n') \iff l = (b, n) \land f(b) = (b', n')
\]

\[ f \{ ws \} \overset{\text{def}}{=} \{ l' \mid \exists h. (l \in ws) \land f(l) = l'\}
\]

\[ f \{ b \} \overset{\text{def}}{=} \{ b(b) \mid b \in (S \cap \text{dom}(f))\}
\]

\[ f_2 \circ f_1 \overset{\text{def}}{=} \{ (b_1, b_3, n_2 + n_3) \mid \exists h_2. f_1(b_1) = (b_2, n_2) \land f_2(b_2) = (b_3, n_3)\}
\]

\[ \text{closed}(S, \Sigma) \iff c(S, \Sigma) \subseteq S
\]

\[ c(S, \Sigma) \overset{\text{def}}{=} \bigcup_k c_k(S, \Sigma), \text{ where } c_k(S, \Sigma) \text{ is inductively defined:}
\]

\[ c_0(S, \Sigma) \overset{\text{def}}{=} S
\]

\[ c_k+1(S, \Sigma) \overset{\text{def}}{=} \{ b' \mid \exists b, n, n'. (b \in c_k(S, \Sigma)) \land \Sigma(b)(n) = (b', n')\}
\]

**Figure 23.**Footprint Matching and Evolution of $\mu$ in Our Simulation
Towards Certified Separate Compilation for Concurrent Programs

The key to support stack pointer escape is to describe the possible changes of \( \mu \) (i.e., \( S, S \) and the mapping \( f \) between them) during executions. In Definition 44, we are allowed to pick a new \( \mu' \) at an interaction point (see case 2). The new \( \mu' \) and the earlier \( \mu \) should be related by EvolveG(\( \mu, \mu', \ldots \)), which is defined in Fig. 23. In its definition, Evolve(\( \mu, \mu', \ldots \)) requires that \( \mu' \) is well-formed, \( \mu', \Sigma \) and \( \mu', \Sigma \) are closed and the mapping \( \mu', f \) can only be enlarged. The condition (\( \mu', S - \mu, S \subseteq \mathbb{F} \)) says that a module cannot leak out arbitrary addresses, instead it can only leak out pointers pointing to its own stack.

Similarly, the environment step (see case 3 in Definition 44) may also modify \( \mu \), but the updates are confined using EvolveR(\( \mu, \mu', \ldots \)), which is also defined in Fig. 23. Here the condition (\( \mu, S - \mu, S \subseteq \mathbb{F} = \emptyset \)) says that it is impossible for the environment to leak out pointers pointing to the stack of the current module.

Other parts in the simulation definition (Definition 44) are very similar to Definition 2. In particular, the silent step case (case 1) is not changed (though we slightly modified the presentation), since the shared state updates are not exposed to the environment between two interaction points.

**Definition 44 (Module-Local Downward Simulation).**

\((sl, ge, y) \triangleq_\varphi (tl, ge', \pi)\) if

for all \( f, k, \Sigma, \sigma, F, \) and \( \mu = (\text{dom}(\Sigma), \text{dom}(\sigma), \varphi) \) if sl.InitCore(y, f) \( \equiv k \), \( \varphi(ge) = ge' \), initM(\( \mu, ge, \Sigma, \sigma \)), and \( F \cap \text{dom}(\Sigma) = F \cap \text{dom}(\sigma) = \emptyset \), then there exists \( i \in \text{index} \) and \( k \) such that \( tl.\text{InitCore}(\pi, f) = k \), and

\((F, (k, \Sigma), \text{emp}, \bullet) \triangleq_\varphi (F, (k, \sigma), \text{emp}, \bullet)\).

Here we define \((F, (k, \Sigma), \Delta_0, \beta) \triangleq_\mu^k (F, (k, \sigma), \delta_0, \beta)\) (where the bit \( \beta \) is either \( o \) or \( \bullet \)) as the largest relation such that, whenever \((F, (k, \Sigma), \Delta_0, \beta) \triangleq_\mu^k (F, (k, \sigma), \delta_0, \beta)\), then \( \text{wf}(\mu, (\Sigma, F)) \) and the following are true:

1. \( \forall k', \Sigma', \Delta. \) if \( \beta = o \) and \( F \vdash (k, \Sigma) \overset{\tau}{\rightarrow} (k', \Sigma') \) and \( \text{HGuar}(\mu, \Delta_0 \cup \Delta, \mathbb{F}) \), then one of the following holds:
   a. there exists \( j \) such that
      i. \( j < i \), and
      ii. \((F, (k', \Sigma'), \Delta_0 \cup \Delta, \varnothing) \triangleq_\mu^j (F, (k, \sigma), \delta_0, \varnothing)\);
   b. there exist \( k', \sigma', \delta \) and \( j \) such that
      i. \( F \vdash (k, \sigma) \overset{\tau}{\rightarrow}^\delta (k', \sigma') \), and
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)) \), and
      iii. \((F, (k', \Sigma'), \Delta_0 \cup \Delta, \varnothing) \triangleq_\mu^j (F, (k', \sigma'), \delta_0 \cup \delta, \varnothing)\);
2. \( \forall k', \Sigma', \Delta. \) if \( \beta = o, i \neq \tau \) and \( F \vdash (k, \Sigma) \overset{\tau}{\rightarrow} (k', \Sigma') \) and \( \text{HGuar}(\mu, \Delta_0, \mathbb{F}) \),
   then there exist \( \kappa', \delta, \sigma', \kappa', \mu' \) and \( j \) such that
   a. \( F \vdash (k', \sigma') \overset{\tau}{\rightarrow} (k', \sigma') \), and
      \( F \vdash (k', \sigma') \overset{\tau}{\rightarrow} (k', \sigma') \), and
   b. \( \text{FPmatch}(\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F)) \), and
   c. \( \text{EvolveG}(\mu, \mu', \Sigma, \sigma', F, \mathbb{F}) \), and
   d. \((F, (k', \Sigma), \text{emp}, \bullet) \triangleq_\mu^j (F, (k', \sigma'), \text{emp}, \bullet)\);
3. \( \forall \sigma', \Sigma', \mu'. \) if \( \beta = \bullet \) and \text{Rely}(\( \mu, \mu' \), \( (\Sigma, \Sigma', F) \) \( (\sigma, \sigma', F) \))
   then there exists \( j \) such that
   \((F, (k, \Sigma), \text{emp}, \varnothing) \triangleq_\mu^j (F, (k, \sigma'), \text{emp}, \varnothing)\).

**Lemma 45 (Transitivity).**

If \((sl, ge, y) \triangleq_\varphi (sl_1, ge_1, y_1), (sl_1, ge_1, y_1) \triangleq_\varphi (tl, ge', \pi)\), \( \varphi(ge) = ge' \) and \( \forall \varphi \subseteq \varphi \text{ dom}(\varphi) \), then \((sl, ge, y) \triangleq_{\varphi \circ \varphi} (tl, ge', \pi)\).

**Compositionality and the non-preemptive global downward simulation.** To support stack pointer escape, the whole program simulations also allow one to pick a new \( \mu' \) at interaction points (see case 3 in Definition 46). The new definition of reach-close modules (see Definition 47) also allows \( S \) to change, which is similar to the reach-closed definition in Compositional CompCert.

**Definition 46 (Whole-Program Downward Simulation).**

Let \( \hat{P} = \text{letLin} \ f_1 | \ldots | f_m, \hat{F} = \text{letLin} \ f_1 | \ldots | f_m, \Gamma = \{(sl_1, ge_1, y_1), \ldots, (sl_m, ge_m, y_m)\} \) and \( \Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\} \).

\( ^3 \beta \) is introduced to simplify the presentation only. It is used to make the rely step explicit (i.e. take the clause 3 out from the clause 2 in Def. 44), so the proofs can be cleaner.

29
Definition 47 (Reach Closed Module). ReachClose(sl, ge, γ) \iff, for all f, k, Σ, F and S, if sl.InitCore(γ, f) = k, S = dom(Σ), ge ⊆ S, F ∩ S = ∅, and closed(S, Σ), then RC(F, S, (κ, Σ), ●).

Here RC is defined as the largest relation such that, whenever RC(F, S, (κ, Σ), β), then all the following holds:
1. for all κ′, Σ′ and Δ, if β = o and F \vdash (k′, Σ′) \vdashΔ (k, Σ′), then G(Δ, F, S) and RC(F, S, (κ′, Σ′), o).
2. for all κ′ and i, if β = o and i \neq r and F \vdash (κ, Σ) \vdashΔ (k′, Σ), then there exists S′ such that S′ = cl(S, Σ), and S′ ⊊ S ⊊ F, and RC(F, S′, (k′, Σ), ●).
3. for all Σ′ and S′, if β = ● and R(Σ, Σ′, F, S) and S′ = cl(S, Σ′) and (S′ \setminus S) \cap F = ∅, then RC(F, S′, (κ, Σ′), o).

Lemma 48 (Compositionality).
For any f_1, \ldots, f_n, φ, Γ = \{(sl_1, ge_1, γ_1), \ldots, (sl_m, ge_m, γ_m)\}, Π = \{(tl_1, ge_1, π_1), \ldots, (tl_m, ge_m, π_m)\}, if
\[ \forall i \in \{1, \ldots, m\}. wd(sl_i) \land wd(tl_i) \land \text{ReachClose}(sl_i, ge_i, γ_i) \land (sl_i, ge_i, γ_i) \preceq_φ (tl_i, ge_i, π_i), \]
then
\[ \text{let } Γ \text{ in } f_1 \mid \ldots \mid f_n \text{ let } Π \text{ in } f_1 \mid \ldots \mid f_n. \]

Flip of the non-preemptive global simulation.

Definition 49 (Whole-Program Upward Simulation). Let \[ \tilde{P} = \text{let } f_1 | \ldots | f_m \text{ in } f_1 | \ldots | f_n \text{, } \tilde{P} = \text{let } Π | \text{ in } f_1 | \ldots | f_n | \text{, } \tilde{P} = \text{let } Π | \text{ in } f_1 | \ldots | f_n. \]

\[ \tilde{P} \preceq_φ \tilde{P} \iff \forall i \in \{1, \ldots, m\}. φ\langle ge_i \rangle = ge_i, \]
for any Σ, σ, μ, \tilde{W}, if initM(φ, \cup ge_i, Σ, σ), μ = (dom(Σ), dom(σ), φ), and (\tilde{P}, σ) \rightarrow load \tilde{W}, then there exists \tilde{W}, i \in index such that (\tilde{P}, σ) \rightarrow load \tilde{W} and (\tilde{W}, emp) \preceq_μ (\tilde{W}, emp).

Here we define (\tilde{W}, \delta_0) \preceq_μ (\tilde{W}, \delta_0) as the largest relation such that whenever (\tilde{W}, \delta_0) \preceq_μ (\tilde{W}, \delta_0), then the following are true:
1. \[ \tilde{W}.t = \hat{W}.t, \text{ and } \tilde{W}.d = \hat{W}.d, \text{ and } wf(μ, \tilde{W}, Σ, \tilde{W}.σ); \]
Towards Certified Separate Compilation for Concurrent Programs

2. \( \forall \mathcal{W}', \delta. \) if \( \mathcal{W} \xrightarrow{\delta} \mathcal{W}' \), then one of the following holds:
   a. there exists \( j \) such that
      i. \( j < i \), and
      ii. \( (\mathcal{W}', \delta_0 \cup \delta) \leq_{\mu} (\mathcal{W}, \Delta_0) \);
   b. there exist \( \mathcal{W}', \Delta \) and \( j \) such that
      i. \( \mathcal{W} \xrightarrow{\Delta} \mathcal{W}' \), and
      ii. FPmatch\( (\mu, (\Delta_0 \cup \Delta, \mathcal{W}, \Sigma), (\delta_0 \cup \delta, \text{curF}(\mathcal{W}))) \), and
      iii. \( (\mathcal{W}', \delta_0 \cup \Delta) \leq_{\mu} (\mathcal{W}, \Delta_0 \cup \Delta) \);
   c. \( \forall \mathcal{T}' \), \( d', \sigma', o. \) if \( o \neq \tau \) and \( \exists \mathcal{T}' \). \( \mathcal{W} \xrightarrow{\sigma_{\text{emp}}} (\mathcal{T}', t', d', \sigma') \), then there exist \( \mathcal{W}', \Delta, \mu', \mathcal{T}', \Sigma' \) and \( j \) such that for any \( t' \in \text{dom}(\mathcal{T}') \),
      we have
      a. \( \mathcal{W} \xrightarrow{\Delta} \mathcal{W}' \), and \( \mathcal{W}' \xrightarrow{\sigma_{\text{emp}}} (\mathcal{T}', t', d', \Sigma') \), and
      b. FPmatch\( (\mu, (\Delta_0 \cup \Delta, \mathcal{W}, \Sigma), (\delta_0, ((\mathcal{W}, T)(\mathcal{W}, t)), F)) \), and
      c. \( ((\mathcal{T}', t', d', \sigma'), \text{emp}) \leq_{\mu'} ((\mathcal{T}', t', d', \Sigma'), \text{emp}) \);
   d. if \( \mathcal{W} \xrightarrow{\tau_{\text{emp}}} \text{done} \), then there exist \( \mathcal{W}' \) and \( \Delta \) such that
      a. \( \mathcal{W} \xrightarrow{\Delta} \mathcal{W}' \), and \( \mathcal{W}' \xrightarrow{\tau_{\text{emp}}} \text{done} \), and
      b. FPmatch\( (\mu, (\Delta_0 \cup \Delta, \mathcal{W}, \Sigma), (\delta_0, \text{curF}(\mathcal{W}))) \).

\textbf{Lemma 50 (Flip).}
For any \( f_1, \ldots, f_n, \varphi, \Gamma = \{(s_l, g_{e_l}, y_l), \ldots, (s_m, g_{e_m}, y_m)\} \), \( \Pi = \{(t_l, g_{e_l}', \pi_1), \ldots, (t_m, g_{e_m}', \pi_m)\} \), if \( \forall i. \text{det}(t_i) \) and
\[
\text{let } \Gamma \text{ in } f_1 \ | \ \ldots \ | \ f_m \leq_{\varphi} \text{ let } \Pi \text{ in } f_1 \ | \ \ldots \ | \ f_m ,
\]
then
\[
\text{let } \Pi \text{ in } f_1 \ | \ \ldots \ | \ f_m \leq_{\varphi} \text{ let } \Gamma \text{ in } f_1 \ | \ \ldots \ | \ f_m .
\]

\textbf{NPDRF preservation.}

\textbf{Lemma 51 (NPDRF Preservation).}
For any \( \mathcal{P}, \mathcal{P}, \varphi, \Sigma \) and \( \sigma \), if \( \mathcal{P} \leq_{\varphi} \mathcal{P} \), \( \text{initM}(\varphi, \mathcal{GE}(\mathcal{P}, \Gamma), \Sigma, \sigma) \), and \( \text{NPDRF}(\mathcal{P}, \Sigma) \), then \( \text{NPDRF}(\mathcal{P}, \sigma) \).

\textbf{E.2 Proof of Transitivity of Simulation (Lemma 45)}
For all \( f, k, \Sigma, \varphi, F, S \) and \( \text{S} = \text{dom}(\Sigma) \), \( \text{S} = \text{dom}(\sigma) \), and \( \mu = (\Sigma, S, (\varphi ' \circ \varphi)_{|S}) \), if \( \text{sl} \text{InitCore}(\gamma, f) = k \), \( (\varphi ' \circ \varphi) \| \text{ge} \| = \text{ge}' \), \( \text{initM}(\varphi '\circ \varphi, \text{ge}, \Sigma, \sigma) \), and \( F \cap \text{dom}(\Sigma) = F \cap \text{dom}(\sigma) = \emptyset \), we know \( S = (\varphi ' \circ \varphi) \{S\} \). Let \( S_1 = \varphi ' \{S\} \) and \( L_1 = \varphi ' \{\text{loc}(\Sigma)\} \). We know
\[
\text{S}_1 \subseteq \text{dom}(\varphi ') , \ \varphi ' \{S_1\} = (\varphi ' \circ \varphi) \{S\} = \text{dom}(\sigma) , \ \text{locs}(\sigma) = \varphi ' \{L_1\} .
\]
Since \( \text{inv}(\varphi ' \circ \varphi, \Sigma, \sigma) \), we know there exists \( \Sigma_1 \) such that
\[
\text{dom}(\Sigma_1) = S_1 , \ \text{locs}(\Sigma_1) = L_1 , \ \text{inv}(\varphi, \Sigma, \Sigma_1) , \ \text{inv}(\varphi, \Sigma_1, \sigma) .
\]
Since \( \varphi ' \| \text{ge} \| = \text{ge}' \) and \( \text{dom}(\varphi ' \circ \varphi) \subseteq \text{dom}(\varphi) \), we know
\[
\text{ge} \subseteq \text{locs}(\Sigma) , \ \text{dom}(\Sigma) = S \subseteq \text{dom}(\varphi) , \ \text{ge}' \subseteq \text{locs}(\Sigma_1) , \ \text{dom}(\Sigma_1) = S_1 \subseteq \text{dom}(\varphi ') .
\]
Pick \( F_1 \) such that \( F_1 \cap S_1 = \emptyset \). Let \( \mu_1 = (S_1, \Sigma_1, \varphi ' |_S) \) and \( \mu_2 = (S_1, \varphi ' \{S_1\}, \varphi ' |_{S_1}) \). From \( (s_l, g_{e_l}, \gamma) \leq_{\varphi} (s_l, g_{e_l}, y_l) \), we know there exists \( k_1 \) such that \( \text{sl}_1 \text{InitCore}(y_1, f) = k_1 \) and there exists \( i_1 \) in index such that
\[
(F, (k_1, \Sigma_1), \text{emp}, o) \leq_{\mu_1} i_1 (F_1, (k_1, \Sigma_1), \text{emp}, o) .
\]
Thus we know
\[
S_1 = \text{cl}(S_1, \Sigma_1) .
\]
From \( (s_l, g_{e_l}, y_1) \leq_{\varphi} (t_l, g_{e_l}', \pi) \), we know there exists \( \kappa \) such that \( t_l \text{InitCore}(\pi(f) = \kappa \) and there exists \( i_2 \) in index such that
\[
(F_1, (k_1, \Sigma_1), \text{emp}, o) \leq_{\mu_1} i_2 (F, (k, \sigma), \text{emp}, o) .
\]
By Lemma 52, let \( i = (i_2, i_1) \), we know
\[
(F, (k, \Sigma), \text{emp}, o) \leq_{\mu} i (F, (k, \sigma), \text{emp}, o) .
\]
Thus we are done.
Lemma 52. If

1. \( (F, (k, \Sigma), \Lambda_0, \beta) \preceq_{\mu_1}^i (F_1, (k_1, \Sigma_1), \Lambda_{10}, \beta) \);
2. \( (F_1, (k_1, \Sigma_1), \Lambda_{10}, \beta) \preceq_{\mu_1}^{i_j} (F, (k, \sigma), \delta_0, \beta) \);
3. \( \mu_1 = (S, S_1, f_1), \mu_2 = (S_1, S, f_2), \mu = (S, S, f_2 \circ f_1), i = (i_2, i_1) \);
4. if \( \beta = \bullet \), then \( \text{Inv}(f_1, \Sigma, \Sigma_1) \) and \( \text{Inv}(f_2, \Sigma_1, \sigma) \).

then \( (F, (k, \Sigma), \Lambda_0, \beta) \preceq_{\mu_1}^i (F, (k, \sigma), \delta_0, \beta) \).

Proof. By co-induction. From the premises, we know

\[ \text{wf}(\mu_1, (\Sigma, F, (\Sigma_1, F_1)), \ S \subseteq \text{dom}(\Sigma), \ S_1 \subseteq \text{dom}(\Sigma_1), \ \text{wf}(\mu_2, (\Sigma_1, F_1), (\sigma, F)). \]

Thus we know

\[ \begin{align*}
\text{injective}(f_1, \Sigma), & \quad S = \text{dom}(f_1), \quad f_1 \{S\} \subseteq S_1, \quad f_1 \{S \cap F\} \subseteq F_1, \\
\text{injective}(f_2, \Sigma_1), & \quad S_1 = \text{dom}(f_2), \quad f_2 \{S_1\} \subseteq S, \quad f_2 \{S_1 \cap F_1\} \subseteq F, \quad f_2 \{S_1 - F_1\} \cap F = \emptyset.
\end{align*} \]

From injective\( (f_1, \Sigma) \) and \( \text{injective}(f_2, \Sigma_1) \) and \( \text{Inv}(f_1, \Sigma, \Sigma_1) \), we know

\[ \text{injective}(f_2 \circ f_1, \Sigma). \]

Thus we know

\[ \text{wf}(\mu, (\Sigma, F, (\sigma, F)). \]

Also, since \( \text{Inv}(f_1, \Sigma, \Sigma_1) \) and \( \text{Inv}(f_2, \Sigma_1, \sigma) \), we know \( \text{Inv}(f_2 \circ f_1, \Sigma, \sigma) \). Also we need to prove the following:

1. \( \forall \kappa', \Sigma', \Delta, \text{if } \beta = \circ \text{ and } F \vdash (k, \Sigma) \xrightarrow{\Delta} (k', \Sigma') \) and \( \text{HGuar}(\mu, \Lambda_0 \cup \Delta, F) \), then one of the following holds:

   a. there exists \( j \) such that
      i. \( j < i \), and
      ii. \( (F, (k', \Sigma'), \Lambda_0 \cup \Delta, \circ) \preceq_{\mu_1}^i (F, (k, \Sigma), F_1) \),

   b. or, there exist \( \kappa', \sigma', \delta \) and \( j \) such that
      i. \( F \vdash (k, \Sigma) \xrightarrow{\Delta} (k', \sigma', \delta) \),
      ii. \( \text{FPmatch}(\mu, (\Lambda_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)) \),
      iii. \( (F, (k', \Sigma'), \Lambda_0 \cup \Delta, \circ) \preceq_{\mu_1}^i (F, (k', \Sigma'), F_1) \).

