Parameterized Memory Models and Concurrent Separation Logic (extended version)

Rodrigo Ferreira
Yale University
rodrigo@cs.yale.edu

Xinyu Feng
Toyota Technological Institute at Chicago
feng@tti-c.org

Zhong Shao
Yale University
shao@cs.yale.edu

Abstract

Formal reasoning about concurrent programs is usually done with the assumption that the underlying memory model is sequentially consistent, i.e. the execution outcome is equivalent to an interleaving of instructions according to the program order. However, memory models in reality are weaker in order to accommodate compiler and hardware optimizations. To simplify the reasoning, many memory models provide a guarantee that data-race-free programs behave in a sequentially consistent manner, the so-called DRF-guarantee. The DRF-guarantee removes the burden of reasoning about relaxations when the program is well-synchronized.

In this paper, we formalize relaxed memory models by giving a parameterized operational semantics to a concurrent programming language. Behaviors of a program under a relaxed memory model are defined as behaviors of a set of *related* programs under the *sequentially consistent model*. This semantics is parameterized in the sense that different memory models can be obtained by using different relations between programs. We present one particular relation that we believe accounts for the majority of memory models and sequential optimizations. We then show that the derived semantics has the DRF-guarantee, using a notion of race-freedom captured by an operational grainless semantics. Our grainless semantics also bridges concurrent separation logic (CSL) and relaxed memory models naturally, which allows us to finally prove the folklore theorem that CSL is sound with relaxed memory models.

1. Introduction

For many years, optimizations of sequential code — by both compilers and architectures — have been the major source of performance improvement for computing systems. Compiler transformations, superscalar pipelines, and memory caches are some of the artifacts used to achieve that. However, these optimizations were designed to preserve only the sequential semantics of the code. When placed in a concurrent context, many of them violate the so-called sequential consistency [Lamport 1979], which requires that the instructions in each thread be executed following the program order.

A classical example to demonstrate this problem is Dekker's mutual exclusion algorithm [Dijkstra 1968] as shown below:

Initially $[x] = [y] = 0$ and $x \neq y$		
$[x] := 1;$		$[y] := 1;$
$v_1 := [y];$	$\|\|$	$v_2 := [x];$
if $v_1 = 0$ **then** *critical section*		**if** $v_2 = 0$ **then** *critical section*

where $[e]$ refers to the memory cell at the location e. Its correctness in the sequentially consistent memory model is ensured by the invariant that we would never have $v_1 = v_2 = 0$ when the conditional statements are reached. However, memory models in reality often relax the ordering of memory accesses and their visibility to other threads to create room for optimizations. Many of them allow reordering of the first two statements in each thread above, thus breaking the invariant. Other synchronization algorithms are susceptible to failure in a similar fashion, which has been a well-known problem [Boehm 2005, Adve and Gharachorloo 1996].

The semantics of concurrent programming languages rely on a formal memory model to rigorously define how threads interact through a shared memory system. Many relaxed memory models have been proposed in the computer architecture community [Dubois et al. 1986, Adve and Hill 1990, Goodman 1989, Gharachorloo et al. 1990]. A tutorial about the subject is given by Adve and Gharachorloo [1996], and a detailed survey is given by Mosberger [1993]. Formalization of memory models for languages such as Java [Manson et al. 2005, Cenciarelli et al. 2007], and C++ [Boehm and Adve 2008] and x86 multiprocessor machine code [Owens et al. 2009] were also developed recently. These models typically allow some relaxation of the program order and provide mechanisms for enforcing ordering when necessary. These mechanisms are commonly referred to as barriers, fences, or strong/ordered operations at the machine level, and locks, synchronization blocks and volatile operations at the high level. The majority of the models provide the so-called DRF-guarantee [Adve and Hill 1993], in which data-race-free programs (i.e. well-synchronized programs) behave in a sequentially consistent manner. DRF-guarantee is also known as the fundamental property [Saraswat et al. 2007] of a memory model. It is desirable because it frees the programmer from reasoning about idiosyncrasies of memory models when the program is well-synchronized.