Proof. From \( (F, (k, \Sigma), \Lambda_0, \beta) \preceq_{\mu_1}^i (F_1, (k_1, \Sigma_1), \Lambda_{10}, \beta) \), we know one of the following holds:

(A) there exists \( j_1 \) such that
   i. \( j_1 < i_1 \), and
   ii. \( (F_1, (k_1, \Sigma_1), \Lambda_1, \circ) \preceq_{\mu_1}^{j_1} (F_1, (k_1, \Sigma_1), \Lambda_{10}, \circ) \);

(B) or, there exist \( k'_1, \Sigma'_1, \Delta_1 \) and \( j_1 \) such that
   i. \( F_1 \dashv \vdash (k_1, \Sigma_1) \xrightarrow{\Delta_1} (k'_1, \Sigma'_1) \),
   ii. \( \text{FPmatch}(\mu_1, (\Lambda_0 \cup \Delta, \Sigma), (\Delta_0 \cup \Delta, F_1)) \),
   iii. \( (F_1, (k'_1, \Sigma'_1), \Lambda_0 \cup \Delta, \circ) \preceq_{\mu_1}^{j_1} (F_1, (k'_1, \Sigma'_1), \Lambda_{10} \cup \Lambda_1, \circ) \).

For case (A), let \( j = (i_2, j_1) \). By the co-induction hypothesis, we know

\[ (F, (k', \Sigma'), \Lambda_0 \cup \Delta, \circ) \preceq_{\mu_1}^j (F, (k, \Sigma), F_1) \]

Since \( j_1 < i_1 \), we know

\[ j < i. \]

Thus we are done.

For case (B), from \( \text{FPmatch}(\mu_1, (\Lambda_0 \cup \Delta, \Sigma), (\Lambda_{10} \cup \Lambda_1, F_1)) \), we know

\[ G(\Lambda_{10} \cup \Lambda_1, F_1, \text{LShare}(\mu_1, \Sigma)). \]

Thus we know

\[ \text{HGuar}(\mu_2, \Lambda_{10} \cup \Lambda_1, F_1). \]

From \( (F_1, (k_1, \Sigma_1), \Lambda_{10}, \beta) \preceq_{\mu_1}^{i_2} (F, (k, \Sigma), F_1) \), by Lemma 53, we know one of the following holds:

(B1) there exists \( j_2 \) such that
   i. \( j_2 < i_2 \), and
   ii. \( (F_1, (k'_1, \Sigma'_1), \Lambda_{10} \cup \Lambda_1, \circ) \preceq_{\mu_1}^{j_2} (F, (k, \Sigma), F_1) \);
Towards Certified Separate Compilation for Concurrent Programs

(B2) or, there exist $\kappa', \sigma', \delta$ and $j_2$ such that
i. $F \vdash (k, \sigma) \xrightarrow{\delta} (k', \sigma')$, and
ii. $\text{FPmatch}(\mu_2, (\Lambda_{10} \cup \Lambda_1, \Sigma_1), (\delta_0 \cup \delta, F))$, and
iii. $(\mathcal{F}_1, (k_1', \Sigma_1'), (\Lambda_{10} \cup \Lambda_1, o) \xrightarrow{\mu_1} (F, (k', \sigma'), \delta_0 \cup \delta, o)$.

For case (B1), let $j = (j_2, j_1)$. By the co-induction hypothesis, we know

$$(\mathcal{F}, (k', \Sigma'), \Lambda_0 \cup \Lambda, o) \xrightarrow{\mu} (F, (k, \sigma), \delta_0, o).$$

Since $j_2 < i_2$, we know $j < i$.

For case (B2), let $j = (j_2, j_1)$. By the co-induction hypothesis, we know

$$(\mathcal{F}, (k', \Sigma'), \Lambda_0 \cup \Lambda, o) \xrightarrow{\mu} (F, (k', \sigma'), \delta_0 \cup \delta, o).$$

From $\text{FPmatch}(\mu_1, (\Lambda_0 \cup \Lambda, \Sigma), (\Lambda_{10} \cup \Lambda_1, \mathcal{F}_1))$ and $\text{FPmatch}(\mu_2, (\Lambda_{10} \cup \Lambda_1, \Sigma_1), (\delta_0 \cup \delta, F))$, by Lemma 55, we know

$$\text{FPmatch}(\mu, (\Lambda_0 \cup \Lambda, \Sigma), (\delta_0 \cup \delta, F)).$$

Thus we are done.

2. $\forall k', i$, if $\beta = o, i \neq \tau$ and $\mathcal{F} \vdash (k, \Sigma) \xrightarrow{\mu} (k', \Sigma)$ and $\text{HGuar}(\mu, \Lambda_0, \mathcal{F})$, then there exist $\kappa', \delta, \sigma', \kappa', \mu'$ and $j$ such that

a. $F \vdash (k, \sigma) \xrightarrow{\delta} (k', \sigma')$, and

$F \vdash (k', \sigma') \xrightarrow{\mu} (k'', \sigma'')$, and

b. $\text{FPmatch}(\mu, (\Lambda_0, \Sigma), (\delta_0 \cup \delta, F))$, and

c. $\text{EvolveG}(\mu, (\Sigma, \sigma', \mathcal{F}, F), (\delta_0 \cup \delta, \Sigma))$, and

d. $(\mathcal{F}, (k', \Sigma), \text{emp}, \bullet) \xrightarrow{\mu} (F, (k'', \sigma''), \text{emp}, \bullet)$.

Proof. From $(\mathcal{F}, (k, \Sigma), (\Lambda_0, \beta)) \xrightarrow{\mu}(\mathcal{F}_1, (k_1, \Sigma_1), (\Lambda_{10}, \beta))$, we know there exist $k_1', k_1 \in \Lambda_0, \Sigma_1$, $k_1', \mu_1'$ and $j_1$ such that

a. $\mathcal{F}_1 \vdash (k_1, \Sigma_1) \xrightarrow{\mu}(k_1', \Sigma_1')$, and $\mathcal{F}_1 \vdash (k_1', \Sigma_1') \xrightarrow{\mu}(k_1'', \Sigma_1'')$, and

b. $\text{FPmatch}(\mu_1, (\Lambda_0, \Sigma_1), (\delta_0 \cup \delta, F_1))$, and

c. $\text{EvolveG}(\mu_1, (\Sigma_1', \sigma', \mathcal{F}_1, F_1), (\delta_0 \cup \delta, \Sigma_1))$, and

d. $(\mathcal{F}_1, (k_1', \Sigma_1'), \text{emp}, \bullet) \xrightarrow{\mu}(\mathcal{F}_1, (k_1'', \Sigma_1''), \text{emp}, \bullet)$.

From $\text{FPmatch}(\mu_1, (\Lambda_0, \Sigma), (\Lambda_{10} \cup \Lambda_1, \mathcal{F}_1))$, we know

$$G(\Lambda_{10} \cup \Lambda_1, \mathcal{F}_1, \text{LShare}(\mu_1, \Sigma)).$$

Thus we know

$$\text{HGuar}(\mu_2, (\Lambda_{10} \cup \Lambda_1, \mathcal{F}_1)).$$

From $(\mathcal{F}_1, (k_1, \Sigma_1), (\Lambda_{10}, \beta)) \xrightarrow{\mu}(\mathcal{F}_1, (k_1, \Sigma_1), (\Lambda_{10}, \beta))$, by Lemma 54, we know there exist $\kappa', \delta, \sigma', \kappa', \mu_2'$ and $j_2$ such that

a. $F \vdash (k, \sigma) \xrightarrow{\delta} (k', \sigma')$, and

$F \vdash (k', \sigma') \xrightarrow{\mu} (k'', \sigma'')$, and

b. $\text{FPmatch}(\mu_2, (\Lambda_{10} \cup \Lambda_1, \Sigma_1), (\delta_0 \cup \delta, F))$, and

c. $\text{EvolveG}(\mu_2, (\Sigma_1', \sigma', \mathcal{F}_1, F), (\delta_0 \cup \delta, \Sigma_1))$, and

d. $(\mathcal{F}_1, (k_1', \Sigma_1'), \text{emp}, \bullet) \xrightarrow{\mu}(\mathcal{F}_1, (k_1'', \Sigma_1''), \text{emp}, \bullet)$.

Suppose $\mu_1' = (S', S_1', f_1')$ and $\mu_2' = (S_1', S', f_2')$. Since $\text{EvolveG}(\mu_1, \mu_1', \Sigma_1, \mathcal{F}_1, F)$ and $\text{EvolveG}(\mu_2, \mu_2', \Sigma_1', \mathcal{F}_1, F)$, we know

$$S_1' = \text{cl}(S_1, \Sigma_1') \quad \text{and} \quad S_2' = \text{cl}(S_1, \Sigma_1).$$

Thus $S_1' = S_2'$. Let $\mu' = (S', S', f_2' \circ f_1')$. Let $j = (j_2, j_1)$. By the co-induction hypothesis, we know

$$(\mathcal{F}, (k', \Sigma), \text{emp}, \bullet) \xrightarrow{\mu}(F, (k'', \sigma''), \text{emp}, \bullet).$$

From $\text{FPmatch}(\mu_1, (\Lambda_0, \Sigma), (\Lambda_{10} \cup \Lambda_1, \mathcal{F}_1))$ and $\text{FPmatch}(\mu_2, (\Lambda_{10} \cup \Lambda_1, \Sigma_1), (\delta_0 \cup \delta, F))$, by Lemma 55, we know

$$\text{FPmatch}(\mu, (\Lambda_0, \Sigma), (\delta_0 \cup \delta, F)).$$

From $\text{EvolveG}(\mu_1, \mu_1', \Sigma_1', \mathcal{F}_1, F)$, we know $\text{EvolveG}(\mu, \mu_1', \Sigma_1', \mathcal{F}_1, F)$. By Lemma 56, we know

$$\text{EvolveG}(\mu, \mu_1', \Sigma_1', \mathcal{F}_1).$$

Thus we are done.

3. $\forall \sigma', \Sigma_1', \mu'$, if $\beta = \bullet$ and $\text{Rely}(\mu, \mu', (\Sigma, \Sigma_1', \mathcal{F}, (\sigma_1', \sigma', F)))$, then there exists $j$ such that $(\mathcal{F}, (k, \Sigma_1'), \text{emp}, \circ) \xrightarrow{\mu}(F, (k, \sigma_1'), \text{emp}, \circ)$. 

33
We first construct $f'$, Let $S'' = S - S$. Thus $S'' \cap F = \emptyset$. From $S' = \text{cl}(S, \Sigma') \subseteq \text{dom}(\Sigma')$, $S' = \text{cl}(S, \sigma')$, $(S' - S) \cap F = \emptyset$.

We first construct $f'_1$. Let $S'' = S - S$. Thus $S'' \cap F = \emptyset$. From $S' = \text{cl}(S, \Sigma')$, we know $S' = S \cup S''$. Pick an arbitrary set of fresh blocks $S''_1$ such that $|S''_1| = |S''|$, $S''_1 \cap \text{dom}(\Sigma') = \emptyset$, $S''_1 \cap S_1 = \emptyset$, $S''_1 \cap F_1 = \emptyset$.

Pick an arbitrary function $f_b$ mapping blocks in $S''$ to the blocks of $S''_1$ such that

$$
injective(f_b) \quad \text{and} \quad \text{dom}(f_b) = S'' \quad \text{and} \quad \text{range}(f_b) = S''_1$$

Define the function $f''_1$ as follows.

$$\forall b \in S''. f''_1(b) = (f(b), 0).$$

Thus we know

$$injective(f''_1) \quad \text{and} \quad \text{dom}(f''_1) = S'' \quad \text{and} \quad f''_1(S'') = S''_1.$$ 

Let $f'_1 = f_1 \cup f''_1$. Since $\text{wif}(\mu_1, \Sigma, F, F_1)$, we know

$$injective(f'_1) \quad \text{and} \quad \text{dom}(f'_1) = S' \quad \text{and} \quad f'_1(S') \subseteq S_1 \cup S''_1 \quad \text{and} \quad f'_1(S' \cap F) \subseteq F_1 \quad \text{and} \quad f'_1(S' - F) \cap F_1 = \emptyset.$$

Next we construct $S'_1$. Let $S'_1 = S_1 \cup S''_1$. Then we construct $f'_2$. Define a function $f''_2$ as follows:

$$\text{dom}(f''_2) = S''_1 \quad \text{and} \quad \forall b \in S''_1. f''_2(b) = f'(f_b^{-1}(b)).$$

Since $f''_2(S'') \subseteq S'$, we know

$$f''_2(S'') = f'(\text{dom}(f'_2)) \subseteq S'.$$

Let $f'_2 = f_2 \cup f''_2$. Since $\text{wif}(\mu_2, \Sigma_1, F, F_1)$ and $f'_{2}(S' - F) \cap F = \emptyset$, we know

$$\text{dom}(f'_2) = S'_1 \quad \text{and} \quad f'_2(S'_1) \subseteq S' \quad \text{and} \quad f'_2(S'_1 \cap F_1) \subseteq F \quad \text{and} \quad f'_2(S'_1 - F_1) \cap F = \emptyset.$$

Also we know

$$f'_2 \circ f' = (f_2 \circ f_1) \cup (f''_2 \circ f''_1) = f'.$$

Finally we construct $\Sigma'_1$. Define

$$\text{dom}(\Sigma'_1) = \text{dom}(\Sigma_1) \cup S''_1$$

$$\forall b_0 \in \text{dom}(\Sigma_1) - S_1, \text{dom}(\Sigma'_1(b_0)) = \text{dom}(\Sigma_1(b_0)) \quad \forall i. \Sigma'_1(b_i)(i) = \Sigma_1(b_i)(i)$$

$$\forall b_{10} \in S_1 - f_1(S) - F_1, \text{dom}(\Sigma'_1(b_{10})) = \emptyset$$

$$\forall b_{11} \in f_1(S) - F_1, \text{dom}(\Sigma'_1(b_{11})) = \{i \mid \exists b, n. f_1(b) = (b_{11}, n) \land (b, i - n) \in \text{locs}(\Sigma')\}$$

$$\forall i. \Sigma'_1(b_{11})(i) = \begin{cases} v' & \text{if } \exists b, n. f_1((b, n)) = (b_{11}, i) \land \Sigma'(b(i - n)) = v' \not\equiv \top \\ v & \text{if } \exists b, n. b', i', b', n'. f_1((b, n)) = (b_{11}, i - n) = (b', i') \land f'_1(b') = (b', n') \land v = (b', i' + n') \\ \end{cases}$$

$$\forall b_1 \in S_1 \cap F_1, \text{dom}(\Sigma'_1(b_1)) = \{i \mid (b_1, i) \in (f_1(\text{locs}(\Sigma') \cap \text{dom}(\Sigma'_1(b_1)))) \cup (\text{locs}(\Sigma_1) - f_1(\text{locs}(\Sigma) \cap \text{dom}(\Sigma'_1(b_1)))) \}$$

$$\forall (b_1, i) \in f_1(\text{locs}(\Sigma) \cap \text{dom}(\Sigma'_1(b_1))) \implies \Sigma'_1(b_1)(i) = \Sigma_1(b_1)(i)$$

$$\forall b_2 \in S''_1. \text{dom}(\Sigma'_1(b_2)) = \text{dom}(\Sigma'(f_b^{-1}(b)))$$

$$\forall i. \Sigma'_1(b_2)(i) = \begin{cases} v' & \text{if } \exists b_2, b', i, n, f_b^{-1}(b_2) = b' \land \Sigma'(b(i))(i) = v' \not\equiv \top \\ v & \text{if } \exists b_2, b', i, n, f_b^{-1}(b_2) = b'_1 \land \Sigma'(b(i))(i) = (b', i') \land f'_1(b') = (b, n) \land v = (b, i' + n) \\ \end{cases}$$

Below we prove the following:

- forward($\Sigma'_1, \Sigma_1$).

We only need to prove $\text{locs}(\Sigma'_1) \cap \text{dom}(\Sigma'_1) \subseteq \text{locs}(\Sigma_1)$. By the definition of $\Sigma'_1$, we only need to prove: for any $b_i$ and $i$, if $b_1 \in (f_1(S) - F_1) \cup (S_1 \cap F_1)$ and $i \in \text{dom}(\Sigma'_1(b_1))$, then $(b_1, i) \in \text{locs}(\Sigma_1)$.

If $b_i \in (f_1(S) - F_1)$, we know there exist $b$ and $n$ such that $f_1(b) = (b_1, n)$ and $(b, i - n) \in \text{locs}(\Sigma')$. Since $b \in S \subseteq \text{dom}(\Sigma)$ and $\text{locs}(\Sigma') \cap \text{dom}(\Sigma'_1) \subseteq \text{locs}(\Sigma)$, we know $(b, i - n) \in \text{locs}(\Sigma)$. Since Inv($f_1, \Sigma, \Sigma_1$), we know $(b_1, i) \in \text{locs}(\Sigma_1)$. 

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng
Towards Certified Separate Compilation for Concurrent Programs

If \( b_1 \in (\mathbb{S}_1 \cap \mathbb{F}_1) \), we know \((b_1, i) \in (f_1(\text{locs}(\Sigma')) \cap \|\mathbb{S}_1\|) \cup (\text{locs}(\Sigma_1) - f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|))\). Since \(\text{locs}(\Sigma') \cap \|\text{dom}(\Sigma')\| \subseteq \text{locs}(\Sigma)\), we know \(f_1(\text{locs}(\Sigma') \cap \|\mathbb{S}_1\|) \subseteq f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|)\). Since injective \((f_1, \Sigma, \Sigma_1)\), we know \(f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|) \subseteq \text{locs}(\Sigma_1)\). Thus \((b_1, i) \in \text{locs}(\Sigma_1)\).

- **injective\(f_2', \Sigma'_1\).**
  
  We prove the following.
  - injective\((f_2, \Sigma_1)\) and forward\((\Sigma_1, \Sigma'_1)\), we know injective\((f_2, \Sigma'_1)\).

- **injective\(f_2'', \Sigma'_1\).**
  
  For any \( b_1, b_2, i_1, i_2, b_1', b_2', n_1 \) and \( n_2 \), if \((b_1, i_1) \neq (b_2, i_2), (b_1, i_1) \in \text{locs}(\Sigma'_1), (b_2, i_2) \in \text{locs}(\Sigma'_1), f_2''(b_1) = (b_1', n_1) \) and \( f_2''(b_2) = (b_2', n_2) \), by the definition of \(f_2''\), we know

  \[
  b_1 \in \mathbb{S}'_1, \quad b_2 \in \mathbb{S}'_1, \quad f_2''(b_1) = f'(fb_1^{-1}(b_1)), \quad f_2''(b_2) = f'(fb_2^{-1}(b_2)).
  \]

  Also by the definition of \(\Sigma'_1\), we know

  \[
  \text{dom}(\Sigma_1(b_1)) = \text{dom}(\Sigma'(fb_1^{-1}(b_1))) \quad \text{and} \quad \text{dom}(\Sigma_1(b_2)) = \text{dom}(\Sigma'(fb_2^{-1}(b_2))).
  \]

  Thus

  \[
  (fb_1^{-1}(b_1), i_1) \in \text{locs}(\Sigma') \quad \text{and} \quad (fb_2^{-1}(b_2), i_2) \in \text{locs}(\Sigma').
  \]

  Since injective\((f', \Sigma')\), we know \(b_1' \neq b_2'\) or \(i_1 + n_1 \neq i_2 + n_2\). Thus injective\((f_2'', \Sigma'_1)\).

- \( \forall b_1, b_2, i_1, i_2, b_1', b_2', n_1, n_2, (b_1, i_1) \in \text{locs}(\Sigma'_1) \land (b_2, i_2) \in \text{locs}(\Sigma'_1) \land f_2(b_1) = (b_1', n_1) \land f_2(b_2) = (b_2', n_2) \Rightarrow (b_1' \neq b_2') \lor (i_1 + n_1 \neq i_2 + n_2). \)

  Since \(b_1 \in \mathbb{S}'_1\), we consider three cases:

  1. \(b_1 \in f_1(\mathbb{S}) - \mathbb{F}_1\). Since \(\text{dom}(\Sigma'_1(b_1)) = \emptyset\), the above holds trivially.

     \[
     b_1 \in f_1(\mathbb{S}) - \mathbb{F}_1
     \]

     By the definition of \(f_2''\), we know \(b_2 \in \mathbb{S}'_1\) and \(f_2''(b_2) = f'(fb_2^{-1}(b_2))\). Also by the definition of \(\Sigma'_1\), we know

     \[
     \text{dom}(\Sigma'_1(b_1)) = \{i \mid \exists b, n. f_1(b) = (b_1, n) \land (b, i - n) \in \text{locs}(\Sigma')\}
     \]

     Since \(f_2 \circ f_1 \subseteq f'\), we know there exist \(b\) and \(n\) such that \(f_1(b) = (b_1, n)\) and \((b, i - n) \in \text{locs}(\Sigma')\) and \(f'(b) = (b_1', i_1 + n)\).

     Also \((fb_1^{-1}(b_2), i_2) \in \text{locs}(\Sigma')\). Since injective\((f', \Sigma')\), we know \(b_1' \neq b_2'\) or \(i_1 + n_1 \neq i_2 + n_2\).

     \[
     b_1 \in \mathbb{S}'_1 \cap \mathbb{F}_1
     \]

     Since \(f_2 \mathbb{S} \cap \mathbb{F}_1 \subseteq \mathbb{F}_1\), we know \(b_2' \in \mathbb{F}_1\). By the definition of \(f_2''\), we know \(b_2 \in \mathbb{S}'_1\) and \(f_2''(b_2) = f'(fb_2^{-1}(b_2))\).

     Since \(\mathbb{S}'_1 \cap \mathbb{F}_1 = \emptyset\), we know \(fb_2^{-1}(b_2) \notin \mathbb{F}_1\). Since \(f' \mathbb{S} \cap \mathbb{F}_1 \cap \mathbb{F} = \emptyset\), we know \(b_2' \notin \mathbb{F}_1\). Thus \(b_1' \neq b_2'\).

  From injective\((f_2'')\), and the definition of \(f_2''\), we are done.

- **\(\sigma \in \frac{\mathbb{F}}{f_1(\mathbb{S}) \cap \|\mathbb{S}_1\|} \quad \Sigma'\).**

  Since \(f_1(\mathbb{S}) \subseteq \mathbb{S}_1\) and \(\text{Inv}(f_1, \Sigma, \Sigma_1)\), we know \(f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}\|) \subseteq \text{locs}(\Sigma_1) \cap \|\mathbb{S}_1\|\). Thus \(\mathbb{F} - f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|) \subseteq f_2(\mathbb{S} - \mathbb{F}_1) \subseteq \mathbb{F}_1\). From \(\sigma \in \frac{\mathbb{F} - f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|)}{f_1(\mathbb{S}) \cap \|\mathbb{S}_1\|} \quad \Sigma'\), we are done.