However, as Boudol and Petri [2009] pointed out in their last year's POPL paper, most memory models are defined axiomatically by giving partial orders of events in the execution traces of programs, which are more abstract than operational semantics of languages that are normally used to model the execution of programs and also to reason about them. Also, they "*only establish a very abstract version of the DRF-guarantee, from which the notion of a program, in the sense of programming languages, is actually absent*" [Boudol and Petri 2009]. This gap, we believe, partly explains why most program logics for concurrency verification are proved sound only in sequentially consistent memory models, and their soundness in relaxed memory models is rarely discussed.

For instance, the soundness of concurrent separation logic (CSL) [O'Hearn 2007] in sequentially consistent models has been proved in various ways [Brookes 2007, Calcagno et al. 2007, Feng et al. 2007, Hobor et al. 2008], which all show directly or indirectly that CSL-verified programs are race-free. So it seems quite obvious that CSL is sound with any memory model that gives the DRF-guarantee, as Hobor et al. [2008] argued that it "*permits only well-synchronized programs to execute, so we can [...] execute in an interleaving semantics or even a weakly consistent memory model*". However, to our best knowledge, this folklore theorem has never been formally proved. Actually proving it is non-trivial,

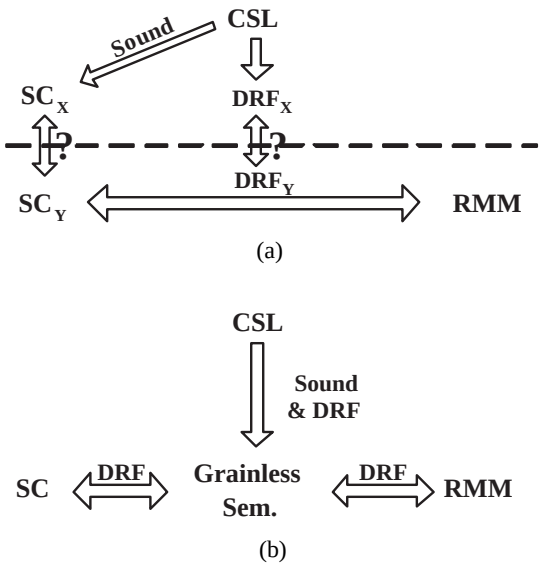

Figure 1. (a) the gap between the language-side (above the dashed line) and the memory-model-side (below the line); we use subscripts X and Y to represent the different formulations in the two sides; (b) our solution: a new RMM and a grainless semantics. Here single arrows represent (informally) logical implications. Double arrows represent logical equivalence, with premises annotated on top. The single arrow and the double arrows on the left and right in (b) correspond to Lemmas 6.8, 5.5 and 5.6 respectively.

and is especially difficult in an operational setting, because the two sides (CSL and memory models) use different semantics of languages and different notions of data-race-freedom (as shown in Fig. 1 (a)).

In this paper, we propose a new approach to formalizing relaxed memory models by giving a parameterized operational semantics to a concurrent programming language. Behaviors of a program under a relaxed memory model are defined as behaviors of a set of *related* programs under the *sequentially consistent model*. This semantics is parameterized in that different relations between programs yield different memory models. We present one particular relation that is weaker than many memory models and accounts for the majority of sequential optimizations. We then give an operational grainless semantics to the language, which gives us an operational notion of data-race-freedom. We show that our derived relaxed semantics has the DRF-guarantee. Our grainless semantics also bridges CSL and relaxed memory models naturally and allows us to prove the soundness of CSL in relaxed memory models. Our paper makes the following new contributions.