- **\(\Sigma_1 \in \frac{\mathbb{F}_1 - \mathbb{S}_1}{\mathbb{F}_1 \cap \|\mathbb{S}_1\|} \quad \Sigma'_1\).**

  Since \(f_1(\mathbb{S}) \subseteq \mathbb{S}_1\), we know \(\mathbb{F}_1 - \mathbb{S}_1 \subseteq \mathbb{F}_1 - f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|)\). Thus we only need to prove \(\Sigma_1 \in \frac{\mathbb{F}_1 - f_1(\text{locs}(\Sigma) \cap \|\mathbb{S}_1\|)}{\mathbb{F}_1 \cap \|\mathbb{S}_1\|} \quad \Sigma'_1\). By the above definition of \(\Sigma'_1\), we are done.

- **\(\text{Inv}(f_2', \Sigma'_1, \sigma')\).**

  For any \(b_1, i, b_1', i', \) if \((b_1, i) \in \text{locs}(\Sigma'_1)\) and \(f_2''(b_1, i) = (b_1', i')\), by the definition of \(\Sigma'_1\), we consider the following cases:

  - \(b_1 \in f_1(\mathbb{S}) - \mathbb{F}_1\). Thus there exist \(b\) and \(n\) such that \(f_1(b) = (b_1, n)\) and \(\Sigma'(b)(i - n) \in \text{locs}(\Sigma)\) and \(\Sigma'(b)(i - n) \xrightarrow{f'} \Sigma'_1(b)(i)\).

  Since \(f_2''(b_1, i) = (b_1', i')\), we know

  \[
  f_2''((b, i - n)) = (b_1', i').
  \]

  From \(\text{Inv}(f', \Sigma', \sigma')\), we know \((b_1', i') \in \text{locs}(\sigma')\) and \(\Sigma'(b')(i - n) \xrightarrow{f'} \sigma'(b')(i')\). Since \(f_2'' = f'\), we know \(\Sigma'_1(b)(i) \xrightarrow{f'} \sigma'(b')(i')\).
By induction over $\exists n$ such that $f_1((b_0, n)) = (b_1, n_1)$ and $b_1 \in S_1$. Since $\text{Inv}(f_1', \Sigma', \Sigma'_1)$, we know $\Sigma'_1(b_1(n_1)) \not\subseteq \Sigma'_1(b_1(n_1))$.

Thus there exists $n$ such that

$$\Sigma'_1(b_1(n_1)) = f_1'((b', n')) = (b, n).$$

Thus $b \in \text{cl}(S_1, \Sigma'_1)$. 

* $k = m + 1$. Then we have $b_0, n_0$ such that $b_0 \in S$ and $\Sigma'(b_0(n_0)) = (b', n')$. Then we know there exist $b_1$ and $n_1$ such that $f_1'((b_0, n_0)) = (b_1, n_1)$. By the induction hypothesis, we have $b_1 \in \text{cl}(S_1, \Sigma'_1)$. Since $\text{Inv}(f_1', \Sigma', \Sigma'_1)$, we know $\Sigma'_1(b_1(n_1)) \not\subseteq \Sigma'_1(b_1(n_1))$.

Thus there exists $n$ such that

$$\Sigma'_1(b_1(n_1)) = f_1'((b', n')) = (b, n).$$

Thus $b \in \text{cl}(S_1, \Sigma'_1)$.

* $\text{cl}(S_1, \Sigma'_1) \subseteq S'_1$.
Towards Certified Separate Compilation for Concurrent Programs

We need to prove: for any \( k, \) \( \text{cl}_k(S_1, \Sigma_1') \subseteq S_1' \). By induction over \( k \).

- \( k = 0 \). We know \( \text{cl}_0(S_1, \Sigma_1') = S_1 \subseteq S_1' \).
- \( k = m + 1 \). By the induction hypothesis, we know

\[
\text{cl}_{m+1}(S_1, \Sigma_1') \subseteq \{ b' \mid \exists b, i, i', (b \in S_1') \wedge \Sigma_1'(b)(i) = (b', i') \}
\]

We only need to prove the following:

\[
\forall b, i, b', i'. (b \in S_1') \wedge \Sigma_1'(b)(i) = (b', i') \implies b' \in S_1'
\]

By the definition of \( \Sigma_1' \), we consider the following cases:

* \( b \in f_1(S) \ominus F_i \). Thus there exist \( b_0 \) and \( i_0 \) such that \( f_1((b_0, i_0)) = (b, i), (b_0, i_0) \in \text{locs}(\Sigma') \) and \( \Sigma'(b_0)(i_0) \xrightarrow{f_1} \Sigma_1'(b)(i) \).

Thus there exist \( b'_0 \) and \( i'_0 \) such that \( \Sigma'(b_0)(i_0) = (b'_0, i'_0) \) and \( f'_1((b'_0, i'_0)) = (b', i') \). Thus \( b' \in S_1' \).

* \( b \in S_1 \cap F_i \) and \( (b, i) \in f_1(S) \cap \| \Sigma \| \). The same as the earlier case.

* \( b \in S_1 \cap F_i \) and \( (b, i) \in \text{locs}(\Sigma_1) - f_1(\text{locs}(\Sigma) \cap \| \Sigma \|) \). Thus \( \Sigma_1'(b)(i) = \Sigma_1(b)(i) \). Since \( S_1 = \text{cl}(S_1, \Sigma_1) \), we know \( b' \in S_1 \).

Let \( \mu_1 = (S_1, S_1', f_1') \) and \( \mu_2 = (S_1', S_1, f_1) \). Thus we know

\[
\text{EvolveR}(\mu_1, \mu_2', \Sigma_1, F, F_i, \Sigma_1') \text{ and } \text{EvolveR}(\mu_2, \mu_1', \Sigma_1', F_i, F, \Sigma_1) \text{ and } \mu' = (S_1', S_1', f_1' \circ f_1').
\]

Thus from \( (F, (k, \Sigma), \Delta_0, \sigma) \preceq_{\mu}^j (F_1, (k_1, \Sigma_1), \Delta_0, \sigma) \) and \( F \vdash (k, \Sigma) \xrightarrow{\Delta} (k', \Sigma') \) and \( \text{HGuar}(\mu, \Delta_0 \cup \Delta, F) \), then one of the following holds:

1. there exists \( j \) such that
   a. \( j < i \), and
   b. \( (F, (k', \Sigma'), \Delta_0 \cup \Delta, \sigma) \preceq_{\mu}^j (F, (k, \Sigma), \Delta_0, \sigma) \);
2. or, there exist \( k', \Sigma', \Delta' \) and \( \Delta'' \) such that
   a. \( F \vdash (k, \Sigma) \xrightarrow{\Delta} (k', \Sigma') \), and
   b. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \Sigma), (\Delta_0 \cup \Delta, F)) \), and
   c. \( (F, (k', \Sigma'), \Delta_0 \cup \Delta, \sigma) \preceq_{\mu}^j (F, (k', \Sigma'), \Delta_0 \cup \Delta, \sigma) \).

Proof: By induction over \( n \).

1. \( n = 0 \). Trivial.
2. \( n = m + 1 \). We know there exists \( k'', \Sigma'', \Delta' \) and \( \Delta'' \) such that

\[
F \vdash (k, \Sigma) \xrightarrow{\Delta} (k'', \Sigma''), \quad F \vdash (k'', \Sigma'') \xrightarrow{\Delta_m+1} (k', \Sigma'), \quad \Delta = \Delta' \cup \Delta''.
\]

Since \( \text{HGuar}(\mu, \Delta_0 \cup \Delta, F) \), we know

\[
\text{HGuar}(\mu, \Delta_0 \cup \Delta', F).
\]

Since \( (F, (k, \Sigma), \Delta_0, \sigma) \preceq_{\mu}^j (F, (k, \Sigma), \Delta_0, \sigma) \), we know one of the following holds:

(a) there exists \( i' \) such that
   i. \( i' < i \), and
   ii. \( (F, (k', \Sigma'''), \Delta_0 \cup \Delta', \sigma) \preceq_{\mu}^{i'} (F, (k, \Sigma), \Delta_0, \sigma) \);
(b) or, there exist \( k'', \sigma'', \Delta' \) and \( i' \) such that
   i. \( F \vdash (k, \Sigma) \xrightarrow{\Delta} (k'', \Sigma'') \), and
   ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta', \Sigma), (\Delta_0 \cup \Delta', F)) \), and

37
Thus we have finished the proof.

**Lemma 54.**

By induction over \( \beta_1 \) there exists

1. \( (F, (k, \Sigma'), \Delta_0 \cup \Delta, \circ) \prec_i (F, (k, \sigma), \delta_0, \circ) \);

(a1) there exists \( j \) such that
   1. \( j < i' \), and
   2. \( (F, (k', \Sigma'), \Delta_0 \cup \Delta, \circ) \prec_j (F, (k, \sigma), \delta_0, \circ) \);

(a2) or, there exist \( k', \sigma', \delta \) and \( j \) such that
   1. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   2. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   3. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   4. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and

Since \( F \vdash (k, \Sigma) \stackrel{r}{\rightarrow}^n (k'', \Sigma'') \), we know forward(\( \Sigma, \Sigma'' \)) and (\( \text{locs}(\Sigma) - \text{locs}(\Sigma'') \)) \( \subseteq \text{A'} \cdot \text{ws} \cap \text{A}' \cdot \text{rs} \). Thus, if \( \text{FPmatch} (\mu, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)) \), then

Thus we are done.

For case (b), by the induction hypothesis, we know one of the following holds:

(b1) there exists \( j \) such that
   1. \( j < i' \), and
   2. \( (F, (k', \Sigma'), \Delta_0 \cup \Delta, \circ) \prec_j (F, (k, \sigma), \delta_0, \circ) \);

(b2) or, there exist \( k', \sigma', \delta'' \) and \( j \) such that
   1. \( F \vdash (k'', \sigma'') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   2. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   3. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and
   4. \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^*(k', \sigma') \), and

Thus we are done.

For case (b2), we know

Thus we are done.

Thus we have finished the proof. \( \square \)

**Lemma 54.** If \( (F, (k, \Sigma), \Delta_0, \circ) \prec_i (F, (k, \sigma), \delta_0, \circ) \) and \( F \vdash (k, \Sigma) \stackrel{r}{\rightarrow}^n (k', \Sigma') \) and \( F \vdash (k', \Sigma') \stackrel{r}{\rightarrow}^m (k'', \Sigma'') \) and \( i \neq r \) and\n
\( \text{HGuar} (\mu, \Delta_0 \cup \Delta, F) \), then there exist \( k'', \delta', \sigma', k', \mu' \) and \( j \) such that

1. \( F \vdash (k, \sigma) \stackrel{r}{\rightarrow}^n (k', \sigma') \), and \( F \vdash (k', \sigma') \stackrel{r}{\rightarrow}^m (k'', \Sigma') \), and
2. \( F \vdash (k', \Sigma') \stackrel{r}{\rightarrow}^m (k'', \Sigma'') \), and
3. \( F \vdash (k'', \Sigma'') \stackrel{r}{\rightarrow}^m (k', \Sigma') \), and
4. \( (\Sigma, \Sigma', \text{emp}, \bullet) \prec_j (F, (k, \sigma), \delta_0, \circ) \).

**Proof.** By induction over \( n \).

1. \( n = 0 \). Trivial.
2. \( n = m + 1 \). We know there exists \( k'' ', \Sigma'', \Delta' \) and \( \Delta'' \) such that

\( F \vdash (k, \Sigma) \stackrel{r}{\rightarrow}^n (k'', \Sigma''), \quad F \vdash (k'', \Sigma'') \stackrel{r}{\rightarrow}^m (k', \Sigma') \), \( \Delta = \Delta' \cup \Delta'' \).

Since \( \text{HGuar} (\mu, \Delta_0 \cup \Delta, F) \), we know

\( \text{HGuar} (\mu, \Delta_0 \cup \Delta, F) \).

Since \( (F, (k, \Sigma), \Delta_0, \circ) \prec_i (F, (k, \sigma), \delta_0, \circ) \), we know one of the following holds:

(a) there exists \( i' \) such that
   1. \( i' < i \), and
   2. \( (F, (k'', \Sigma'''), \Delta_0 \cup \Delta, \circ) \prec_j (F, (k, \sigma), \delta_0, \circ) \);

(b) or, there exist \( k'', \sigma'', \delta' \) and \( i' \) such that
   1. \( F \vdash (k, \sigma) \stackrel{r}{\rightarrow}^n (k'', \sigma'') \), and
Towards Certified Separate Compilation for Concurrent Programs

ii. \(\text{FPmatch}(\mu, (\Lambda_0 \cup \Lambda', \Sigma), (\delta_0 \cup \delta', F))\), and

iii. \((F, (k'', \Sigma''), \Lambda_0 \cup \Lambda', o) \vdash^\mu (F, (k'', \Sigma''), \delta_0 \cup \delta', o)\).

For case (a), by the induction hypothesis, we are done.

For case (b), by the induction hypothesis, we know there exist \(\kappa', \delta, \sigma', \kappa, \mu'\) and \(j\) such that

1. \(F \vdash (k'', \sigma'') \models^F (k', \sigma')\), and \(F \vdash (k', \sigma') \models^\mu (k'', \sigma'')\).

2. \(\text{FPmatch}(\mu, (\Lambda_0 \cup \Lambda, \Sigma''), (\delta_0 \cup \delta' \cup \delta, F))\), and

3. \(\text{EvolveG}(\mu, \mu', \Sigma', \sigma', F, F)\), and

4. \((F, (k'', \Sigma'), \text{emp}, \bullet) \vdash^\mu (F, (k'', \Sigma''), \text{emp}, \bullet)\).

Since \(F \vdash (k, \Sigma) \models^F (k'', \Sigma'')\), we know forward(\(\Sigma, \Sigma''\)) and (\(\text{locs}(\Sigma) - \text{locs}(\Sigma'')\)) \(\subseteq \Lambda'. \text{ws} \cap \Lambda'. \text{rs}\). Thus, from \(\text{FPmatch}(\mu, (\Lambda_0 \cup \Lambda, \Sigma''), (\delta_0 \cup \delta' \cup \delta, F))\), by Lemma 57, then

\(\text{FPmatch}(\mu, (\Lambda_0 \cup \Lambda, \Sigma), (\delta_0 \cup \delta' \cup \delta, F))\).

Thus we are done.

Thus we have finished the proof. \(\Box\)

**Lemma 55.** If \(\text{FPmatch}(\mu_1, (\Lambda_1, \Sigma_1), (\Lambda_2, F_2))\), \(\text{FPmatch}(\mu_2, (\Lambda_2, \Sigma_2), (\Lambda_3, F_3))\), \(\text{wf}(\mu_1, (\Sigma_1, F_1))\), \(\text{wf}(\mu_2, (\Sigma_2, F_2))\), \(\mu_1 = (S_1, S_2, f_1)\), \(\mu_2 = (S_3, S_4, f_2)\) and \(\mu = (S_1, S_3, f_2 \circ f_1)\), then \(\text{FPmatch}(\mu, (\Lambda_1, \Sigma_1), (\Lambda_3, F_3))\).

**Proof.** From \(\text{FPmatch}(\mu_1, (\Lambda_1, \Sigma_1), (\Lambda_2, F_2))\) and \(\text{FPmatch}(\mu_2, (\Lambda_2, \Sigma_2), (\Lambda_3, F_3))\), we know

\[
\begin{align*}
\Lambda_2, \text{rs} &\cup \Lambda_2, \text{ws} \subseteq [\|F_2\|] \cup f_1(\text{locs}(\Sigma_1) \cap \|S_1\|), \\
\Lambda_2, \text{rs} \cap f_1(\text{locs}(\Sigma_1) \cap \|S_1\|) &\subseteq f_1(\text{locs}(\Sigma_1) \cap \Lambda_1, \text{rs}), \\
\Lambda_2, \text{ws} \cap f_1(\text{locs}(\Sigma_1) \cap \|S_1\|) &\subseteq f_1(\text{locs}(\Sigma_1) \cap \Lambda_1, \text{ws}), \\
\Lambda_3, \text{rs} &\cup \Lambda_3, \text{ws} \subseteq [\|F_3\|] \cup f_2(\text{locs}(\Sigma_2) \cap \|S_2\|), \\
\Lambda_3, \text{rs} \cap f_2(\text{locs}(\Sigma_2) \cap \|S_2\|) &\subseteq f_2(\text{locs}(\Sigma_2) \cap \Lambda_2, \text{rs}), \\
\Lambda_3, \text{ws} \cap f_2(\text{locs}(\Sigma_2) \cap \|S_2\|) &\subseteq f_2(\text{locs}(\Sigma_2) \cap \Lambda_2, \text{ws}).
\end{align*}
\]

From \(\text{wf}(\mu_1, (\Sigma_1, F_1), (\Sigma_2, F_2))\) and \(\text{wf}(\mu_2, (\Sigma_2, F_2), (\Sigma_3, F_3))\), we know

- \(\text{inj}(f_1, \Sigma_1)\), \(S_1 \subseteq \text{dom}(f_1)\), \(f_1(\|S_1\|) \subseteq S_2\), \(f_1(\|F_1\| \cap S_1) \subseteq F_2\), \(f_1(S_1 - F_1) \cap F_2 = \emptyset\),
- \(\text{inj}(f_2, \Sigma_2)\), \(S_2 \subseteq \text{dom}(f_2)\), \(f_2(S_2) \subseteq S_3\), \(f_2(F_2) \subseteq F_3\), \(f_2(S_2 - F_2) \cap F_3 = \emptyset\).

Thus

\[
\begin{align*}
\Lambda_3, \text{rs} &\subseteq \|F_3\| \cup (\Lambda_3, \text{rs} \cap f_2(\text{locs}(\Sigma_2) \cap \|S_2\|)) \subseteq \|F_3\| \cup f_2(\text{locs}(\Sigma_2) \cap \|S_2\|) \\
\Lambda_3, \text{ws} &\subseteq \|F_3\| \cup (\Lambda_3, \text{ws} \cap f_2(\text{locs}(\Sigma_2) \cap \|S_2\|)) \subseteq \|F_3\| \cup f_2(\text{locs}(\Sigma_2) \cap \|S_2\|).
\end{align*}
\]

Similarly, we can also prove \(\Lambda_3, \text{rs} \subseteq \|F_3\| \cup (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \|S_1\|)\).

Also, since \(f_1(\text{locs}(\Sigma_1)) \subseteq \text{locs}(\Sigma_2)\) and \(f_1(\|S_1\|) \subseteq S_2\), we know

\(f_1(\text{locs}(\Sigma_1) \cap \|S_1\|) \subseteq \text{locs}(\Sigma_2) \cap \|S_2\|\).

Since \(\text{inj}(f_2, \Sigma_2)\), we know

\(f_2(\text{locs}(\Sigma_2) \cap \Lambda_2, \text{rs}) \cap f_2(\text{locs}(\Sigma_1) \cap \|S_1\|) \subseteq f_2(\text{locs}(\Sigma_1) \cap \|S_1\|)\).

Thus

\[
\begin{align*}
\Lambda_3, \text{rs} &\cap (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \|S_1\|) \\
&\subseteq (\Lambda_3, \text{rs} \cap f_2(\text{locs}(\Sigma_2) \cap \|S_2\|)) \cap (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \|S_1\|) \\
&\subseteq f_2(\text{locs}(\Sigma_2) \cap \Lambda_2, \text{rs}) \cap f_2(\text{locs}(\Sigma_1) \cap \|S_1\|) \\
&\subseteq f_2(\text{locs}(\Sigma_1) \cap \|S_1\|) \cap (\Lambda_3, \text{rs} \cap f_1(\text{locs}(\Sigma_1) \cap \|S_1\|)) \\
&\subseteq (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \Lambda_2, \text{rs}) \\
&\subseteq (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \Lambda_1, \text{rs}.
\end{align*}
\]

Similarly, we can also prove \(\Lambda_3, \text{ws} \cap (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \|S_1\|) \subseteq (f_2 \circ f_1)(\text{locs}(\Sigma_1) \cap \Lambda_1, \text{rs})\). Thus we are done. \(\Box\)

**Lemma 56.** If \(\text{EvolveG}(\mu_1, \mu'_1, \Sigma_1, F_1, F_2)\), \(\text{EvolveG}(\mu_2, \mu'_2, \Sigma_2, F_2, F_3)\), \(\mu_1 = (S_1, S_2, f_1)\), \(\mu_2 = (S_3, S_4, f_2)\), \(\mu = (S_1, S_3, f_2 \circ f_1)\), \(\mu'_1 = (S'_1, S'_2, f'_1)\), \(\mu'_2 = (S'_3, S'_4, f'_2)\), and \(\mu' = (S'_1, S'_3, f'_2 \circ f'_1)\), then \(\text{EvolveG}(\mu, \mu', \Sigma_1, \Sigma_2, F_1, F_3)\).

**Proof.** From \(\text{EvolveG}(\mu_1, \mu'_1, \Sigma_1, F_1, F_2)\) and \(\text{EvolveG}(\mu_2, \mu'_2, \Sigma_2, F_2, F_3)\), we know
Thus we are done.

\[ \text{Lemma 57. If } \text{FPmatch}(\mu, (\Lambda, \Sigma'), (\delta, F)), \text{dom}(\mu, \mathcal{S}) = \text{dom}(\Sigma), \text{forward}(\Sigma, \Sigma') \text{ and } (\text{locs}(\Sigma) - \text{locs}(\Sigma')) \subseteq \Delta. \text{ws} \cap \Delta. \text{rs}, \text{then FPmatch}(\mu, (\Lambda, \Sigma), (\delta, F)). \]

**Proof.** From FPmatch(\mu, (\Lambda, \Sigma'), (\delta, F)), we know

\[ \frac{\delta. \text{rs} \cup \delta. \text{ws} \subseteq \| F \| \cup \mu. \mathcal{S} \subseteq \text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \|}{\delta. \text{rs} \cap \mu. \mathcal{S} \subseteq \text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \| \subseteq \text{locs}(\Sigma) \cap \delta. \text{rs} \cap \mu. \mathcal{S} \subseteq \| \mu. \mathcal{S} \|.} \]

From forward(\Sigma, \Sigma'), we know locs(\Sigma') \cap \| \text{dom}(\Sigma) \| \subseteq \text{locs}(\Sigma). Since \mu. \mathcal{S} \subseteq \text{dom}(\Sigma), we know

\[ \text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \| \subseteq \text{locs}(\Sigma) \cap \| \mu. \mathcal{S} \|. \]

Thus

\[ \frac{\mu. \mathcal{S} \subseteq \text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \|}{\| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \text{locs}(\Sigma) \cap \| \mu. \mathcal{S} \|.} \]

Thus we have

\[ \delta. \text{rs} \cap \delta. \text{ws} \subseteq \| F \| \cup \mu. \mathcal{S} \subseteq \text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \|. \]

Since (locs(\Sigma) - locs(\Sigma')) \subseteq \Delta. \text{ws} \cap \Delta. \text{rs}, we know

\[ \text{locs}(\Sigma) \cap \| \mu. \mathcal{S} \| \subseteq (\text{locs}(\Sigma') \cap \| \mu. \mathcal{S} \|) \cup (\Delta. \text{ws} \cap \Delta. \text{rs} \cap \text{locs}(\Sigma)). \]

Thus

\[ \frac{\delta. \text{rs} \cap \mu. \mathcal{S} \subseteq \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \text{locs}(\Sigma) \cap \| \mu. \mathcal{S} \|.}{\delta. \text{rs} \cap \mu. \mathcal{S} \subseteq \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \| \mu. \mathcal{S} \| \subseteq \text{locs}(\Sigma) \cap \| \mu. \mathcal{S} \|.} \]

Thus we are done. \( \square \)

### E.3 Proof of Compositionality of Simulation (Lemma 48)

For any \( T, t, d \) and \( \Sigma, \) if (let \( \Gamma \text{ in } f_1 | \ldots | f_n, \mathcal{S} \) ) \( \Rightarrow (T, t, d, \Sigma, \) ) we know there exist \( F_1, \ldots, F_n, \mathcal{K}_1, \ldots, \mathcal{K}_n \) such that

\[ \text{initList}(\Sigma, (F_1, \ldots, F_n)), \quad \forall i \in \{1, \ldots, n\}. \mathcal{K}_i = \text{initRtMd}(\Gamma, f_i, F_i), \]

\[ T = \{ t_1 \sim \mathcal{K}_1, \ldots, t_n \sim \mathcal{K}_n \}, \quad t \in \{ t_1, \ldots, t_n \}, \quad d = \{ t_1 \sim 0, \ldots, t_n \sim 0 \}. \]

From initList(\Sigma, (F_1, \ldots, F_n)), we know

\[ \forall i \in \{1, \ldots, n\}. (\text{dom}(\Sigma) \cap F_i = \emptyset) \land (\forall j \neq i. F_i \cap F_j = \emptyset). \]

From \( \forall i \in \{1, \ldots, n\}. \mathcal{K}_i = \text{initRtMd}(f_i, F_i, F_i), \) we know there exist \( a_1, \ldots, a_n, k_1, \ldots, k_n \) such that

\[ \forall i \in \{1, \ldots, m\}. a_i \in \{1, \ldots, m\} \land k_i = \text{sl} \cdot \text{InitCore}(f_i, a_i) \land \mathcal{K}_i = (\text{sl} \cdot \text{InitCore}(f_i), a_i). \]

Similarly, for any \( T \) and \( \sigma, \) if (let \( \Pi \text{ in } f_1 | \ldots | f_n, \sigma \) ) \( \Rightarrow (T, t, d, \sigma, \) ) we know there exist \( F_1, \ldots, F_n, K_1, \ldots, K_n \) such that

\[ \text{initList}(\sigma, (F_1, \ldots, F_n)), \quad \forall i \in \{1, \ldots, n\}. K_i = \text{initRtMd}(\Pi, f_i, F_i), \quad T = \{ t_1 \sim K_1, \ldots, t_n \sim K_n \}. \]

From initList(\sigma, (F_1, \ldots, F_n)), we know

\[ \forall i \in \{1, \ldots, n\}. (\text{dom}(\sigma) \cap F_i = \emptyset) \land (\forall j \neq i. F_i \cap F_j = \emptyset). \]

From \( \forall i \in \{1, \ldots, n\}. K_i = \text{initRtMd}(\Pi, f_i, F_i), \forall i \in \{1, \ldots, m\}. \text{dom}(\text{sl} \cdot \text{InitCore}(f_i)) = \text{dom}(\text{tl} \cdot \text{InitCore}(\pi_i)) \) and the assumption that entries of different modules are different, we know there exist \( k_1, \ldots, k_n \) such that

\[ \forall i \in \{1, \ldots, n\}. k_i = \text{tl} \cdot \text{InitCore}(\pi_i, f_i) \land K_i = (\text{tl} \cdot (\pi_i, F_i, k_i)). \]
Towards Certified Separate Compilation for Concurrent Programs

For any \( \mu, \Sigma \) and \( \mathbb{S} \), if \( ge \subseteq \text{locs}(\Sigma) \), \( \text{dom}(\Sigma) = \mathbb{S} \subseteq \text{dom}(\varphi) \), \( \text{cl}(\mathbb{S}, \Sigma) = \mathbb{S} \), \( \text{locs}(\sigma) = \varphi(\text{locs}(\Sigma)) \), \( \text{Inv}(\varphi, \Sigma, \sigma) \) and \( \mu = (\mathbb{S}, \varphi(\mathbb{S}), \varphi(\delta)) \), since \( ge = \bigcup_i ge_i \), we know

\[
\forall i \in \{1, \ldots, m\}, \geq_i \subseteq \text{locs}(\Sigma) .
\]

From

\[
\forall a_i \in \{1, \ldots, m\}, (sl_{a_i}, ge_{a_i}, \gamma_{a_i}) \triangleq_{\varphi} (t_{a_i}, ge_{a_i}, \pi_{a_i}) ,
\]

we know

\[
\forall a_i \in \{1, \ldots, m\}, \exists j_i \in \text{index}. (F_i, (k_{j_i}, \Sigma), \text{emp}, \bullet) \triangleq_{\mu} (F_i, (k_{j_i}, \sigma), \text{emp}, \bullet). \]

Since \( t \in \{t_1, \ldots, t_n\} \), from

\[
(F_t, (k_t, \Sigma), \text{emp}, \bullet) \triangleq_{\mu} (F_t, (k_t, \sigma), \text{emp}, \bullet) ,
\]

we know there exists \( j \) such that

\[
(F_t, (k_t, \Sigma), \text{emp}, \circ) \triangleq_{\mu} (F_t, (k_t, \sigma), \text{emp}, \circ) .
\]

Then, by the following Lemma 58, we know

\[
((T, t, d, \Sigma), \text{emp}) \triangleq_{\mu} ((T, t, d, \sigma), \text{emp}) .
\]

**Lemma 58.** If

1. \( T = \{t_1 \sim (sl_{t_1}, F_{t_1}, k_{t_1}), \ldots, t_n \sim (sl_{t_n}, F_{t_n}, k_{t_n})\} \), \( T = \{t_1 \sim (t_{a_1}, F_{a_1}, k_{a_1}), \ldots, t_n \sim (t_{a_n}, F_{a_n}, k_{a_n})\} \),
2. \( t \in \{t_1, \ldots, t_n\} \), \( (F_t, (k_t, \Sigma), \Delta_0, \circ) \triangleq_{\mu} (F_t, (k_t, \sigma), \delta_0, \circ) \), and \( \text{RC}(F_t, \mu, \Sigma, (k_t, \Sigma), \circ) \),
3. for any \( t' \neq t \), there exists \( \delta' \in \text{index} \) such that \( (F_{t'}, (k_{t'}, \Sigma_0), \text{emp}, \bullet) \triangleq_{\mu} (F_{t'}, (k_{t'}, \sigma_0), \text{emp}, \bullet) \), and \( \text{RC}(F_{t'}, \mu, \Sigma, (k_{t'}, \Sigma_0), \bullet) \),
4. \( \forall t' \neq t \), \( (F_{t'} \cap F_{t} = \emptyset) \lor (F_{t'} \cap F_{t} = \emptyset) \), and for any \( t \), we have \( \text{wd}(sl_t) \) and \( \text{wd}(t_{a_t}) \),
5. \( \Sigma_{0} \xrightarrow{L_{\Delta_0, \circ}} \Sigma_{0}, \sigma_{0} \xrightarrow{L_{\Delta_0, \circ}} \sigma, \text{forward}(\Sigma_{0}, \Sigma), \text{forward}(\sigma_{0}, \sigma) \),

then \( ((T, t, d, \Sigma), \Delta_0) \triangleq_{\mu} ((T, t, d, \sigma), \Delta_0) \).

**Proof.** By co-induction. Let \( \hat{W} = (T, t, d, \Sigma) \) and \( \hat{W} = (T, t, d, \sigma) \). We need to prove the following:

1. if \((T, t, d, \Sigma) \xrightarrow{L} \hat{W}'\), then one of the following holds:
    a. there exists \( j \) such that
       i. \( j < i \), and
       ii. \( (\hat{W}', \Delta_0 \cup \Delta) \triangleq_{\mu} (\hat{W}, \delta_0) \);
    b. or, there exist \( \hat{W}', \delta \) and \( j \) such that
       i. \((T, t, d, \sigma) \xrightarrow{\delta} \hat{W}'\), and
       ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F_t)) \), and
       iii. \((\hat{W}', \Delta_0 \cup \Delta) \triangleq_{\mu} (\hat{W}', \delta_0 \cup \delta) \);

**Proof.** From \((T, t, d, \Sigma) \xrightarrow{L} \hat{W}'\), we know there exist \( k_{t'} \) and \( \Sigma' \) such that

\[
\hat{W}' = (T'(t \sim (sl_{t'}, F_{t'}, k_{t'})), t, d, \Sigma'), \quad F_{t'} \vdash (k_{t'}, \Sigma) \xrightarrow{\Delta} (k_{t'}, \Sigma')
\]

From \((F_{t'}, (k_{t'}, \Sigma), \Delta_0, \circ) \triangleq_{\mu} (F_{t'}, (k_{t'}, \sigma), \delta_0, \circ) \) and \( \text{RC}(F_{t'}, \mu, \Sigma, (k_{t'}, \Sigma), \circ) \), we know one of the following holds:

\(A\) there exists \( j \) such that

1. \( j < i \), and
2. \((F_{t'}, (k_{t'}, \Sigma'), \Delta_0 \cup \Delta) \triangleq_{\mu} (F_{t'}, (k_{t'}, \sigma), \delta_0, \circ) \);

\(B\) or, there exist \( k_{t'}, \sigma', \delta \) and \( j \) such that

3. \( F_t \vdash (k_{t'}, \sigma') \xrightarrow{\delta} (k_{t'}, \sigma') \), and
4. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F_t)) \), and
5. \((F_{t'}, (k_{t'}, \Sigma'), \Delta_0 \cup \Delta, \circ) \triangleq_{\mu} (F_{t'}, (k_{t'}, \sigma'), \delta_0 \cup \delta, \circ) \);

For case \(A\):

From \( F_{t'} \vdash (k_{t'}, \Sigma) \xrightarrow{\Delta} (k_{t'}, \Sigma') \), by \( \text{wd}(sl_t) \), we know

\[
\Sigma \xrightarrow{L_{\Delta_0, \circ}} \Sigma', \quad \text{forward}(\Sigma, \Sigma') .
\]
From $\Sigma_0 \vdash_{\Delta_0, ws} \Sigma$ and forward($\Sigma_0, \Sigma$), we know

$$\Sigma_0 \vdash_{\Delta_0, ws(\Delta_0, ws)} \Sigma', \quad \text{forward}(\Sigma_0, \Sigma').$$

From (2), by the co-induction hypothesis, we know

$$(\widetilde{W}', \Delta_0 \cup \Delta) \not\preceq^I_{\mu} (\widetilde{W}, \delta_0).$$

For case (B):

From (3), we know there exists $\widetilde{W}'$ such that

$$\widetilde{W}' = \{ T(t \triangleright (l_t, F_t, k'_t)) \mid t, d, \sigma' \}, \quad (T, t, d, \sigma) \not\preceq_{\mu}^I \widetilde{W}'$$

From $F_t \vdash_{(k_t, \Sigma)} \mu \mapsto \Sigma'$, by $\text{wd}(sl_t)$, we know

$$\Sigma \vdash_{\Delta, ws} \Sigma', \quad \text{forward}(\Sigma, \Sigma').$$

From $\Sigma_0 \vdash_{\Delta_0, ws} \Sigma$ and forward($\Sigma_0, \Sigma$), we know

$$\Sigma_0 \vdash_{\Delta_0, ws(\Delta, ws)} \Sigma', \quad \text{forward}(\Sigma_0, \Sigma').$$

Similarly, from $\sigma_0 \vdash_{\Delta_0, ws} \sigma$, forward($\sigma_0, \sigma$), and (3), by $\text{wd}(tl_t)$, we know

$$\sigma_0 \vdash_{\Delta, ws(\Delta, ws)} \sigma', \quad \text{forward}(\sigma_0, \sigma').$$

From (5), by the co-induction hypothesis, we know

$$(\widetilde{W}', \Delta_0 \cup \Delta) \not\preceq^I_{\mu} (\widetilde{W}, \delta_0 \cup \delta).$$

Thus we are done for this clause. 

---

2. $\forall t', d', o$. if $o \neq \tau$ and $\exists t'$. $(T, t, d, \Sigma) \vdash_{\text{emp}} (T', t', d', \Sigma)$, then there exist $\widetilde{W}'$, $\delta$, $\mu'$, $T'$, $\sigma'$ and $j$ such that for any $t' \in \text{dom}(T')$, 

a. $(T, t, d, \sigma) \not\preceq_{\mu} (T', t', d', \sigma')$, and
b. $\text{FPmatch}(\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, T(t.F)))$, and
c. $(T', t', d', \Sigma, \text{emp}) \not\preceq^I_{\mu} (T', t', d', \sigma', \text{emp})$.

**Proof.** Since $\exists t'$. $(T, t, d, \Sigma) \vdash_{\text{emp}} (T', t', d', \Sigma)$, we know one of the following two cases hold:

(A) there exist $i$ and $k'_t$ such that

$$i \neq \text{ret}, \quad i \neq \tau, \quad F_t \vdash_{(k_t, \Sigma)} i \mapsto (k'_t, \Sigma), \quad T' = T \{ t \triangleright (sl_t, F_t, k'_t) \}.$$

From $(F_t, (k_t, \Sigma), \Delta_0, o) \vdash_{\text{emp}} (F_t, (k_t, \Sigma), \delta_0, o)$ and $\text{RC}(F_t, \mu, \Sigma, (k_t, \Sigma), o)$, we know there exist $\kappa'_t, \sigma', \delta, \kappa'_t, \mu'$ and $j_t$ such that

1. $F_t \vdash (k_t, \sigma) \not\preceq_{\delta} (\kappa'_t, \sigma')$, and $F_t \vdash (\kappa'_t, \sigma') \vdash_{\text{emp}} \kappa'_t, \sigma'$, and
2. blocks($\Delta_0, rs \cup \Delta_0, ws$) $\subseteq F_t \cup \mu, \Sigma$, and
3. $\text{FPmatch}(\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F_t))$, and
4. $\text{EvolveG}(\mu, \mu', \Sigma, \sigma', F_t, \Sigma)$, and
5. $(F_t, (k'_t, \Sigma), \text{emp}, \text{emp}, \bullet) \vdash_{\mu'} (F_t, (k'_t, \sigma'), \text{emp}, \text{emp}, \bullet)$.

Let $\widetilde{W}' = T \{ t \triangleright (tl_t, F_t, k'_t) \}$, $(T, t, d, \sigma) \not\preceq_{\mu} (T', t', d', \sigma')$, and $T' = T \{ t \triangleright (tl_t, F_t, k'_t) \}$.

From (5), we know there exists $j'_t$ such that

$$(F_t, (k'_t, \Sigma), \text{emp}, o) \vdash_{\mu'} j'_t (F_t, (k'_t, \sigma'), \text{emp}, o).$$

For any $t' \neq t$, from $F_t' \cap F_t = \emptyset$ and $\text{blocks}(\Delta_0, ws) \subseteq F_t \cup \mu, \Sigma$, we know

$$\llbracket F_t' - \mu, \Sigma \rrbracket \subseteq \mu - \Delta_0, ws.$$
Towards Certified SeparateCompilation for Concurrent Programs

From $\Sigma_0 \xrightarrow{\delta_0, ws} \Sigma$, we know

$$\Sigma_0 \xrightarrow{\text{FPmatch}(\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F_t)))} \Sigma.$$  

For the target states $\sigma_t$ and $\sigma$, from FPmatch($\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F_t))$, we know

$$\delta_0, ws \cup \delta, ws \subseteq \|F_t\| \cup \mu.f \{\text{locs}(\Sigma) \cap \|\mu.S\|\}.$$  

Since forward($\Sigma_0, \Sigma$) and $\mu.S \subseteq \text{dom}(\Sigma_0)$, we know

$$\text{locs}(\Sigma) \cap \|\mu.S\| \subseteq \text{locs}(\Sigma_0) \cap \|\mu.S\|.$$  

Then, from $F_t \cap F_t = \emptyset$, we know

$$\|F_t\| - \mu.f \{\text{locs}(\Sigma_0) \cap \|\mu.S\|\} \subseteq \| - (\delta_0, ws \cup \delta, ws).$$  

From $\sigma_0 \xrightarrow{\delta_0, ws} \sigma$, forward($\sigma_0, \sigma$), and (1), by wd($t_0$), we know

$$\sigma_0 \xrightarrow{\|F_t\| - \mu.f \{\text{locs}(\Sigma_0) \cap \|\mu.S\|\}} \sigma', \text{ forward}(\sigma_0, \sigma').$$  

Thus

$$\sigma_0 \xrightarrow{\|F_t\| - \mu.f \{\text{locs}(\Sigma_0) \cap \|\mu.S\|\}} \sigma'.$$  

Also, since EvolveG($\mu, \mu', \Sigma, \sigma', F_t, F_t$) and $F_t \cap F_t = \emptyset$ and $F_t \cap F_t = \emptyset$, we know

$$\text{EvolveR}(\mu, \mu', \Sigma, \sigma', F_t, F_t).$$  

From ($F_t'\cup(k_{t'}, \Sigma_0, \text{emp}, \bullet) \triangleleft_{\mu'}^\delta (F_t', \sigma_t'\cup(k_{t'}, \sigma_0', \text{emp}, \bullet)$ and RC($F_t, \mu, \Sigma, (k_{t'}, \Sigma_0, \bullet)$, we know there exists $j_{t'}$ such that

$$(F_t', (k_{t'}, \Sigma), \text{emp}, \sigma) \triangleleft_{\mu'}^\delta (F_t', (k_{t'}, \sigma'), \text{emp}, \sigma).$$  

Also, by Lemma 59, we know

$$(F_t', (k_{t'}, \Sigma), \text{emp}, \bullet) \triangleleft_{\mu'}^\delta (F_t', (k_{t'}, \sigma'), \text{emp}, \bullet).$$  

Thus, for any $t'$ (no matter whether $t' = t$ or $t' \neq t$), by the co-induction hypothesis, we know there exists $j$ such that

$$(T', t', d', \Sigma), \text{emp}) \triangleleft_{\mu'}^\delta ((T', t', d', \sigma'), \text{emp}).$$  

(b) there exists $k_{t'}$ such that

$$F_t \rightarrow_{\text{ret}^{\text{emp}}} (k_{t'}, \Sigma), \quad T' = T \setminus t, \quad d' = d' \setminus t.$$  

From ($F_t', (k_{t'}, \Sigma, \Delta_0, \sigma) \triangleleft_{\mu'}^\delta (F_t, (k_{t}, \sigma, \delta_0, \sigma)$ and RC($F_t, \mu, \Sigma, (k_{t'}, \Sigma_0, \bullet)$, we know there exist $k_{t'}$, $\sigma'$, $\delta$, $k_{t''}$, $\mu'$ and $j_{t}$ such that

1. $F_t \rightarrow_{\text{ret}^{\text{emp}}} (k_{t'}, \sigma'), \text{ and } F_t \rightarrow_{\text{ret}^{\text{emp}}} (k_{t''}, \sigma')$, and
2. blocks($\Delta_0, rs \cup \Delta_0, ws) \subseteq F_t \cup \mu.S$, and
3. FPmatch($\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F_t)$), and
4. EvolveG($\mu, \mu', \Sigma, \sigma', F_t, F_t$).

Let $\tilde{W}' = (T \{ t \rightarrow (t, F_t, k_{t'}))^j, t, d', \sigma')$. From (1), we know there exists $T'$ such that for any $t' \in \text{dom}(T \setminus t)$, we have

$$(T', t, d, \sigma) \xrightarrow{\sigma_{\text{emp}}} (T', t', d', \sigma'), \quad T' = T \setminus t.$$  

Similarly as (a), we know there exists $j_{t'}$ such that

$$(F_t', (k_{t'}, \Sigma), \text{emp}, \sigma) \triangleleft_{\mu'}^\delta (F_t', (k_{t'}, \sigma'), \text{emp}, \sigma).$$  

Thus, by the co-induction hypothesis, we know

$$(T', t', d', \Sigma), \text{emp}) \triangleleft_{\mu'}^\delta ((T', t', d', \sigma'), \text{emp}).$$  

Thus we are done for this clause. 

3. if $(T, t, d, \Sigma) \xrightarrow{\text{emp}} \text{done}$, then there exist $\tilde{W}'$ and $\delta$ such that

a. $(T, t, d, \sigma) \xrightarrow{\delta} \tilde{W}'$, and $\tilde{W}' : \xrightarrow{\text{emp}} \text{done}$, and
b. FPmatch($\mu, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F_t)$). 