First, we propose a simple, operational and parameterized approach to formalizing memory models. We model the behaviors of a program as the behaviors of a set of related programs in the interleaving semantics. The idea is shown by the prototype rule.

$$\frac{(c, c'') \in \Lambda \quad \langle c'', \sigma\rangle \longmapsto \langle c', \sigma'\rangle}{[\Lambda]\ \langle c, \sigma\rangle \longmapsto \langle c', \sigma'\rangle}$$

Our relaxed semantics is parameterized over the relation Λ. At each step, the original program c is substituted with a related program c'', and then c'' executes one step following the normal interleaving semantics. Definition of the semantics is simple: the only difference between it and the standard interleaving semantics is this rule and a corresponding rule that handles the case that a program aborts. It is trivial to see that instantiating Λ with the identity relation gives us a sequentially consistent memory model.

$$\begin{array}{lll}
(\textit{Expr}) & e & ::= n \mid x \mid e_1 \texttt{+} e_2 \mid \texttt{-}e \mid \ldots \\
(\textit{BExpr}) & b & ::= \mathsf{false} \mid b_1 \Rightarrow b_2 \mid e_1 \texttt{=} e_2 \mid e_1 \texttt{<} e_2 \\
(\textit{Comm}) & c & ::= x := e \mid x := [e] \mid [e] := e' \\
 & & \quad \mid \textbf{skip} \mid x := \textbf{cons}(e_1, \ldots, e_n) \mid \textbf{dispose}(e) \\
 & & \quad \mid c_1; c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c \\
 & & \quad \mid c_1 \parallel c_2 \mid \textbf{atomic}\ c
\end{array}$$

Figure 2. Syntax

Second, we give a particular instantiation of Λ — called program subsumption ($\preceq$) — which can relate a sequential segment of a thread between barriers with any other sequential segments that have the same or fewer observational behaviors. This gives programmers a simple and extensional view of relaxed memory models. The derived semantics is weaker than many existing memory models. It allows behaviors such as reordering of any two data independent memory operations, write buffers with read bypassing, and those obtained by the absense of cache coherence and store atomicity.

Third, our semantics gives us a simple way to prove the soundness of sequential program transformations in a relaxed memory model: now we only need to prove the transformations preserve the subsumption relation used to instantiate Λ. Then the DRF-guarantee of our relaxed semantics gives us their soundness in concurrent settings for data-race-free programs. Furthermore, existing works on verification of sequential program transformations [Benton 2004, Leroy 2006, Yang 2007] have developed techniques to prove observational equivalence or simulation relations, which are stronger than this instantiation of Λ. Therefore our work makes it possible to incorporate these techniques into this framework and reuse the existing verification results.

Fourth, we give a grainless semantics to concurrent programs. The semantics is inspired by previous work on grainless trace semantics [Reynolds 2004, Brookes 2006], but it is operational instead of denotational. Since it permits only race-free programs to execute, the semantics gives us an operational formulation of data-race-freedom. As shown in Fig. 1 (b), it also bridges the sequential consistency semantics and our relaxed semantics, which greatly simplifies the proofs of the DRF-guarantee.

Last but not least, we finally give a formal proof of the folklore theorem that CSL is sound in relaxed memory models. As Fig. 1 (b) shows, we first prove that CSL guarantees the data-race-freedom and partial correctness of programs in our grainless semantics. This, combined with the DRF-guarantee of our relaxed semantics, gives us the soundness of CSL in the relaxed model.

2. The Language and Interleaving Semantics

The syntax of the language is presented in Fig. 2. Arithmetic expressions (e) and boolean expressions (b) are pure: they do not access memory. To simplify the presentation, we assume in this paper that parallel threads only share read-only variables, therefore evaluation of expressions would not be interfered by other threads. The command $x := [e]$ reads the value at the memory location e and saves it in x. $[e] := e'$ stores e' at the location e. $x := \textbf{cons}(e_1, \ldots, e_n)$ allocates a fresh memory block containing n consecutive memory cells initialized with values $e_1, \ldots, e_n$. The starting location is non-deterministic and is saved in x. **dispose**(e) frees the memory cell at the location e. The parallel composition $c_1 \parallel c_2$ executes c_1 and c_2 in parallel. **atomic** c ensures that the execution of c is not interrupted by other threads.

$$
\begin{array}{rrcl}
(\textit{Location}) & \ell & ::= & n\ (\textit{natural number}) \\
(\textit{LocSet}) & rs, ws & \in & \mathcal{P}(\textit{Location}) \\
(\textit{Heap}) & h & \in & \textit{Location} \rightharpoonup_{\text{fin}} \textit{Integer} \\
(\textit{Store}) & s & \in & \textit{Variable} \to \textit{Integer} \\
(\textit{State}) & \sigma & ::= & (h, s) \\
(\textit{Footprint}) & \delta & ::= & (rs, ws) \\
(\textit{ThrdTree}) & T & ::= & c \mid \langle\!\langle T, T \rangle\!\rangle c
\end{array}
$$