43
Figure 24. The execution in the auxiliary lemmas for the compositionality proofs.

Proof. From \((T, t, d, \Sigma) \xrightarrow{\tau \text{ emp}} \text{done}\), we know

\[ F_t \vdash (k_t, \Sigma) \xrightarrow{\text{ret emp}} (k_t', \Sigma), \quad \text{dom}(T) = \{ t \}. \]

From \((F_t, (k_t, \Sigma), \Delta_0, \circ) \preceq_{\mu} (F_t, (k_t, \sigma), \Delta_0, \circ)\) and RC\((F_t, \mu.\Sigma, (k_t, \Sigma), \circ)\), we know there exist \(k_t', \delta, \sigma', k_t''\) such that

\[ F_t \vdash (k_t, \sigma) \xrightarrow{\tau} (k_t', \sigma'), \quad F_t \vdash (k_t', \sigma') \xrightarrow{\text{ret emp}} (k_t'', \sigma'), \quad \text{FPmatch}(\mu, (\Delta_0, \Sigma), (\Delta_0 \cup \delta, F_t)). \]

Let \(\tilde{W}' = (T \mid t \leadsto (tl_t, F_t, k_t'))\), \(t, d, \sigma'\). Since \(\text{dom}(T) = \{ t \}\), we know

\[ (T, t, d, \sigma) \xrightarrow{\tau \delta} \tilde{W}', \quad \tilde{W}' \xrightarrow{\text{emp}} \text{done}. \]

Thus we are done. \(\square\)

Thus we have finished the proof. \(\square\)

Lemma 59 (From A to D in Fig. 24). If

1. \((\mathcal{F}_{\Sigma_A}, \text{emp, } \bullet) \preceq_{\mu} (F, (\kappa, \sigma_A), \text{emp, } \bullet), \)
2. \(\Sigma_A \vdash [F \mu_{\Sigma_A}] \mapsto \Sigma_D, \sigma_A \vdash [F \mu_{\Sigma_A}] (\text{box}(\Sigma_D), \mu_{\Sigma_D}) \mapsto \sigma_D, \text{forward}(\Sigma_A, \Sigma_D), \text{forward}(\sigma_A, \sigma_D), \text{and EvolveR}(\mu_A, \mu_D, \Sigma_D, \sigma_D, \mathcal{F}, F)\),

then \((\mathcal{F}_{\Sigma_D}, \text{emp, } \bullet) \preceq_{\mu} (F, (\kappa, \sigma_D), \text{emp, } \bullet)\).

Proof. By co-induction. From \((\mathcal{F}_{\Sigma_A}, \text{emp, } \bullet) \preceq_{\mu} (F, (\kappa, \sigma_A), \text{emp, } \bullet)\), we know

\[ \text{wf}(\mu_A, (\Sigma_A, \mathcal{F}), (\sigma_A, F)). \]

From EvolveR\((\mu_A, \mu_D, \Sigma_D, \sigma_D, \mathcal{F}, F)\), we know

\[ \text{wf}(\mu_D, (\Sigma_D, \mathcal{F}), (\sigma_D, F)), \quad \mu_A \cdot f \subseteq \mu_D \cdot f, \quad \text{Inv}(\mu_D \cdot f, \Sigma_D, \sigma_D), \]

\[ \mu_D \cdot S = \text{cl}(\mu_A \cdot S, \Sigma_D), \quad (\mu_D \cdot S - \mu_A \cdot S) \cap \mathcal{F} = \emptyset. \]

For any \(\Sigma_G, \Sigma_D, \mu_G\), suppose

\[ \Sigma_D \vdash [F \mu_{\Sigma_A}] \mapsto \Sigma_G, \sigma_D \vdash [F \mu_{\Sigma_A}] (\text{box}(\Sigma_D), \mu_{\Sigma_D}) \mapsto \sigma_G, \text{forward}(\Sigma_D, \Sigma_G), \text{forward}(\sigma_D, \sigma_G), \text{EvolveR}(\mu_D, \mu_G, \Sigma_G, \sigma_G, \mathcal{F}, F). \]

From \((\mu_D \cdot S - \mu_A \cdot S) \cap \mathcal{F} = \emptyset\) and \(\Sigma_D \vdash [F \mu_{\Sigma_A}] \mapsto \Sigma_G, \text{we know}\)

\[ \Sigma_D \vdash [F \mu_{\Sigma_A} \mapsto \Sigma_G. \]

From \(\Sigma_A \vdash [F \mu_{\Sigma_A} \mapsto \Sigma_D, \text{we know}\)

\[ \Sigma_A \vdash [F \mu_{\Sigma_A} \mapsto \Sigma_G. \]
Towards Certified Separate Compilation for Concurrent Programs

From \((\mu_D, S) - \mu_A, S) \cap F = \emptyset\), we know

\[
\mu_D, f \{\mu_D, S - \mu_A, S\} \subseteq \mu_D, f \{\mu_D, S - F\}.
\]

Since \(\mu_D, f \{\mu_D, S - F\} \cap F = \emptyset\), we know

\[
\mu_D, f \{\mu_D, S - \mu_A, S\} \cap F = \emptyset.
\]

Thus

\[
\mu_D, f \{\mu_D, S - \mu_A, S\} \cap \{\text{locs}(\Sigma_D) \cap \|\mu_D, S\|\} - \{\text{locs}(\Sigma_A) \cap \|\mu_A, S\|\} \cap \|F\| = \emptyset.
\]

From \(\mu_A, f \subseteq \mu_D, f\) and \(\mu_A, S \subseteq \text{dom}(\mu, f)\), we know

\[
(\mu_D, f \{\text{locs}(\Sigma_D) \cap \|\mu_D, S\|\} - \mu_A, f \{\text{locs}(\Sigma_A) \cap \|\mu_A, S\|\}) \cap \|F\| = \emptyset.
\]

From \(\sigma_D \longleftarrow_F \mu_D, f \{\text{locs}(\Sigma_D) \cap \|\mu_D, S\|\}\) \(\sigma_G\), we know

\[
\sigma_D \longleftarrow_F \mu_D, f \{\text{locs}(\Sigma_A) \cap \|\mu_A, S\|\}\ = \sigma_G.
\]

From \(\sigma_A \longleftarrow_F \mu_A, f \{\text{locs}(\Sigma_A) \cap \|\mu_A, S\|\}\) \(\sigma_D\), we know

\[
\sigma_A \longleftarrow_F \mu_A, f \{\text{locs}(\Sigma_A) \cap \|\mu_A, S\|\}\ = \sigma_G.
\]

From forward(\(\Sigma_A, \Sigma_D\)), forward(\(\sigma_A, \sigma_D\)), forward(\(\Sigma_D, \Sigma_G\)) and forward(\(\sigma_D, \sigma_G\)), we know

forward(\(\Sigma_A, \Sigma_G\)), forward(\(\sigma_A, \sigma_G\)).

Let \(\mu_E = (\Sigma_E, S_E, f_E)\), where \(\Sigma_E = \text{cl}(\mu_A, S, \Sigma_G)\), \(S_E = \text{cl}(\mu_A, S, \Sigma_G)\) and \(f_E = (\mu_G, f)|_{\Sigma_E}\). Since \(\mu_D, S = \text{cl}(\mu_A, S, \Sigma_D)\), we know \(\mu_A, S \subseteq \mu_D, S\). Since \(\mu_G, S = \text{cl}(\mu_D, S, \Sigma_G)\), we know \(S_E \subseteq \mu_G, S\). Similarly we know \(S_E \subseteq \mu_G, S\). Since \(\text{dom}(\mu_G, f) = \mu_G, S\), we know dom(\(f_E\) = \(S_E\). Since \(\mu_A, f \subseteq \mu_D, f \subseteq \mu_G, f\) and \(\mu_A, S \subseteq S_E\), we know

\[
\mu_A, f \subseteq f_E.
\]

Since \(\text{Inv}(\mu_G, f, \Sigma_G, \sigma_G)\), dom(\(\mu_A, f\) = \mu_A, S and \(\mu_A, f \{\mu_A, S\} \subseteq \mu_A, S\), by Lemma 61, we know \((\mu_G, f) \{\Sigma_E\} \subseteq S_E\). Thus

\[
f_E \{\Sigma_E\} \subseteq S_E.
\]

Since \(\text{wf}(\mu_G, (\Sigma_G, F), (\sigma_G, F))\), we know

\[
\text{wf}(\mu_E, (\Sigma_G, F), (\sigma_G, F)).
\]

Since \(\text{Inv}(\mu_G, f, \Sigma_G, \sigma_G)\) and \(\Sigma_E = \text{cl}(\Sigma_E, \Sigma_G)\), by Lemma 62, we know

\[
\text{Inv}(f_E, \Sigma_G, \sigma_G).
\]

Since \(S_E \subseteq \mu_G, S\), \((\mu_G, S - \mu_D, S) \cap F = \emptyset\) and \((\mu_D, S - \mu_A, S) \cap F = \emptyset\), we know

\[
(S_E - \mu_A, S) \cap F = \emptyset.
\]

Thus we know

\[
\text{EvolveR}(\mu_A, \mu_E, \Sigma_G, \sigma_G, F).
\]

From \((F, (k, \Sigma_A), \text{emp}, \bullet) \preceq^{i}_{\mu_A} (F, (k, \sigma_A), \text{emp}, \bullet)\), we know there exists \(j\) such that

\[
(F, (k, \Sigma_G), \text{emp}, o) \preceq^{i}_{\mu_E} (F, (k, \sigma_G), \text{emp}, o).
\]

Since \((\mu_G, S - \mu_D, S) \cap F = \emptyset\), \((\mu_D, S - \mu_A, S) \cap F = \emptyset\) and \((S_E - \mu_A, S) \cap F = \emptyset\), we know

\[
(\mu_G, S - S_E) \cap F = \emptyset.
\]

By Lemma 60, we know

\[
(F, (k, \Sigma_G), \text{emp}, o) \preceq^{i}_{\mu_G} (F, (k, \sigma_G), \text{emp}, o).
\]

Thus we are done. \(\square\)
Lemma 60 (From E to G in Fig. 24). If

1. \( (F, (k, \Sigma), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \),

2. \( \Sigma_G \xrightarrow{H - \Delta_0 \text{ ws} \Delta} \Sigma, \sigma_G \xrightarrow{H - \Delta_0 \text{ ws} \Delta} \sigma \), forward\((\Sigma_G, \Sigma)\), forward\((\sigma_G, \sigma)\), \( \mu_E = (\Sigma_E, \Delta_0, \circ, \delta_0, \circ) \), \( \mu_G = (\Sigma_G, \Delta_0, \circ) \), \( \Sigma_F = \{ (\Sigma_E, \Delta_0, \circ) \} \), \( \Sigma_G = \{ (\Sigma_G, \Delta_0, \circ) \} \), \( \delta_0 = \delta \). 

Then \( (F, (k, \Sigma), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \).

Proof. By co-induction. Since forward\((\Sigma_G, \Sigma)\) and \( \text{wf}(\mu_G, (\Sigma_G, F), (\sigma, F)) \), we know

\[ \text{wf}(\mu_G, (\Sigma, F), (\sigma, F)). \]

Also we need to prove the following.

1. \( \forall k', \Sigma', \Delta. \text{ if } (k, \Sigma) \xrightarrow{\Delta} (k', \Sigma') \), then one of the following holds:
   a. there exists \( j \) such that
      i. \( j < i \), and
      ii. \( (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \);
   b. or, there exist \( k', \sigma', \delta \) and \( j \) such that
      i. \( F \vdash (k, \sigma) \xrightarrow{\delta} (k', \sigma') \), and
      ii. \( \text{FPmatch}(\mu_E, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)) \), and
      iii. \( (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k', \sigma'), \Delta_0, \circ) \).

Proof. From \( (F, (k, \Sigma), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \), we know one of the following holds:

(A) there exists \( j \) such that

1. \( j < i \), and
2. \( (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \);

(B) or, there exist \( k', \sigma', \delta \) and \( j \) such that

3. \( F \vdash (k, \sigma) \xrightarrow{\delta} (k', \sigma') \), and
4. \( \text{FPmatch}(\mu_E, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)) \), and
5. \( (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k', \sigma'), \Delta_0, \circ) \).

For case (A):

From \( (F, (k, \Sigma), \Delta_0, \circ) \xrightarrow{\Delta} (k', \Sigma') \), by \( \text{wd}(sl) \), we know

\[ \Sigma \xrightarrow{H - \Delta \text{ ws} \Delta} \Sigma' \text{, } \text{forward}(\Sigma, \Sigma'). \]

Since \( \Sigma_G \xrightarrow{H - \Delta_0 \text{ ws} \Delta} \Sigma \) and forward\((\Sigma_G, \Sigma)\), we know

\[ \Sigma_G \xrightarrow{H - (\Delta_0 \cup \Delta) \text{ ws} \Delta} \Sigma' \text{, } \text{forward}(\Sigma_G, \Sigma'). \]

From (2), by the co-induction hypothesis, we know

\[ (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k, \sigma), \Delta_0, \circ) \text{.} \]

For case (B):

Since \( (F, (k, \Sigma), \Delta_0, \circ) \xrightarrow{\Delta} (k', \Sigma') \), \( \Sigma_G \xrightarrow{H - \Delta_0 \text{ ws} \Delta} \Sigma \) and forward\((\Sigma_G, \Sigma)\), we know

\[ \Sigma_G \xrightarrow{H - (\Delta_0 \cup \Delta) \text{ ws} \Delta} \Sigma' \text{, } \text{forward}(\Sigma_G, \Sigma'). \]

From (3), \( \sigma_G \xrightarrow{H - \Delta_0 \text{ ws} \Delta} \sigma \) and forward\((\sigma_G, \sigma)\), by \( \text{wd}(tl) \), we know

\[ \sigma_G \xrightarrow{H - (\delta_0 \cup \delta) \text{ ws} \Delta} \sigma' \text{, } \text{forward}(\sigma_G, \sigma'). \]

From (5), by the co-induction hypothesis, we know

\[ (F, (k', \Sigma'), \Delta_0, \circ) \preceq_{H,F} (F, (k', \sigma'), \Delta_0, \circ) \text{.} \]

From (4), by Lemma 63, we know

\[ \text{FPmatch}(\mu_G, (\Delta_0 \cup \Delta, \Sigma), (\delta_0 \cup \delta, F)). \]

Thus we have done for this clause.
Towards Certified Separate Compilation for Concurrent Programs

2. \( \forall k', i \text{ if } i \neq \tau \text{ and } \mathcal{F} \vdash (k, \Sigma) \rightarrow_{\text{emp}} (k', \Sigma) \),

then there exist \( \kappa', \delta, \sigma', \kappa', \mu_H \) and \( j \) such that

a. \( \mathcal{F} \vdash (k, \sigma) \rightarrow_{\delta} (k', \sigma') \), and

\( \mathcal{F} \vdash (k', \sigma') \rightarrow_{\text{emp}} (k'', \sigma') \), and

b. \( \text{blocks}(\Delta_0, rs \cup \Delta_0, ws) \subseteq \mathcal{F} \cup S_G \) and

c. \( \text{FPmatch}(\mu_G, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F)) \), and

(d) \( \text{EvolveG}(\mu_G, \mu_H, \Sigma, \sigma', \mathcal{F}, F) \), and

e. \( (\mathcal{F}, (k', \Sigma), \text{emp}, \bullet) \rightarrow_{\mu_H} (F, (k'', \sigma'), \text{emp}, \bullet) \).

Proof. From \( (\mathcal{F}, (k, \Sigma), \Delta_0, \circ) \rightarrow_{\mu_F} (F, (k, \sigma), \delta_0, \circ) \), we know there exist \( \kappa', \delta, \sigma', \kappa', \mu_F \) and \( j \) such that

(1) \( \mathcal{F} \vdash (k, \sigma) \rightarrow_{\delta} (k', \sigma') \), and

\( \mathcal{F} \vdash (k', \sigma') \rightarrow_{\text{emp}} (k'', \sigma') \), and

(2) \( \text{blocks}(\Delta_0, rs \cup \Delta_0, ws) \subseteq \mathcal{F} \cup S_E \) and

(3) \( \text{FPmatch}(\mu_E, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F)) \), and

(4) \( \text{EvolveG}(\mu_E, \mu_F, \Sigma, \sigma', \mathcal{F}, F) \), and

(5) \( (\mathcal{F}, (k', \Sigma), \text{emp}, \bullet) \rightarrow_{\mu_F} (F, (k'', \sigma'), \text{emp}, \bullet) \).

From (2), since \( S_E \subseteq S_G \), we know

\( \text{blocks}(\Delta_0, rs \cup \Delta_0, ws) \subseteq \mathcal{F} \cup S_G \) .

From (3), by Lemma 63, we know

\( \text{FPmatch}(\mu_G, (\Delta_0, \Sigma), (\delta_0 \cup \delta, F)) \) .

Suppose \( \mu_F = (S_F, S_F, f_F) \). From (4), we know

\( \text{wd}(\mu_F, (\Sigma, \mathcal{F}), (\sigma', F)) \), \( f_E \subseteq f_F \) \text{ and } \( \text{Inv}(f_F, \Sigma, \sigma') \),

\( S_F = \text{cl}(S_E, \Sigma) \), \( S_E = \text{cl}(S_E, \sigma') \), \( (S_F \cup S_E) \subseteq \mathcal{F} \) .

Let \( \mu_H = (S_H, S_H, f_H) \), where \( S_H = \text{cl}(S_G, \Sigma) \), \( S_H = \text{cl}(S_G, \sigma') \) and \( f_H = f_G \cup (f_F|_{S_H \cup S_G}) \). Thus we know \( S_G \subseteq S_H \), \( S_G \subseteq S_H \) and \( f_G \subseteq f_H \). Since \( S_E \subseteq S_G \) and \( S_E \subseteq S_G \), we know \( S_F \subseteq S_H \) and \( S_F \subseteq S_H \). By Lemma 64, we know

\( S_H - S_G \subseteq S_F \).

Since \( S_E \subseteq S_G \) and \( (S_F - S_E) \subseteq \mathcal{F} \), we know

\( S_H - S_G \subseteq S_F \cup S_E \subseteq \mathcal{F} \).

Since \( S_F \subseteq S_H \) and \( (S_G - S_E) \cap \mathcal{F} = \emptyset \), we know

\( S_F - S_E \subseteq (S_H - S_E) \cap \mathcal{F} = ((S_H - S_G) \cup (S_G - S_E)) \cap \mathcal{F} = S_H - S_G \).

Also we know \( \text{dom}(f_H) = S_H \). Since \( f_F \{S_F\} \subseteq S_F \), we know

\( f_F\{S_H - S_G\} \subseteq S_F = \text{cl}(S_E, \sigma') \subseteq \text{cl}(S_G, \sigma') = S_H \).

Since \( f_G\{S_G\} \subseteq S_G \), we know

\( f_H\{S_H\} \subseteq S_H \).

Also we know

\( f_H = f_G \cup (f_F|_{S_H - S_E}) = (f_G|_{S_G - S_E}) \cup f_E \cup (f_F|_{S_H - S_E}) = (f_G|_{S_G - S_E}) \cup f_F \).

Since \( \text{injective}(f_G, \Sigma, G) \), \( \text{dom}(f_G) = S_G \subseteq \text{dom}(\Sigma_G) \) and \( \text{forward}(\Sigma_G, \Sigma) \), we know

\( \text{injective}(f_G, \Sigma) \).

Since \( (S_G - S_E) \cap \mathcal{F} = \emptyset \), \( f_G\{S_G - \mathcal{F}\} \cap F = \emptyset \), \( (S_H - S_G) \subseteq \mathcal{F} \) and \( f_F\{S_F \cap \mathcal{F}\} \subseteq F \), we know

\( f_G\{S_G - S_E\} \cap F = \emptyset \), \( f_F\{S_H - S_G\} \subseteq F \).

Since \( f_E \subseteq f_F \) and \( \text{injective}(f_F, \Sigma) \), we know

\( \text{injective}(f_H, \Sigma) \).

Since \( S_F \subseteq \text{dom}(\Sigma) \), \( S_G \subseteq \text{dom}(\Sigma) \) and \( \text{forward}(\Sigma_G, \Sigma) \), we know

\( S_H \subseteq \text{dom}(\Sigma) \).