Figure 3. Runtime objects

$$
\begin{array}{rcl}
\textit{emp} & \stackrel{\text{def}}{=} & (\emptyset, \emptyset) \\
\delta \cup \delta' & \stackrel{\text{def}}{=} & (\delta.rs \cup \delta'.rs, \delta.ws \cup \delta'.ws) \\
\delta \subseteq \delta' & \stackrel{\text{def}}{=} & (\delta.rs \subseteq (\delta'.rs \cup \delta'.ws)) \wedge (\delta.ws \subseteq \delta'.ws) \\
\delta \subset \delta' & \stackrel{\text{def}}{=} & (\delta \subseteq \delta') \wedge (\delta \neq \delta')
\end{array}
$$

Figure 4. Auxiliary definitions

$$
\begin{array}{rrcl}
(\textit{SeqContext}) & \mathbf{E} & ::= & [\,] \mid \mathbf{E}; c \\
(\textit{ThrdContext}) & \mathbf{T} & ::= & [\,] \mid \langle\!\langle \mathbf{T}, T \rangle\!\rangle c \mid \langle\!\langle T, \mathbf{T} \rangle\!\rangle c
\end{array}
$$

Figure 5. Contexts

The command **atomic** c can be viewed as a synchronization block in high-level languages. It is also similar to the "volatile" keyword in Java. On the other hand, we can take a very low-level view and treat **atomic** as an annotation for hardware supported atomic operations with memory barriers. For instance, we can simulate a low-level compare-and-swap (CAS) operation:

atomic $\{\ v := [\ell];$ **if** $v{=}x$ **then** $[\ell] := y$ **else skip**; $y := v\ \}$

Higher-level synchronization primitives such as semaphores and mutexes can be implemented using this primitive construct. Also in this paper we only consider non-nested atomic blocks and we do not have parallel compositions in the block.

Before presenting the operational semantics of the language, we first define the runtime constructs in Fig. 3. Program states consist of heaps and stores. The heap is a partial mapping from memory locations to integer values. The store maps variables to integers. Memory locations are just natural numbers. They are first class values, so the language supports pointer arithmetic. The thread tree is either a command c, which can be viewed as a single thread; or two sub-trees running in parallel, with the parent node c being the command to be executed after the two sub-trees both terminate.

We give a contextual operational semantics for the language. Sequential contexts and thread contexts are defined in Fig. 5. They show the places where the execution of primitive commands occur. Sequential execution of threads is shown in Fig. 6, which is mostly standard. We use $[\![e]\!]_s$ to represent the evaluation of e with the store s. The definition is omitted here. The execution of a normal primitive command is modeled by the labeled transition $(\_ \xrightarrow[\delta]{\mathbf{u}} \_)$. Here the footprint δ is defined in Fig. 3 as a pair (rs, ws), which records the memory locations that are read and written in this step. Recording the footprint allows us to discuss races between threads in the following sections. Since we assume threads only share read-only variables, accesses of variables do not cause races and we do not record variables in footprints. A step aborts if it accesses memory locations that are not in the domain of the heap.

The transition $(\_ \xrightarrow[\delta]{\mathbf{o}} \_)$ models the execution of **cons** and **dispose**. We use the label $\mathbf{o}$ instead of $\mathbf{u}$ to distinguish them from other commands. They are at higher abstraction levels than other primitive commands that may have direct hardware implementations, but we decide to support them in our language because they are important high-level language constructs. Their implementations usually require synchronizations to be thread-safe, so we model them as built-in synchronized operations and they cannot be reordered in our relaxed semantics. In this paper we call them (along with atomic blocks and fork/join of threads) *ordered operations*. Remaining operations are called *unordered*.