From (1), by \( \text{wd}(tl) \), we know

\( \sigma \xrightarrow{H-\delta, ws} \sigma' \), \( \text{forward}(\sigma, \sigma') \).
Since $\sigma_G \xrightarrow{H-\delta_0, ws} \sigma$ and forward($\sigma_G, \sigma$), we know
$$\sigma_G \xrightarrow{H-(\delta_0 \cup \delta), ws} \sigma', \quad \text{forward($\sigma_G, \sigma'$)}.$$  

From (3), we know
$$(\delta_0 \cup \delta), ws \subseteq \|F\| \cup f_G(\{\text{locs($\Sigma$)} \cap \|S_E\|\}).$$  

Since $f_G\{S_G - S_E\} \cap F = \emptyset$ and injective($f_G, \Sigma$), we know
$$f_G(\{\text{locs($\Sigma$)} \cap \|S_G - S_E\|\}) \cap (\|F\| \cup f_G(\{\text{locs($\Sigma$)} \cap \|S_E\|\})) = \emptyset.$$  

Thus
$$f_G(\{\text{locs($\Sigma$)} \cap \|S_G - S_E\|\}) \subseteq \|H - (\delta_0 \cup \delta), ws\|.$$  

As a result, since $f_H = (f_G|_{S_G - S_E}) \cup f_F$, forward($\Sigma_G, F$), $f_G(\{\text{locs($\Sigma$)}\}) \subseteq \text{locs($\sigma_G$)}$ and $f_F(\{\text{locs($\Sigma$)}\}) \subseteq \text{locs($\sigma'$)}$, we know
$$f_H(\{\text{locs($\Sigma$)}\}) = f_G(\{\text{locs($\Sigma$)} \cap \|S_G - S_E\|\}) \cup f_F(\{\text{locs($\Sigma$)} \cap \|S_E\|\} \cap \text{locs($\sigma'$)} \subseteq (\|H - (\delta_0 \cup \delta), ws\| \cap \text{locs($\sigma_G$)}) \cup \text{locs($\sigma'$)} \subseteq \text{locs($\sigma'$)}.$$  

Thus we have proved $\text{wf($\mu_H, \Sigma, \sigma'$)}$. Since $f_H = f_G \cup (f_F|_{S_G - S_E})$, $f_G\{S_G \cap F\} \subseteq F$ and $f_F\{S_H - S_G\} \subseteq F$, we know
$$f_H\{S_G \cap F\} \subseteq f_G\{S_G \cap F\} \cup f_F\{(S_H - S_G) \cap F\} \subseteq F.$$  

Also, since $f_G\{S_G - F\} \cap F = \emptyset$ and $(S_H - S_G) \subseteq F$, we know
$$f_H\{S_H - F\} \cap F = (f_G\{S_G - F\} \cup f_F\{(S_H - S_G) \cap F\}) \cap F = \emptyset.$$  

Thus we have proved $\text{wf($\mu_H, (\Sigma, F), (\sigma', F)$)}$. Since $\text{Inv}(f_G, \Sigma, \sigma_G)$ and forward($\Sigma_G, \Sigma$), we know: for any $b, n, b'$ and $n'$, if $(b, n) \in \text{locs($\Sigma$)}$ and $(f_G|_{S_G - S_E})(b, n) = (b', n')$, we know
$$(b', n') \in \text{locs($\sigma_G$)}, \quad \Sigma_G(b)(n) \xrightarrow{f_G} \sigma_G(b')(n').$$  

Since blocks($\Delta_0, ws$) $\subseteq F \cup S_E$ and $(S_G - S_E) \cap F = \emptyset$, we know
$$\|S_G - S_E\| \subseteq \|H - \Delta_0, ws\|.$$  

Since $\Sigma_G \xrightarrow{H-\Delta_0, ws} \Sigma$ and $b \in S_G - S_E$, we know
$$\Sigma_G(b)(n) = \Sigma(b)(n).$$  

Since $f_G(\{\text{locs($\Sigma$)} \cap \|S_G - S_E\|\}) \subseteq \|H - (\delta_0 \cup \delta), ws\| \cap \text{locs($\sigma_G$)}$, we know
$$(b', n') \in \text{locs($\sigma'$)}, \quad \sigma_G(b')(n') = \sigma'(b')(n').$$  

Thus we know
$$\Sigma(b)(n) \xrightarrow{f_G} \sigma'(b')(n').$$  

Since $f_H = f_G \cup (f_F|_{S_G - S_E}) = (f_G|_{S_G - S_E}) \cup f_F$ and $\text{Inv}(f_H, \Sigma, \sigma')$, we know
$$\text{Inv}(f_H, \Sigma, \sigma').$$  

Thus we have proved $\text{EvolveG($\mu_G, \mu_H, \Sigma, \sigma', F, F$)}$. Also, since $S_F - S_E = S_H - S_G$ and $(S_G - S_E) \cap F = \emptyset$, we know
$$S_H - S_F \cap F = \emptyset.$$  

Below we consider the case when $(F, (\kappa', \Sigma), \text{emp}, *) \xleftarrow{f_F} (F, (\kappa'', \sigma', \text{emp}, *))$. We prove
$$(F, (\kappa', \Sigma), \text{emp}, *) \xrightarrow{\mu_H} (F, (\kappa'', \sigma', \text{emp}, *)) \quad \text{(E.2)}$$  

Proof: by co-induction. For any $\sigma_j, \Sigma_j, \mu_j$, suppose $\mu_j = (\Sigma_j, S_j, f_j)$ and
$$\Sigma \xrightarrow{\|S_H\|} \Sigma_j, \sigma' \xrightarrow{\|F\| \cup \text{forward($\Sigma_j$)}} \|S_j\|, \text{forward($\Sigma_j$, forward($\sigma_j$, $\sigma_j$), $\text{EvolveR($\mu_H, \mu_j, \Sigma_j, \sigma_j, F, F$)}$).}$$  

Since $(S_H - S_F) \cap F = \emptyset$ and $\Sigma \xrightarrow{\|S_H\|} \Sigma_j$, we know
$$\Sigma \xrightarrow{\|S_H\|} \Sigma_j.$$  

Since $f_H\{S_H - F\} \cap F = \emptyset$, we know
Towards Certified Separate Compilation for Concurrent Programs

\[ f_H \{ S_H - S_F \} \cap F = \emptyset . \]

Since \( f_F \subseteq f_H \) and \( S_F \subseteq \text{dom}(f_F) \), we know

\[ (f_H \{ \text{locs}(\Sigma) \cap \| S_H \| \} - f_F \{ \text{locs}(\Sigma) \cap \| S_F \| \}) \cap \| F \| = \emptyset . \]

Since \( \sigma' \| F - f_F \{ \text{locs}(\Sigma) \cap \| S_F \| \} \sigma_j \), we know

\[ \sigma' \| F - f_F \{ \text{locs}(\Sigma) \cap \| S_F \| \} \sigma_j \).

Let \( \mu_I = (S_I, S_I, f_I) \), where \( S_I = \text{cl}(S_F, \Sigma_I) \), \( S_I = \text{cl}(S_F, \sigma_I) \) and \( f_I = f_I|_{S_I} \). Since \( S_F \subseteq S_H \), \( S_F \subseteq S_H \), \( S_J = \text{cl}(S_H, \Sigma_J) \) and \( S_J = \text{cl}(S_H, \sigma_J) \), we know

\[ S_I \subseteq S_J, \quad S_I \subseteq S_J. \]

Since \( \text{dom}(f_I) = S_J \), we know

\[ \text{dom}(f_I) = S_I. \]

Since \( f_F \subseteq f_H \), \( f_H \subseteq f_I \) and \( S_F \subseteq S_I \), we know

\[ f_F \subseteq f_I \subseteq f_J. \]

Since \( \text{Inv}(f_I, \Sigma_I, \sigma_I) \), \( \text{dom}(f_F) = S_F \) and \( f_F \{ S_F \} \subseteq S_F \), by Lemma 61, we know \( f_J \{ S_I \} \subseteq S_I \). Thus

\[ f_I \{ S_I \} \subseteq S_I. \]

Since \( \text{wf}(\mu_I, (\Sigma_I, F), (\sigma_I, F)) \), we know

\[ \text{wf}(\mu_I, (\Sigma_I, F), (\sigma_I, F)). \]

Since \( \text{Inv}(f_I, \Sigma_I, \sigma_I) \) and \( S_I = \text{cl}(S_I, \Sigma_I) \), by Lemma 62, we know

\[ \text{Inv}(f_I, \Sigma_I, \sigma_I). \]

Since \( S_I \subseteq S_J \), \( (S_J - S_H) \cap F = \emptyset \) and \( (S_H - S_F) \cap F = \emptyset \), we know

\[ (S_I - S_J) \cap F = \emptyset . \]

Thus we know

\[ \text{Evolve}(\mu_F, \mu_I, \Sigma_I, \sigma_I, F). \]

From \( (F, (k', \Sigma), \text{emp}, \bullet) \prec_{\mu_F} (F, (k'', \sigma'), \text{emp}, \bullet) \), we know there exists \( j' \) such that

\[ (F, (k', \Sigma_I), \text{emp}, \circ) \prec_{j'} (F, (k'', \sigma_I), \text{emp}, \circ). \]

Since \( (S_I - S_F) \cap F = \emptyset \), \( (S_J - S_H) \cap F = \emptyset \) and \( (S_H - S_F) \cap F = \emptyset \), we know

\[ (S_I - S_J) \cap F = \emptyset . \]

Thus by the co-induction hypothesis, we know

\[ (F, (k', \Sigma_I), \text{emp}, \circ) \prec_{j'} (F, (k'', \sigma_I), \text{emp}, \circ). \]

Thus we have proved (E.2). Thus we have done for this clause. \( \square \)

Thus we are done. \( \square \)

**Lemma 61.** If \( f_1 \subseteq f_2 \), \( \text{dom}(f_1) = S_1, f_1 \{ S_1 \} \subseteq S_1, S_2 = \text{cl}(S_1, \Sigma) \subseteq \text{dom}(f_2), S_2 = \text{cl}(S_1, \sigma) \) and \( \text{Inv}(f_2, \Sigma, \sigma) \), then \( f_2 \{ S_2 \} \subseteq S_2 \).

**Proof:** We prove: for any \( k, f_2 \{ \text{cl}_k(S_1, \Sigma) \} \subseteq \text{cl}_k(S_1, \sigma) \). By induction over \( k \).

- Base case: \( k = 0 \). From \( f_1 \subseteq f_2 \), \( \text{dom}(f_1) = S_1 \) and \( f_1 \{ S_1 \} \subseteq S_1 \), we are done.

- Inductive step: \( k = n + 1 \). For any \( b' \) and \( b'_1 \) such that \( b' \in \text{cl}_{n+1}(S_1, \Sigma) \) and \( f_2(b') = (b'_1, \_ \_ \_) \), we know there exist \( b, n \) and \( n' \) such that

\[ b \in \text{cl}_n(S_1, \Sigma), \quad \Sigma(b)(n) = (b', n'). \]

Since \( \text{cl}_n(S_1, \Sigma) \subseteq \text{dom}(f_2) \), we know there exist \( b_1 \) and \( n_1 \) such that

\[ f_2(b(n)) = (b_1, n_1). \]

By the induction hypothesis, we know \( f_2 \{ \text{cl}_n(S_1, \Sigma) \} \subseteq \text{cl}_n(S_1, \sigma) \). Thus

\[ b_1 \in \text{cl}_n(S_1, \sigma). \]

From \( \text{Inv}(f_2, \Sigma, \sigma) \), we know

\[ f_2(b(n')) = \sigma(b_1)(n_1). \]

Thus \( b'_1 \in \text{cl}_{n+1}(S_1, \sigma) \). Thus \( f_2 \{ \text{cl}_{n+1}(S_1, \Sigma) \} \subseteq \text{cl}_{n+1}(S_1, \sigma) \).

Thus we are done. \( \square \)

49
Lemma 62. If $f_1 \subseteq f_2$, $\text{dom}(f_1) = S_1$, $S_1 = \text{cl}(S_1, \Sigma)$, and $\text{Inv}(f_2, \Sigma, \sigma)$, then $\text{Inv}(f_1, \Sigma, \sigma)$.

Proof: For any $b, n, b'$ and $n'$ such that $(b, n) \in \text{locs}(\Sigma)$ and $f_1((b, n)) = (b', n')$, since $f_1 \subseteq f_2$, we know $f_2((b, n)) = (b', n')$. Since $\text{Inv}(f_2, \Sigma, \sigma)$, we know 

$$(b', n') \in \text{locs}(\sigma), \quad \Sigma(b)(n) \xrightarrow{f_2} \sigma(b')(n').$$

Suppose $\Sigma(b)(n) = v_1$ and $\sigma(b')(n') = v_2$. Then

$$(v_1 \notin \Pi) \wedge (v_1 = v_2) \lor v_1 \in \Pi \wedge (v_2 = v) = v_2.$$
Towards Certified Separate Compilation for Concurrent Programs

Since \( \Sigma_G \xrightarrow{\text{H-sofs}_s} \Sigma \) and \( \text{blocks}(\Delta_0, \text{ws}) \subseteq \mathbb{P} \cup \mathbb{S}_E \), we know
\[
b_m \in \mathbb{P} \cup \mathbb{S}_E.
\]
Also we know \( b_m \in \text{cl}(\mathbb{S}_G, \Sigma_G) = \mathbb{S}_G \). Since \( (\mathbb{S}_G - \mathbb{S}_E) \cap \mathbb{P} = \emptyset \), we know
\[
b_m \in \mathbb{S}_E.
\]
Thus \( b \in \text{cl}(\mathbb{S}_E, \Sigma) = \mathbb{S}_F \). Thus we are done. \( \square \)

### E.4 Proof of Flip of Simulation (Lemma 50)

To prove Lemma 50, we first define a relation \( \simeq \) between source and target programs. We also define a relation \( \simeq^i \) which is parameterized with an index \( i \). This index describes the number of steps at the target that corresponds to zero steps of the source.

**Definition 65 (Whole-Program Existential Relational).**

Let \( \tilde{\mathbb{P}} = \{ \text{let in } f_1 | \ldots | f_m \}, \tilde{\mathbb{P}} = \{ \text{let in } f_1 | \ldots | f_m, \Gamma = \{(s_1, g_{e_1}, \gamma_1), \ldots, (s_m, g_{e_m}, \gamma_m)\} \} \) and \( \Pi = \{(t_1, g_{e_1'}, \pi_1), \ldots, (t_m, g_{e_m'}, \pi_m)\} \). \( \tilde{\mathbb{P}} \leadsto_{\phi} \tilde{\mathbb{P}} \) iff \( \forall i \in \{1, \ldots, m\}. \phi(g_{e_i}) = g_{e_i'} \), and for any \( \Sigma, \sigma, \mu, \tilde{\mathbb{W}}, \tilde{\mathbb{W}} \), if \( \text{initM}(\phi, \cup g_{e_1}, \Sigma, \sigma), \mu = (\text{dom}(\Sigma), \text{dom}(\sigma), \phi), (\tilde{\mathbb{P}}, \sigma) \leadsto_{\text{load}} \tilde{\mathbb{W}}, \) and \( (\tilde{\mathbb{P}}, \Sigma) \leadsto_{\text{load}} \tilde{\mathbb{W}} \), then \( \tilde{\mathbb{W}} \simeq_{\phi}^{(\text{emp}, \text{emp})} \tilde{\mathbb{W}} \).

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

1. \( \tilde{\mathbb{W}}, t = \tilde{\mathbb{W}}, t, \) and \( \tilde{\mathbb{W}}, d = \tilde{\mathbb{W}}, d, \) and \( \text{wf}(\mu, \tilde{\mathbb{W}}, \Sigma, \tilde{\mathbb{W}}, \sigma) \);
2. one of the following holds:
   a. there exist \( \tilde{\mathbb{W}}', \Delta, \tilde{\mathbb{W}}' \) and \( \delta \) such that
      i. \( \tilde{\mathbb{W}} \leadsto_{\Delta}^{\delta} \tilde{\mathbb{W}}' \), and \( \tilde{\mathbb{W}} \leadsto_{\delta}^{\tilde{\mathbb{W}}'} \),
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \tilde{\mathbb{W}}, \Sigma), (\delta_0 \cup \delta, ((\tilde{\mathbb{W}}.T)(\tilde{\mathbb{W}}.t)), F)) \),
      iii. \( \tilde{\mathbb{W}}' \simeq_{\mu}^{(\Delta_0 \cup \Delta, \delta)} \tilde{\mathbb{W}} \);
   b. or, there exist \( \tilde{\mathbb{W}}', \Delta, \tilde{\mathbb{W}}', \delta, \mu', \tau', \tilde{\mathbb{W}}, \tilde{\mathbb{W}}', \Sigma' \) and \( \sigma' \) such that
      i. \( \tilde{\mathbb{W}} \leadsto_{\Delta}^{\delta} \tilde{\mathbb{W}}' \), and \( \tilde{\mathbb{W}} \leadsto_{\delta}^{\tilde{\mathbb{W}}'} \);
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \tilde{\mathbb{W}}, \Sigma), (\delta_0 \cup \delta, ((\tilde{\mathbb{W}}.T)(\tilde{\mathbb{W}}.t)), F)) \),
      iii. \( \tilde{\mathbb{W}}' \simeq_{\mu}^{(\Delta_0 \cup \Delta, \delta)} \tilde{\mathbb{W}} \);
   c. or, there exist \( \tilde{\mathbb{W}}', \Delta, \tilde{\mathbb{W}}' \) and \( \delta \) such that
      i. \( \tilde{\mathbb{W}} \leadsto_{\Delta}^{\delta} \tilde{\mathbb{W}}' \), and \( \tilde{\mathbb{W}} \leadsto_{\delta}^{\tilde{\mathbb{W}}'} \),
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \tilde{\mathbb{W}}, \Sigma), (\delta_0 \cup \delta, ((\tilde{\mathbb{W}}.T)(\tilde{\mathbb{W}}.t)), F)) \).

**With index.** Also we define \( \tilde{\mathbb{W}} \simeq_{\mu}^{(\Delta_0, \delta_0)} \tilde{\mathbb{W}} \) as the largest relation such that whenever \( \tilde{\mathbb{W}} \simeq_{\mu}^{(\Delta_0, \delta_0)} \tilde{\mathbb{W}} \), then the following are true:

1. \( \tilde{\mathbb{W}}, t = \tilde{\mathbb{W}}, t, \) and \( \tilde{\mathbb{W}}, d = \tilde{\mathbb{W}}, d, \) and \( \text{wf}(\mu, \tilde{\mathbb{W}}, \Sigma, \tilde{\mathbb{W}}, \sigma) \);
2. one of the following holds:
   a. there exist \( j, \tilde{\mathbb{W}}', \Delta, \tilde{\mathbb{W}}' \) and \( \delta \) such that
      i. \( \tilde{\mathbb{W}} \leadsto_{\Delta}^{\delta} \tilde{\mathbb{W}}' \), and \( \tilde{\mathbb{W}} \leadsto_{\delta}^{\tilde{\mathbb{W}}'} \),
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \tilde{\mathbb{W}}, \Sigma), (\delta_0 \cup \delta, ((\tilde{\mathbb{W}}.T)(\tilde{\mathbb{W}}.t)), F)) \),
      iii. \( \tilde{\mathbb{W}}' \simeq_{\mu}^{(\Delta_0 \cup \Delta, \delta)} \tilde{\mathbb{W}} \);
   b. or, there exist \( j, \tilde{\mathbb{W}}', \Delta, \tilde{\mathbb{W}}', \delta, \mu', \Sigma' \) and \( \sigma' \) such that
      i. \( \tilde{\mathbb{W}} \leadsto_{\Delta}^{\delta} \tilde{\mathbb{W}}' \), and \( \tilde{\mathbb{W}} \leadsto_{\delta}^{\tilde{\mathbb{W}}'} \),
      ii. \( \text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \tilde{\mathbb{W}}, \Sigma), (\delta_0 \cup \delta, ((\tilde{\mathbb{W}}.T)(\tilde{\mathbb{W}}.t)), F)) \),
      iii. \( \text{for any } \tau' \in \text{dom}(\tau') \), we have
      A. \( \tilde{\mathbb{W}}' \vdash_{\text{emp}}^{\sigma} (\tau', \tilde{\mathbb{W}}, \Sigma', \tilde{\mathbb{W}}', \gamma') \), and \( (\tilde{\mathbb{W}}', \Sigma') \vdash_{\text{emp}}^{\sigma} (\tau', \tilde{\mathbb{W}}, \Sigma', \gamma') \), and \( \sigma \neq \tau \), and
B. \((T', t', \delta', \Sigma') \approx (T, t, \delta, \Sigma')\).

c. or, there exist \(\hat{W}, \Delta, \hat{W}'\) and \(\delta\) such that

i. \(\hat{W} \overset{\tau}{\xrightarrow{\Delta}} \hat{W}', \) and \(\hat{W}' : \overset{\text{done}}{\xrightarrow{\text{emp}}}, \) and \(\hat{W} \overset{\tau}{\xrightarrow{\alpha}} \hat{W}', \) and \(\hat{W}' : \overset{\text{done}}{\xrightarrow{\text{emp}}}, \) and

ii. \(\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0 \cup \delta, ((\hat{W}.T)(\hat{W}.t)).F))\).

Lemma 66. If \(\tilde{W}, \Delta_0 \approx (\hat{W}, \delta_0)\) and Safe(\(\tilde{W}\)), then \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta_0, \delta_0}} \hat{W}\).

Lemma 67. If \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta, \delta}} \hat{W}\), then there exists \(i\) such that \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta, \delta}} \hat{W}\).

Lemma 68. If \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta, \delta}} \hat{W}\) and \(\forall t. \text{det}((\hat{W}.T)(t).t), \) then \((\hat{W}, \delta_0) \leq_i (\tilde{W}, \delta_0)\).

Then, for Lemma 24, from \(\text{let } \Gamma \text{ in } f_1 | \ldots | f_m \approx_{\text{gc}, \phi} \text{let } \Pi \text{ in } f_1 | \ldots | f_m, \) we know

\[ \forall i \in \{1, \ldots, m\}. \text{dom}(sl_i, \text{InitCore}(\gamma_i)) = \text{dom}(tl_i, \text{InitCore}(\pi_i)). \]

For any \(T, t, \delta, \Sigma, \mu, S, \) and \(S\), if \((\hat{P}, \Sigma) : \overset{\text{load}}{\xrightarrow{\text{load}}} (T, t, \delta, \Sigma), \) \((\hat{P}, \sigma) : \overset{\text{load}}{\xrightarrow{\text{load}}} (T, t, \delta, \Sigma), \) \(\text{gc} \subseteq \text{locs}(\Sigma), \) \(\text{dom}(\Sigma) = S \subseteq \text{dom}(\phi), \)
\(\phi(S, \Sigma) = \phi(S) \subseteq \phi(S) \subseteq \phi(S), \) \(\text{fpmatch}(\mu, (\hat{W}, \Sigma), ((\hat{W}.T)(\hat{W}.t)).F))\), and
\(\text{let } \Gamma \text{ in } f_1 | \ldots | f_m \approx_{\text{gc}, \phi} \text{let } \Pi \text{ in } f_1 | \ldots | f_m, \) we know there exists \(i \in \text{index}\) such that

\[(\hat{T}, t, \delta, \Sigma, \mu, S) \leq \mu((\hat{T}, t, \delta, \Sigma, \mu, S)). \]

By Lemmas 66, 67, and 68, we know there exists \(j\) such that \(((\hat{T}, t, \delta, \Sigma, \mu, S)) \leq \mu(((\hat{T}, t, \delta, \Sigma, \mu, S))). \) Thus we are done.

Proof of Lemma 66. We prove the following strengthened result:

If \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta_0}} \hat{W}\) and \((\tilde{W}_0, \Delta_0 \cup \Delta_0') \approx (\hat{W}_0, \delta_0 \cup \delta_0')\), then \(\tilde{W} \overset{\tau}{\xrightarrow{\Delta_0, \delta_0}} \hat{W}\).

By co-induction. Then, by induction over \(i\) (using the well-founded ordering of \(\text{index}\)).

1. \(i = 0.\)

From Safe(\(\tilde{W}\)), we know \(\neg(\tilde{W}_0 : \overset{\text{emp}}{\xrightarrow{\text{emp}}} \text{abort})\), we know there exists \(\tilde{W}'\) and \(\Lambda'\) such that

\[ \tilde{W}_0 \overset{\lambda}{\xrightarrow{\Lambda'}} \tilde{W}', \text{ or } \exists \lambda \neq \tau. \tilde{W}_0 \overset{\lambda}{\xrightarrow{\text{emp}}} \tilde{W}', \text{ or } \tilde{W}_0 : \overset{\text{emp}}{\xrightarrow{\text{emp}}} \text{ done}. \]

a. If \(\tilde{W}_0 : \overset{\lambda}{\xrightarrow{\Lambda'}} \tilde{W}',\) from \((\tilde{W}_0, \Delta_0 \cup \Delta_0') \approx (\hat{W}_0, \delta_0 \cup \delta_0'),\) we know there exist \(\tilde{W}', \delta', j\) such that

i. \(\tilde{W}_0 : \overset{\lambda}{\xrightarrow{\delta'}} \text{ and } \tilde{W} : \overset{\delta'}{\xrightarrow{\delta'}} \text{ do } \).

ii. \(\text{FPmatch}(\Delta_0 \cup \Delta_0', \delta_0 \cup \delta_0', \delta, \mu, (\tilde{W}, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F))\) and

iii. \((\tilde{W}', \delta_0 \cup \delta_0', \delta) \approx (\hat{W}', \delta_0 \cup \delta_0', \delta')\).

Thus we know

\[ \tilde{W} : \overset{\lambda}{\xrightarrow{\Delta_0 \cup \Delta_0'}} \tilde{W}', \text{ and } \hat{W} : \overset{\delta}{\xrightarrow{\delta'}} \text{ do } \hat{W}'. \]

From FPmatch(\(\Delta_0 \cup \Delta_0' \cup \Delta', \delta_0 \cup \delta_0' \cup \delta', \mu, (\tilde{W}_0, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F))\), by Lemma 57, we know

\[ \text{FPmatch}(\Delta_0 \cup \Delta_0' \cup \Delta', \delta_0 \cup \delta_0' \cup \delta', \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F)) \] .

Since \(\hat{W}' : \overset{\tau}{\xrightarrow{\text{emp}}} \hat{W}', \tilde{W} : \overset{\tau}{\xrightarrow{\text{emp}}}, \) and \((\tilde{W}', \Delta_0 \cup \Delta_0' \cup \delta) \approx (\hat{W}', \delta_0 \cup \delta_0' \cup \delta'),\) by the co-induction hypothesis, we know

\[ \tilde{W} : \overset{\lambda}{\xrightarrow{\Delta_0 \cup \Delta_0' \cup \Delta', \delta_0 \cup \delta_0' \cup \delta'}} \tilde{W}'. \]

Thus we are done.

b. If \(\lambda \neq \tau\) and \(\tilde{W}_0 : \overset{\lambda}{\xrightarrow{\text{emp}}} \tilde{W}'\), by the operational semantics, we know there exist \(T', \delta'\) and \(\Sigma'\) such that for any \(t' \in \text{dom}(T')\), we have

\[ \tilde{W}_0 : \overset{\lambda}{\xrightarrow{\text{emp}}} (T', t', \delta', \Sigma'). \]

From \((\tilde{W}_0, \Delta_0 \cup \Delta_0') \approx (\hat{W}_0, \delta_0 \cup \delta_0'),\) we know there exist \(\tilde{W}', \delta', \mu', T', \sigma'\) and \(j\) such that for any \(t' \in \text{dom}(T')\), we have

i. \(\tilde{W}_0 : \overset{\lambda}{\xrightarrow{\delta'}} \tilde{W}', \) and \(\tilde{W}' \overset{\lambda}{\xrightarrow{\text{emp}}} (T', t', \delta', \sigma')\), and

ii. \(\text{FPmatch}(\Delta_0 \cup \Delta_0', \delta_0 \cup \delta_0' \cup \delta', \mu, (\tilde{W}_0, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F))\) and

iii. \((T', t', \delta', \Sigma', \text{emp}) \approx (T', t', \delta', \sigma', \text{emp})\).
Towards Certified Separate Compilation for Concurrent Programs

Since \(((T', t', d', \Sigma'), \text{emp}) \preceq_{\mu}^i ((T', t', d', \Sigma'), \text{emp})\), by the co-induction hypothesis, we know

\[(T', t', d', \Sigma') \triangleright_{\mu,(\text{emp},\text{emp})} (T', t', d', \Sigma') .\]

Thus we are done.

c. If \(\hat{W}_0 := \hat{W}\) done, from \((\hat{W}_0, \Delta_0 \cup \Delta'_0) \preceq_{\mu}^i (\hat{W}_0, \delta_0 \cup \delta'_0)\), we know there exist \(\hat{W}'\) and \(\delta'\) such that

i. \(\hat{W}_0 \overset{\tau}{\Rightarrow} \hat{W}'\), and \(\hat{W}' \overset{\delta'}{\Rightarrow} \text{done}\), and

ii. \(\text{FPmatch}(\Delta_0 \cup \Delta'_0, \delta_0 \cup \delta'_0 \cup \delta' \cup \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F))\).

Thus we are done.

2. \(i > 0\).

From \(\text{Safe}((\hat{W}))\), we know \(!((\hat{W}_0) \Rightarrow_{\text{emp}} \text{abort})\), we know there exists \(\hat{W}'\) and \(\Delta'\) such that

\(\hat{W}_0 := \hat{W}'\) or \(\exists o \neq \tau. \hat{W}_0 := o(\text{emp}) \hat{W}'\), or \(\hat{W}_0 := \tau(\text{emp}) \hat{W}'\).

a. If \(\hat{W}_0 := \hat{W}'\), from \((\hat{W}_0, \Delta_0 \cup \Delta'_0) \preceq_{\mu}^i (\hat{W}_0, \delta_0 \cup \delta'_0)\), we know one of the following holds:

i. there exists \(j\) such that

A. \(j < i\), and

B. \((\hat{W}', \Delta_0 \cup \Delta'_0 \cup \Delta') \preceq_{\mu}^j (\hat{W}_0, \delta_0 \cup \delta'_0)\);

ii. or, there exist \(\hat{W}'', \delta'\) and \(j\) such that

A. \(\hat{W}_0 \overset{r}{\Rightarrow} \hat{W}''\), and

B. \(\text{FPmatch}(\Delta_0 \cup \Delta'_0 \cup \Delta' \cup \delta_0 \cup \delta'_0 \cup \delta' \cup \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F))\), and

C. \((\hat{W}', \Delta_0 \cup \Delta'_0 \cup \Delta') \preceq_{\mu}^i (\hat{W}'', \delta_0 \cup \delta'_0 \cup \delta' \cup \mu)\).

For case (i), we know

\(\hat{W} := \hat{W}'\), and \(\hat{W} := \hat{W}'\) done.

Since \(j < i\), by the induction hypothesis, we know

\(\hat{W} \triangleright_{(\Delta_0 \cup \Delta'_0)} \hat{W}''\).

For case (ii), the proof is similar to the above 1(a).

b. If \(\exists o \neq \tau. \hat{W}_0 := o(\text{emp}) \hat{W}'\), the proof is similar to the above 1(b).

c. If \(\hat{W}_0 := \tau(\text{emp}) \hat{W}'\), the proof is similar to the above 1(c).

Thus we are done.

Proof of Lemma 67. We prove \(\hat{W} \triangleright_{\mu,(\text{emp},\text{emp})} \hat{W}\) holds if \(\hat{W}.t = \hat{W}.t\) and \(\hat{W}.d = \hat{W}.d\) and \(\text{wf}(\mu, \hat{W}, \Sigma)\) and one of the following holds:

1. there exist \(\hat{W}', \Delta, \hat{W}'\) and \(\delta\) such that

a. \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and

b. \(\text{FPmatch}(\Delta_0 \cup \Delta_0 \cup \delta_0 \cup \delta'_0 \cup \delta' \cup \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F))\), and

c. \(\hat{W}' \triangleright_{(\Delta_0 \cup \Delta'_0 \cup \delta)} \hat{W}'\).

2. or, there exist \(\hat{W}', \Delta, \hat{W}', \delta, \mu', T', d', o, \Sigma'\) and \(\sigma'\) such that

a. \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and

b. \(\text{FPmatch}(\Delta_0 \cup \Delta_0 \cup \delta_0 \cup \delta'_0 \cup \delta' \cup \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F))\), and

c. for any \(t' \in \text{dom}(\hat{W}')\), we have

i. \(\hat{W}' \overset{o(\text{emp})}{\Rightarrow} (T', t', d', \Sigma')\), and \(\hat{W}' \overset{o(\text{emp})}{\Rightarrow} (T', t', d', \sigma')\), and \(o \neq \tau\), and

ii. \((T', t', d', \Sigma') \overset{(\text{emp}, \text{emp})}{\Rightarrow} (T', t', d', \sigma')\);

3. or, there exist \(\hat{W}', \Delta, \hat{W}'\) and \(\delta\) such that

a. \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\), and \(\hat{W} \overset{r}{\Rightarrow} \hat{W}'\)

b. \(\text{FPmatch}(\Delta_0 \cup \Delta_0 \cup \delta_0 \cup \delta'_0 \cup \delta' \cup \mu, (\hat{W}_0, \Sigma, ((\hat{W}.T)(\hat{W}.t)).F))\).

We prove the above by co-induction. Thus we are done.
Proof of Lemma 68. By co-induction. We need to prove the following:

1. \( \forall \tilde{W}', \delta. \text{if } \tilde{W} \xrightarrow{\tilde{\delta}} \tilde{W}' \text{, then one of the following holds:} \)
   a. there exists \( j \) such that
      i. \( j < i \), and
      ii. \( (\tilde{W}', \delta_0 \cup \delta) \leq f^j (\tilde{W}, \Delta_0) \);  
   b. there exist \( \tilde{W}', \Delta \) and \( j \) such that
      i. \( \tilde{W} \xrightarrow{\delta} \tilde{W}' \), and
      ii. \( \text{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta \cup \Delta, \mu, (\tilde{W}, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F)) \), and
      iii. \( (\tilde{W}', \delta_0 \cup \delta) \leq f^j (\tilde{W}', \Delta_0 \cup \Delta) \).

Proof. By inversion of \( \tilde{W} \xrightarrow{\mu} (\tilde{W}_1, \Delta_0 \cup \Delta) \), we know one of the following holds:

(A) there exist \( \tilde{W}_1, \Delta, \tilde{W}_2, \delta_1, \delta_2 \) and \( i' \) such that
   (1) \( \tilde{W} \xrightarrow{\delta} \tilde{W}_1 \), and \( \tilde{W} \xrightarrow{\delta} \tilde{W}_2 \), and \( \tilde{W}_1 \xrightarrow{\delta_1} \tilde{W}_2 \).
   (2) \( \text{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta_1 \cup \delta_2, \mu, (\tilde{W}, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F)) \).
   (3) \( \tilde{W}_1 \xrightarrow{\mu} (\tilde{W}_2, \delta_0 \cup \delta_1 \cup \delta_2) \).

First, from (3), by the co-induction hypothesis, we know
\[
(\tilde{W}_2, \delta_0 \cup \delta_1 \cup \delta_2) \leq f^j (\tilde{W}_1, \Delta_0 \cup \Delta) \tag{E.3}
\]
Then we prove:
\[
\forall \tilde{W}_0, n_0, n_1, \delta', \delta''. \ (\tilde{W}_1 \xrightarrow{\delta_{n_0}} \tilde{W}_0) \wedge (\tilde{W}_0 \xrightarrow{\delta_{n_1}} \tilde{W}_2) \implies (\tilde{W}_0, \delta_0 \cup \delta_1 \cup \delta') \leq f^{n_0 + n_1} (\tilde{W}_1, \Delta_0 \cup \Delta) \tag{E.4}
\]
By induction over \( n_1 \).
   i. \( n_1 = 0 \). Thus \( \tilde{W}_0 = \tilde{W}_2 \). Since \( \forall t. \text{det}((\tilde{W}.T)(\tilde{W}.t)) \), we know \( \delta' = \delta_2 \). By (E.3), we are done.
   ii. \( n_1 = m + 1 \). Thus there exist \( \tilde{W}_0', \delta_0' \) and \( \delta_1' \) such that \( \delta'' = \delta_0' \cup \delta_1', \tilde{W}_0 \xrightarrow{\delta} \tilde{W}'_0 \) and \( \tilde{W}'_0 \xrightarrow{\delta_1} \tilde{W}_2 \). By the induction hypothesis, we know
\[
(\tilde{W}_0', \delta_0 \cup \delta_1 \cup \delta_1') \leq f^{m+1} (\tilde{W}_1, \Delta_0 \cup \Delta) \ .
\]
To prove \( (\tilde{W}_0, \delta_0 \cup \delta_1 \cup \delta') \leq f^{m+1} (\tilde{W}_1, \Delta_0 \cup \Delta) \), by co-induction. Since \( \forall t. \text{det}((\tilde{W}.T)(\tilde{W}.t)) \), we can finish the proof. Thus we are done of (E.4). Thus, we know (E.5) holds.
\[
(\tilde{W}_1, \delta_0 \cup \delta_1) \leq f^{m+1} (\tilde{W}_1, \Delta_0 \cup \Delta) \tag{E.5}
\]
Finally, since \( \forall t. \text{det}((\tilde{W}.T)(\tilde{W}.t)) \), we know \( \tilde{W}_1 = \tilde{W}' \) and \( \delta_1 = \delta \). From (2), we know \( \text{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta_1, \mu, (\tilde{W}, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F)) \).

From (1) and (E.5), we are done.

(B) there exist \( i', \tilde{W}_1, \Delta, \tilde{W}_2, \delta_1, \delta', \delta'', \tilde{T}', \tilde{t}', \tilde{d}', \sigma' \) and \( \sigma'' \) such that
   (1) \( \tilde{W} \xrightarrow{\tilde{\delta}} \tilde{W}_1 \), and \( \tilde{W} \xrightarrow{\tilde{\delta}} \tilde{W}_2 \), and \( \tilde{W}_1 \xrightarrow{\delta_1} \tilde{W}_2 \).
   (2) \( \text{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta_1, \mu, (\tilde{W}, \Sigma, ((\tilde{W}.T)(\tilde{W}.t)).F)) \).
   (3) for any \( \tilde{t}' \in \text{dom}(\tilde{t}') \), we have
      A. \( \tilde{W}_1 \xrightarrow{o} (\tilde{T}', \tilde{t}', \tilde{d}', \Sigma') \), and \( \tilde{W}_1 \xrightarrow{o} (\tilde{T}', \tilde{t}', \tilde{d}', \sigma') \), and \( o \neq \tau \), and
      B. \( (\tilde{T}', \tilde{t}', \tilde{d}', \Sigma') \xrightarrow{r} (\tilde{T}', \tilde{t}', \tilde{d}', \sigma') \).
Towards Certified Separate Compilation for Concurrent Programs

First, from (3), by the co-induction hypothesis, we know

\[(T', t', d', \sigma') \preceq_\mu (T', t', d', \Sigma'), \text{ emp}.\]  (E.6)

Since \(\forall t. \det((\widehat{W}.T)(t), tl)\), we know

\[(\widehat{W}_1, \delta_0 \cup \delta_1) \preceq_\mu (\widehat{W}_1, \Delta_0 \cup \Delta).\]  (E.7)

Also \(i > 0\) and there exists \(\delta'_i\) such that \(\delta_1 = \delta \cup \delta'_i\),

\[\widehat{W} \xrightarrow{\delta} \delta' \quad \text{and} \quad \widehat{W} \xrightarrow{\delta'_i} \widehat{W}_1.\]

Similarly to the above case (A), we can prove (E.4). Then we know

\[(\widehat{W}', \delta_0 \cup \delta) \preceq_{\mu-1} (\widehat{W}_1, \Delta_0 \cup \Delta).\]  (E.8)

Since \(\forall t. \det((\widehat{W}.T)(t), tl)\), from (1) and (2), we are done.

(C) there exist \(\widehat{W}_1, \Delta, \widehat{W}_1\) and \(\delta_1\) such that

(1) \(\widehat{W} \xrightarrow{\Delta} \widehat{W}_1\), and \(\widehat{W} \xrightarrow{\delta_1} \text{ done}\), and \(\widehat{W} \xrightarrow{\exp} \text{ done}\), and

(2) \(\text{FPmatch}(\Delta_0 \cup \Delta, \delta_0 \cup \delta_1, \mu, (\widehat{W}.\Sigma, ((\widehat{W}.T)(\widehat{W}.t)), F))\).

Since \(\forall t. \det((\widehat{W}.T)(t), tl)\), we know

\[(\widehat{W}_1, \delta_0 \cup \delta_1) \preceq_\mu (\widehat{W}_1, \Delta_0 \cup \Delta).\]  (E.9)

Also \(i > 0\) and there exists \(\delta'_i\) such that \(\delta_1 = \delta \cup \delta'_i\),

\[\widehat{W} \xrightarrow{\delta} \delta' \quad \text{and} \quad \widehat{W} \xrightarrow{\delta'_i} \widehat{W}_1.\]

Similarly to the above case, we can prove (E.4). Then we know

\[(\widehat{W}', \delta_0 \cup \delta) \preceq_{\mu-1} (\widehat{W}_1, \Delta_0 \cup \Delta)\]  (E.10)

Thus we are done.

2. \(\forall T', d', \sigma'. o.\) if \(o \neq \tau\) and \(\exists t'. \widehat{W} \xrightarrow{\sigma} (T', t', \delta', \sigma')\), then there exist \(\widehat{W}', \Delta, \mu, T', \Sigma, j\) such that for any \(t' \in \text{ dom}(T')\), we have
   a. \(\widehat{W} \xrightarrow{\Delta} \widehat{W}'\), and \(\widehat{W}' \xrightarrow{\exp} (T', t', d', \Sigma')\), and
   b. \(\text{FPmatch}(\Delta_0 \cup \Delta, \delta_0, \mu, (\widehat{W}.\Sigma, ((\widehat{W}.T)(\widehat{W}.t)), F))\), and
   c. \((T', t', d', \Sigma'), \text{ emp} \preceq_{\mu} (T', t', d', \Sigma'), \text{ emp})\).

Proof. By inversion of \(\widehat{W} \xrightarrow{\sigma} (T', t', \delta', \Sigma')\), since \(\forall t. \det((\widehat{W}.T)(t), tl)\), we know there exist \(t', \widehat{W}_1, \Delta, \mu, T', \Sigma, d', o\), and \(j\) such that

(1) \(\widehat{W} \xrightarrow{\sigma} \widehat{W}_1\), and

(2) \(\text{FPmatch}(\Delta_0 \cup \Delta, \delta_0, \mu, (\widehat{W}.\Sigma, ((\widehat{W}.T)(\widehat{W}.t)), F))\), and

(3) for any \(t' \in \text{ dom}(T')\), we have
   i. \(\widehat{W}_1 \xrightarrow{\exp} (T', t', d', \Sigma')\), and \(\widehat{W} \xrightarrow{\exp} (T', t', d', \Sigma'), \text{ and } o \neq \tau\), and
   ii. \((T', t', d', \Sigma') \preceq_{\mu} (T', t', d', \Sigma')\).

From (3), by the co-induction hypothesis, we know

\[(T', t', d', \Sigma') \preceq_{\mu} (T', t', d', \Sigma'), \text{ emp})\).

Thus we are done.

3. if \(\widehat{W} \xrightarrow{\exp} \text{ done}\), then there exist \(\widehat{W}'\) and \(\Delta\) such that
   a. \(\widehat{W} \xrightarrow{\Delta} \widehat{W}'\), and \(\widehat{W}' \xrightarrow{\exp} \text{ done}\), and
   b. \(\text{FPmatch}(\Delta_0 \cup \Delta, \delta_0, \mu, (\widehat{W}.\Sigma, ((\widehat{W}.T)(\widehat{W}.t)), F))\).

Proof. By inversion of \(\widehat{W} \xrightarrow{\sigma} (T', t', \delta', \Sigma')\), since \(\forall t. \det((\widehat{W}.T)(t), tl)\), we know there exist \(\widehat{W}_1\) and \(\Delta\) such that

(1) \(\widehat{W} \xrightarrow{\exp} \text{ done}\), and \(\widehat{W} \xrightarrow{\exp} \text{ done}\), and
We only need to prove: for any \( \hat{F}, \hat{P}, \varphi, \Sigma \) and \( \sigma \), if \( \hat{P} \equiv_{\varphi} \hat{F}, \) initM(\( \varphi, GE(\hat{P}, \Gamma), \Sigma, \sigma \)), and \( (\hat{P}, \sigma) \implies \text{Race} \), then \( (\hat{F}, \sigma) \implies \text{Race} \).

Thus we are done.

Thus we have finished the proof.

\[ \Box \]

E.5 Proof of NPDRF Preservation (Lemma 51)

We only need to prove: for any \( \hat{F}, \hat{P}, \varphi, \Sigma \) and \( \sigma \), if \( \hat{P} \equiv_{\varphi} \hat{F}, \) initM(\( \varphi, GE(\hat{P}, \Gamma), \Sigma, \sigma \)), and \( (\hat{P}, \sigma) \implies \text{Race} \), then \( (\hat{F}, \sigma) \implies \text{Race} \).

From \( (\hat{P}, \sigma) \implies \text{Race} \), we have two cases:

1. there exist \( \hat{W}, t_1, t_2, \delta_1, \delta_2, d_1, d_2 \) such that \( (\hat{P}, \sigma) \gg \hat{W}, t_1 \neq t_2, \text{NPpred}(\hat{W}, t_1, (\delta_1, d_1)), \text{NPpred}(\hat{W}, t_2, (\delta_2, d_2)) \) and \( \neg((\delta_1, d_1) \sim (\delta_2, d_2)) \).