We may omit the footprint δ and the labels $\mathbf{u}$ and $\mathbf{o}$ when they are not relevant. We also use R^* to represent the reflexive transitive closure of the relation R. For instance, we use $(\_ \xrightarrow[\delta]{} \_)$ to represent the union of ordered and unordered transitions, and use $(\_ \longrightarrow \_)$ to ignore the footprint, whose reflexive transitive closure is represented by $(\_ \longrightarrow^* \_)$.

Figure 7 defines the interleaving semantics of concurrent programs. Following Vafeiadis and Parkinson [2007], the execution of c in **atomic** c does not interleave with the environment. If c does not terminate, the thread gets stuck. Again, we assume there is no atomic blocks and parallel compositions in c.

Below we give a very simple example to help readers understand our use of contexts and thread trees.

Example 2.1. Suppose $c = (c_1 \parallel c_2); c'$. Then we know $c = \mathbf{T}[\,\mathbf{E}[\,c_1 \parallel c_2\,]\,]$, where $\mathbf{T} = [\,]$ and $\mathbf{E} = [\,]; c'$. After one step, we reach the thread tree $\langle\!\langle c_1, c_2 \rangle\!\rangle(\textbf{skip}; c')$. Then the $\mathbf{T}'$ for the next step can be either $\langle\!\langle [\,], c_2 \rangle\!\rangle(\textbf{skip}; c')$ or $\langle\!\langle c_1, [\,] \rangle\!\rangle(\textbf{skip}; c')$. □

Comparing with standard semantics (e.g. Vafeiadis and Parkinson [2007]), our execution of c above has extra steps caused by the construction of the thread tree and the insertion of **skip** in the front of c'. They can be viewed simply as stuttering steps.

3. Parameterized Relaxed Semantics

In this section, we present our parameterized operational semantics. Then we instantiate it with a relation between sequential programs to capture relaxed memory models and compiler optimizations.

3.1 Parameterized semantics

Figure 8 shows the two new rules of our parameterized semantics. The stepping relation takes Λ as a parameter, which is a binary relation between thread trees:

$$\Lambda \in \mathcal{P}(\textit{ThrdTree} * \textit{ThrdTree})$$

The semantics follows the interleaved semantics presented in Fig. 7, except that at any given step, the current thread tree can be replaced by another thread tree related through the Λ relation. Λ is supposed to provide a set of thread trees that are somehow related to the current thread tree using some notion of equivalence. This Λ-based semantics chooses nondeterministically which command will execute. Therefore, in order to reason about this semantics, one needs to consider all possible commands related through a given instantiation of Λ.

Naturally, different instantiations of Λ yield different semantics. As one can see, this semantics is trivially equivalent to the interleaving semantics shown in Fig. 7 once Λ is instantiated with an identity relation. A more interesting relation to be used as an instantiation of Λ is presented in the following sections.

3.2 Command subsumption

We define a command subsumption relation that

1. preserves synchronized operations of the code;