From \text{NPpred}(\hat{W}, t_1, (\delta_1, d_1)) \) and \text{NPpred}(\hat{W}, t_2, (\delta_2, d_2)) \), we know there exist \( T, d, \sigma, t_1, \pi_1, t_2, \pi_2, t_3, \pi_3, t_4, \pi_4 \), such that

\[
\hat{W} = (T, \_, d), \quad T(t_1) = (\pi_1, t_3, \pi_2, t_4), \quad (\pi_1, t_3, \pi_2, t_4) \xrightarrow{r}\ (\pi_1', t_3', \pi_2', t_4'), \quad d(t_1) = d_1,
\]

\[
T(t_2) = (\pi_2, t_3, \pi_2, t_4), \quad (\pi_2, t_3, \pi_2, t_4) \xrightarrow{r}\ (\pi_2', t_3', \pi_2', t_4'), \quad d(t_1) = d_2.
\]

By the operational semantics, we know

\[
(\hat{P}, \sigma) \gg \gg (T, t_1, d, \sigma), \quad (\hat{P}, \sigma) \gg \gg (T, t_2, d, \sigma), \quad d_1 = d_2 = 0.
\]

From \( \hat{P} \equiv_{\varphi} \hat{F} \), we know there exist \( f_1, \ldots, f_n, \Gamma = \{(s_1, \gamma_1), \ldots, (s_m, \gamma_m)\} \) and \( \Pi = \{(t_1, \pi_1), \ldots, (t_m, \pi_m)\} \), such that \( \hat{P} = \text{let} \Gamma \in f_1 \mid \ldots \mid f_m, \hat{F} = \text{let} \Pi \in f_1 \mid \ldots \mid f_m \) and \( \forall i \in \{1, \ldots, m\}, \text{dom}(s_i, \text{InitCore}(\gamma_i)) = \text{dom}(t_1, \text{InitCore}(\pi_1)) \).

Thus we know there exists \( T \) such that

\[
(\hat{P}, \Sigma) \gg \gg (T, t_1, d, \Sigma), \quad (\hat{F}, \Sigma) \gg \gg (T, t_2, d, \Sigma).
\]

Let \( S = \text{dom}(\Sigma) \), \( S = \varphi \notin S \) and \( \mu = (S, S, \varphi \notin S) \). We know there exists \( i_1, i_2 \in \text{index such that} \)

\[
((T, t_1, d, \sigma, \text{emp}), \text{emp}) \leq_{\mu} ((T, t_1, d, \Sigma, \text{emp}), \text{emp}) \leq_{\mu} ((T, t_2, d, \Sigma, \text{emp}) \).
\]

By Lemma 69, we know there exist \( \hat{W}_1', \Delta_1', \hat{W}_2, \Delta_2', \hat{W}_2', \hat{W}_2' \) and \( \hat{W}_1 \) such that

\[
(\hat{P}, \Sigma) \gg \gg (T, t_1, d, \sigma), \quad (\hat{F}, \Sigma) \gg \gg (T, t_2, d, \sigma), \quad \hat{W}_1 \gg \gg (\hat{W}_1', \Delta_1, \hat{W}_2, \Delta_2, \hat{W}_2'), \quad \Delta_1 \subseteq \Delta_1', \quad \Delta_2 \subseteq \Delta_2', \quad \text{FPmatch}(\mu, (\Delta_1, \Sigma), (\Delta_1', F_1));
\]

\[
(\hat{P}, \Sigma) \gg \gg (T, t_2, d, \sigma), \quad (\hat{F}, \Sigma) \gg \gg (T, t_2, d, \sigma), \quad \hat{W}_2 \gg \gg (\hat{W}_2', \Delta_2, \hat{W}_2'), \quad \Delta_2 \subseteq \Delta_2', \quad \text{FPmatch}(\mu, (\Delta_2, \Sigma), (\Delta_2', F_2));
\]

Let \( \hat{W} = (T, t_1, d, \Sigma) \). Then

\[
\text{NPpred}(\hat{W}, t_1, (\Delta_1', d_1)) \) and \text{NPpred}(\hat{W}, t_2, (\Delta_2', d_2)) \).
\]

Since \( (\Delta_1, d_1) \sim (\Delta_2, d_2) \), we know

\[
(\Delta_1' \sim \Delta_2') \quad \text{and} \quad (\Delta_1' \sim \Delta_2').
\]

Since \( (\hat{P}, \sigma) \gg \gg \hat{W} \), we know

\[
F_1 \cap F_2 = \emptyset.
\]

Since \( \text{wf}(\mu, \Sigma, \sigma) \), by Lemma 70, we know

\[
(\hat{P}, \sigma) \gg \gg (T, t_2, d, \sigma), \quad (\hat{F}, \sigma) \gg \gg (T, t_2, d, \sigma), \quad \hat{W}_2 \gg \gg (\hat{W}_2', \Delta_2, \hat{W}_2'), \quad \Delta_2 \subseteq \Delta_2', \quad \text{FPmatch}(\mu, (\Delta_2, \Sigma), (\Delta_2', F_2)).
\]

Since \( (\hat{P}, \sigma) \gg \gg \hat{W} \), we know

\[
F_1 \cap F_2 = \emptyset.
\]

Thus \( (\hat{P}, \sigma) \gg \gg (T, t_2, d, \sigma), \) Thus we have

\[
(\hat{P}, \sigma) \implies \text{Race}.
\]

2. there exist \( \hat{W} \) and \( n \) such that \( (\hat{P}, \sigma) \gg \gg \hat{W} \) and \( \hat{W} \gg \gg \sigma \) Race.

Suppose \( \hat{W} = (T, t, d, \sigma) \). From \( \hat{P} \equiv_{\varphi} \hat{F} \), we know there exist \( f_1, \ldots, f_n, \Gamma = \{(s_1, \gamma_1), \ldots, (s_m, \gamma_m)\} \) and \( \Pi = \{(t_1, \pi_1), \ldots, (t_m, \pi_m)\} \), such that \( \hat{P} = \text{let} \Gamma \in f_1 \mid \ldots \mid f_m, \hat{F} = \text{let} \Pi \in f_1 \mid \ldots \mid f_m \) and \( \forall i \in \{1, \ldots, m\}, \text{dom}(s_i, \text{InitCore}(\gamma_i)) = \text{dom}(t_i, \text{InitCore}(\pi_i)) \). Thus we know there exists \( T \) such that

\[
(\hat{P}, \Sigma) \gg \gg (T, t, d, \Sigma).
\]
Towards Certified Separate Compilation for Concurrent Programs

Let $S = \text{dom}(\Sigma)$, $S = \text{dom}(\sigma)$, $\mu = (S, S, \varphi[S])$ and $\Delta_0 = \delta_0 = \text{emp}$. We know there exists $i \in \text{index}$ such that

$$\left( (T, t, d, \sigma), \delta_0 \right) \leq^i_j \left( (T, t, d, \Sigma), \Delta_0 \right).$$

We only need to prove $((T, t, d, \Sigma) \implies^* \text{Race})$. By induction over $n$.

a. $n = 1$.

From $(T, t, d, \sigma) \implies \text{Race}$, we know there exist $\hat{W}', o, t_1, t_2, \delta_1, \delta_2, d_1$ and $d_2$ such that

$$\left( (T, t, d, \sigma) \implies^\varphi_{\text{emp}} \hat{W}' \right), \quad o = e \vee o = \text{sw}, \quad t_1 \neq t_2,$$

$$\text{NPpred}(\hat{W}', t_1, (\delta_1, d_1)), \quad \text{NPpred}(\hat{W}', t_2, (\delta_2, d_2)), \quad \neg((\delta_1, d_1) \sim (\delta_2, d_2))$$

Suppose $\hat{W}' = (T', t', d', \sigma)$. From $(T, t, d, \sigma) \implies^\varphi_{\text{emp}} \hat{W}'$, $o = e \vee o = \text{sw}$ and $((T, t, d, \sigma), \delta_0) \leq^i_j ( (T, t, d, \Sigma), \Delta_0 )$, we know there exist $\Delta'', \hat{W}'', \mu', \Sigma'$ and $j$ such that

$$\left( (T, t, d, \Sigma) \implies_j \hat{W}'' \right), \quad \hat{W}'' \implies^\varphi_{\text{emp}} ( (T', t_1, d', \Sigma') \cup (T', t_2, d', \Sigma') \cup (T', t_1, d', \Sigma'), \text{emp} ) \leq^j_{\mu'} \left( (T', t_2, d', \Sigma'), \text{emp} \right).$$

From $\text{NPpred}(\hat{W}', (\delta_1, d_1))$ and $\text{NPpred}(\hat{W}', (\delta_2, d_2))$, we know there exist $t_1, \pi_1, F_1, \kappa_1', \kappa_1'', \sigma_1', t_2, \pi_2, F_2, \kappa_2', \kappa_2'', \sigma_2'$ such that

$$\left( T^j(t_1) = (t_1t_1, \pi_1, F_1, \kappa_1'), \quad (T^j(t_1), d', \Sigma') \right) \\implies_{\delta_1}^j \left( \left( T^j(t_1), d', \Sigma' \right) \right) \quad \text{and} \quad \left( T^j(t_2) = (T^j(t_2), \pi_2, F_2, \kappa_2'), \quad (T^j(t_2), d', \Sigma') \right) \\implies_{\delta_2}^j \left( \left( T^j(t_2), d', \Sigma' \right) \right).$$

By the operational semantics, we know

$$\left( T^j(t_1), d', \Sigma' \right) \\implies_{\delta_1}^j \left( \left( T^j(t_1), d', \Sigma' \right) \right) \quad \text{and} \quad \left( T^j(t_2), d', \Sigma' \right) \\implies_{\delta_2}^j \left( \left( T^j(t_2), d', \Sigma' \right) \right).$$

By Lemma 69, we know there exist $\hat{W}'$, $\delta_1$, $\hat{W}'_1$, $\Delta_1$, $\hat{W}_2$, $\delta_2$ and $\Delta_2$ such that

$$\left( T^j(t_1), d', \Sigma' \right) \\implies_{\delta_1}^j \left( \left( T^j(t_1), d', \Sigma' \right) \right) \quad \text{and} \quad \left( T^j(t_2), d', \Sigma' \right) \\implies_{\delta_2}^j \left( \left( T^j(t_2), d', \Sigma' \right) \right).$$

Let $\hat{W}' = (T', t_1, d', \Sigma')$. Then

$$\text{NPpred}(\hat{W}', (\Delta_1, d_1)) \quad \text{and} \quad \text{NPpred}(\hat{W}', (\Delta_2, d_2)) \implies \text{Race}.$$

Since $\neg((\delta_1, d_1) \sim (\delta_2, d_2))$, we know $\neg((\delta_1, d_1) \sim (\delta_2, d_2))$.

Since $\delta_1 \equiv \delta_1'$ and $\delta_2 \equiv \delta_2'$, we know $\neg((\delta_1' \sim (\delta_2'))$.

Since $F_1 \cap F_2 = \emptyset$ and $\text{wp}(\mu', \Sigma', \sigma)$, by Lemma 70, we know

$$\neg((\Delta_1 \equiv \Delta_2') \implies \text{Race}.$$

Thus we have $((T, t, d, \Sigma) \implies^* \text{Race})$.

b. $n = m + 1$. Thus we know there exist $\hat{W}'$, $o$, $\delta$ such that

$$\left( (T, t, d, \sigma) \implies^\varphi_{\text{emp}} \hat{W}' \right) \quad \text{and} \quad \hat{W}' \implies^* \text{Race}.$$

Since $((T, t, d, \sigma), \delta_0) \leq^i_j ( (T, t, d, \Sigma), \Delta_0 )$, we know there exists $\Delta, \hat{W}', \mu', \sigma$ and $\delta_0$ such that

$$\left( (T, t, d, \Sigma) \implies^\varphi_{\text{emp}} \hat{W}' \right) \quad \text{and} \quad \left( \hat{W}', \delta_0 \right) \leq^i_j \left( \hat{W}', \Delta_0 \right).$$

By the induction hypothesis, we know

$$\hat{W}' \implies^* \text{Race}.$$

Thus we have

$$\left( (T, t, d, \Sigma) \implies^* \text{Race} \right).$$

Thus we are done.

Lemma 69. If $(\hat{W}, \delta_0) \leq^i_j (\hat{W}, \Delta_0)$, Safe$(\hat{W})$ and $\hat{W} \implies^* \hat{W}_1$, then there exist $\hat{W}'$, $\delta$, $\hat{W}'$ and $\Delta$ such that

1. $\hat{W} \implies^* \hat{W}'$, and $\hat{W} \implies^* \hat{W}'$, and $\delta \leq \delta$, and
2. $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0 \cup \delta, ((\hat{W}. T)(\hat{W}. t)) F)).$
Proof. By induction over $n$.

1. $n = 0$. Thus $\delta_1 = \text{emp}$. By induction over $i$ (using the well-founded order).

   a. $\exists j < i$.

      Since $\neg(\hat{W} \xrightarrow{\text{emp}} \text{abort})$, we know there exists $\hat{W}''$ and $\delta''$ such that

      $$\hat{W} : \xrightarrow{\delta''} \hat{W}''$$

      or $\exists \tau \neq \sigma, \hat{W} : \xrightarrow{\text{emp}} \hat{W}''$, or $\hat{W} : \xrightarrow{\text{emp}} \text{done}$

      i. If $\hat{W} : \xrightarrow{\sigma} \hat{W}''$, we let $\hat{W}' = \hat{W}''$ and $\delta = \delta''$. From $(\hat{W}, \delta_0) \leq_i (\hat{W}, \Delta_0)$ and $\neg \exists j < i$, we know there exist $\hat{W}'$ and $\Delta$ such that

      $$\hat{W} \xrightarrow{\Delta} \hat{W}'$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0 \cup \delta, ((\hat{W}.T)(\hat{W}.t)).F))$.

      ii. If $\exists \tau \neq \sigma, \hat{W} : \xrightarrow{\text{emp}} \hat{W}''$, we let $\hat{W}' = \hat{W}$. From $(\hat{W}, \delta_0) \leq_i (\hat{W}, \Delta_0)$, we know there exist $\hat{W}'$ and $\Delta$ such that

      $$\hat{W} \xrightarrow{\Delta} \hat{W}'$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0, ((\hat{W}.T)(\hat{W}.t)).F))$.

      iii. If $\hat{W} : \xrightarrow{\text{emp}} \text{done}$, we let $\hat{W}' = \hat{W}$. From $(\hat{W}, \delta_0) \leq_i (\hat{W}, \Delta_0)$, we know there exist $\hat{W}'$ and $\Delta$ such that

      $$\hat{W} \xrightarrow{\Delta} \hat{W}'$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0, ((\hat{W}.T)(\hat{W}.t)).F))$.

   b. $\exists j < i$.

      Since $\neg(\hat{W} \xrightarrow{\text{emp}} \text{abort})$, we know there exists $\hat{W}''$ and $\delta''$ such that

      $$\hat{W} : \xrightarrow{\delta''} \hat{W}''$$

      or $\exists \tau \neq \sigma, \hat{W} : \xrightarrow{\text{emp}} \hat{W}''$, or $\hat{W} : \xrightarrow{\text{emp}} \text{done}$

      We consider only the case of $\hat{W} : \xrightarrow{\delta''} \hat{W}''$ (the other two cases are similar to the above proofs). From $(\hat{W}, \delta_0) \leq_i (\hat{W}, \Delta_0)$, we know one of the following holds:

      i. there exist $\hat{W}''$, $\Lambda''$, and $j$ such that

      $$\hat{W} \xrightarrow{\Lambda''} \hat{W}''$$

      and $j < i$, and $(\hat{W}'', \delta_0 \cup \delta'') \leq_i (\hat{W}'', \Delta_0 \cup \Lambda'')$.

      By the induction hypothesis, we know there exist $\hat{W}'$, $\delta$, $\hat{W}'$ and $\Delta$ such that

      $$\hat{W}' \xrightarrow{\delta} \hat{W}$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta'' \cup \Delta, \hat{W}'', \Sigma), (\delta_0 \cup \delta \cup \delta, ((\hat{W}''.T)(\hat{W}''.t)).F))$.

      By the operational semantics, we know $((\hat{W}''.T)(\hat{W}''.t)).F = ((\hat{W}.T)(\hat{W}.t)).F$. By Lemma 57, we know

      $$\hat{W} \xrightarrow{\delta''} \hat{W}$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta'' \cup \Delta, \hat{W}'', \Sigma), (\delta_0 \cup \delta'' \cup \delta, ((\hat{W}.T)(\hat{W}.t)).F))$.

      ii. there exist $\hat{W}''$, $\Lambda$, and $\Delta$ such that

      $$\hat{W} \xrightarrow{\Lambda} \hat{W}''$$

      and $\text{FPmatch}(\mu, (\Delta_0 \cup \Lambda, \hat{W}, \Sigma), (\delta_0 \cup \delta'', ((\hat{W}.T)(\hat{W}.t)).F))$.

2. $n = m + 1$. Thus there exist $\hat{W}_2, \delta_2$ and $\delta'_2$ such that

$$\hat{W} : \xrightarrow{\delta_2} \hat{W}_2, \quad \hat{W}_2 : \xrightarrow{\delta'_2} \hat{W}_1, \quad \delta_1 = \delta_2 \cup \delta'_2$$

From $(\hat{W}, \delta_0) \leq_i (\hat{W}, \Delta_0)$, we know one of the following holds:

a. there exist $\hat{W}_2, \Lambda_2$ and $j$ such that

$$\hat{W} \xrightarrow{\Lambda_2} \hat{W}_2$$

and $j < i$, and $(\hat{W}_2, \delta_0 \cup \delta_2) \leq_i (\hat{W}_2, \Delta_0 \cup \Delta_2)$.

By the induction hypothesis, we know there exist $\hat{W}'$, $\delta$, $\hat{W}'$ and $\Delta$ such that

$$\hat{W}_2 \xrightarrow{\delta} \hat{W}$$

and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta_2 \cup \Delta, \hat{W}_2, \Sigma), (\delta_0 \cup \delta_2 \cup \delta, ((\hat{W}_2.T)(\hat{W}_2.t)).F))$.

By the operational semantics, we know $((\hat{W}_2.T)(\hat{W}_2.t)).F = ((\hat{W}.T)(\hat{W}.t)).F$. By Lemma 57, we know

$$\hat{W} \xrightarrow{\delta''} \hat{W}$$

and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta'' \cup \Delta, \hat{W}'', \Sigma), (\delta_0 \cup \delta'' \cup \delta, ((\hat{W}.T)(\hat{W}.t)).F))$.

b. there exist $\hat{W}_2$, $\Lambda_2$ and $\Delta_2$ such that

$$\hat{W} \xrightarrow{\Lambda_2} \hat{W}_2$$

and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta_2, \hat{W}, \Sigma), (\delta_0 \cup \delta_2, ((\hat{W}.T)(\hat{W}.t)).F))$ and $(\hat{W}_2, \text{emp}) \leq_i (\hat{W}_2, \text{emp})$.

By the induction hypothesis, we know there exist $\hat{W}'$, $\delta$, $\hat{W}'$ and $\Delta$ such that

$$\hat{W}_2 \xrightarrow{\Lambda} \hat{W}'$$

and $\text{FPmatch}(\mu, (\Delta_0 \cup \Delta, \hat{W}, \Sigma), (\delta_0 \cup \delta, ((\hat{W}.T)(\hat{W}.t)).F))$.  

Towards Certified Separate Compilation for Concurrent Programs

By the operational semantics, we know \((\check{W}, T)(\check{W}, t)).F = ((\check{W}, T)(\check{W}, t)).F\). By Lemma 57, we know
\[
\frac{F}{\Delta_0 \cup \Delta} \quad \frac{F}{\Delta_1 \cup \Delta}
\]
and \(\delta_1 \subseteq \delta_2 \cup \delta_1\), and FPmatch(\(\mu, (\Delta, \check{W}, \Sigma), (\delta_1, ((\check{W}, T)(\check{W}, t)).F)\)), we know

FPmatch(\(\mu, (\Delta_0 \cup \Delta_2 \cup \Delta, \check{W}, \Sigma), (\delta_0 \cup \delta_2 \cup \delta_1, ((\check{W}, T)(\check{W}, t)).F)\)).

Thus we are done. \(\square\)

**Lemma 70.** If \(\Delta_1 \sim \Delta_2\), FPmatch(\(\mu, (\Delta_1, \Sigma), (\delta_1, F_1)\)), FPmatch(\(\mu, (\Delta_2, \Sigma), (\delta_2, F_2)\)), injective(\(\mu, f, \Sigma\)) and \(F_1 \cap F_2 = \emptyset\), then \(\delta_1 \sim \delta_2\).

**Proof:** From FPmatch(\(\mu, (\Delta_1, \Sigma), (\delta_2, F_1)\)) and FPmatch(\(\mu, (\Delta_2, \Sigma), (\delta_2, F_2)\)), we know

\[
\delta_1.\text{rs} \cap \delta_1.\text{ws} \subseteq \left\| F_1 \right\| \cup \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\},
\delta_1.\text{rs} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \leq \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \Delta_1.\text{rs} \right\} \right\} ;
\delta_1.\text{ws} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \leq \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \Delta_1.\text{ws} \right\} \right\} ;
\delta_2.\text{rs} \cup \delta_2.\text{ws} \subseteq \left\| F_2 \right\| \cup \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\},
\delta_2.\text{rs} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \leq \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{rs} \right\} \right\} ;
\delta_2.\text{ws} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \leq \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{ws} \right\} \right\} .
\]

Thus we know

\[
\left\| F_1 \right\| - \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \cap \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{rs} \right\} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} = \emptyset,
\left\| F_2 \right\| - \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \cap \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{ws} \right\} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} = \emptyset.
\]

Since \(F_1 \cap F_2 = \emptyset\), we know

\[
\left\| F_1 \right\| - \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \cap \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{rs} \right\} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} = \emptyset,
\left\| F_2 \right\| - \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} \cap \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{ws} \right\} \cap \mu. f \left\{ \left\{ \text{locs}(\Sigma) \cap \left\| \mu. S \right\| \right\} \right\} = \emptyset.
\]

Since \(\Delta_1 \sim \Delta_2\), we know

\[
\Delta_1.\text{rs} \cap (\Delta_2.\text{rs} \cup \Delta_2.\text{ws}) = \emptyset \quad \text{and} \quad \Delta_1.\text{ws} \cap \Delta_2.\text{ws} = \emptyset.
\]

Since injective(\(\mu, f, \Sigma\)), we know

\[
\mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_1.\text{rs} \right\} \cap \mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{rs} \right\} = \emptyset,
\mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_1.\text{ws} \right\} \cap \mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{ws} \right\} = \emptyset,
\mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_1.\text{rs} \right\} \cap \mu. f \left\{ \text{locs}(\Sigma) \cap \Delta_2.\text{ws} \right\} = \emptyset.
\]

Thus we know

\[
\delta_1.\text{rs} \cap \delta_2.\text{ws} = \emptyset \quad \text{and} \quad \delta_1.\text{ws} \cap \delta_2.\text{ws} = \emptyset.
\]

Thus \(\delta_1 \sim \delta_2\). \(\square\)