$$
\begin{array}{rcll}
\langle \mathbf{E}[\, x := e \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{u}} & \langle \mathbf{E}[\, \mathbf{skip} \,],\, (h,s')\rangle & \text{if } [\![e]\!]_s = n \text{ and } s' = s[x \rightsquigarrow n] \\
\langle \mathbf{E}[\, x := e \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{u}} & \mathsf{abort} & \text{otherwise} \\
\langle \mathbf{E}[\, x := [e] \,],\, (h,s)\rangle & \xrightarrow[(\{\ell\},\emptyset)]{\mathbf{u}} & \langle \mathbf{E}[\, \mathbf{skip} \,],\, (h,s')\rangle & \text{if } [\![e]\!]_s = \ell,\ h(\ell) = n, \text{ and } s' = s[x \rightsquigarrow n] \\
\langle \mathbf{E}[\, x := [e] \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{u}} & \mathsf{abort} & \text{otherwise} \\
\langle \mathbf{E}[\, [e] := e' \,],\, (h,s)\rangle & \xrightarrow[(\emptyset,\{\ell\})]{\mathbf{u}} & \langle \mathbf{E}[\, \mathbf{skip} \,],\, (h',s)\rangle & \text{if } [\![e]\!]_s = \ell,\ [\![e']\!]_s = n, \ell \in \mathsf{dom}(h) \text{ and } h' = h[\ell \rightsquigarrow n] \\
\langle \mathbf{E}[\, [e] := e' \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{u}} & \mathsf{abort} & \text{otherwise} \\
\langle \mathbf{E}[\, \mathbf{skip}; c \,],\, \sigma\rangle & \xrightarrow[emp]{\mathbf{u}} & \langle \mathbf{E}[\, c \,],\, \sigma\rangle & \\
 & \dots & & \dots \\
\hline
\langle \mathbf{E}[\, x := \mathbf{cons}(e_1,\dots,e_k) \,],\, (h,s)\rangle & \xrightarrow[(\emptyset,ws)]{\mathbf{o}} & \langle \mathbf{E}[\, \mathbf{skip} \,],\, (h',s')\rangle & \text{if } ws = \{\ell,\dots,\ell{+}k{-}1\}, ws \cap \mathsf{dom}(h) = \emptyset, [\![e_i]\!]_s = n_i \\
 & & & \quad s' = s[x \rightsquigarrow \ell] \text{ and } h' = h[\ell \rightsquigarrow n_1,\dots,\ell{+}k{-}1 \rightsquigarrow n_k] \\
\langle \mathbf{E}[\, x := \mathbf{cons}(e_1,\dots,e_k) \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{o}} & \mathsf{abort} & \text{otherwise} \\
\langle \mathbf{E}[\, \mathbf{dispose}(e) \,],\, (h,s)\rangle & \xrightarrow[(\emptyset,\{\ell\})]{\mathbf{o}} & \langle \mathbf{E}[\, \mathbf{skip} \,],\, (h',s)\rangle & \text{if } [\![e]\!]_s = \ell,\ \ell \in \mathsf{dom}(h), \text{ and } h' = h \backslash \{\ell\} \\
\langle \mathbf{E}[\, \mathbf{dispose}(e) \,],\, (h,s)\rangle & \xrightarrow[emp]{\mathbf{o}} & \mathsf{abort} & \text{otherwise} \\
\hline
\langle c,\, \sigma\rangle & \xrightarrow[\delta]{} & \langle c',\, \sigma'\rangle & \text{if } \langle c,\, \sigma\rangle \xrightarrow[\delta]{\mathbf{u}} \langle c',\, \sigma'\rangle \text{ or } \langle c,\, \sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \langle c',\, \sigma'\rangle \\
\langle c,\, \sigma\rangle & \xrightarrow[\delta]{} & \mathsf{abort} & \text{if } \langle c,\, \sigma\rangle \xrightarrow[\delta]{\mathbf{u}} \mathsf{abort} \text{ or } \langle c,\, \sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \mathsf{abort}
\end{array}
$$

Figure 6. Sequential footprint semantics

$$
\begin{array}{rcll}
\langle \mathbf{T}[\, c \,],\, \sigma\rangle & \longmapsto & \langle \mathbf{T}[\, c' \,],\, \sigma'\rangle & \text{if } \langle c,\, \sigma\rangle \longrightarrow \langle c',\, \sigma'\rangle \\
\langle \mathbf{T}[\, c \,],\, \sigma\rangle & \longmapsto & \mathsf{abort} & \text{if } \langle c,\, \sigma\rangle \longrightarrow \mathsf{abort} \\
\langle \mathbf{T}[\, \mathbf{E}[\, \mathbf{atomic}\ c \,] \,],\, \sigma\rangle & \longmapsto & \langle \mathbf{T}[\, \mathbf{E}[\, \mathbf{skip} \,] \,],\, \sigma'\rangle & \text{if } \langle c,\, \sigma\rangle \longrightarrow^* \langle \mathbf{skip},\, \sigma'\rangle \\
\langle \mathbf{T}[\, \mathbf{E}[\, \mathbf{atomic}\ c \,] \,],\, \sigma\rangle & \longmapsto & \mathsf{abort} & \text{if } \langle c,\, \sigma\rangle \longrightarrow^* \mathsf{abort} \\
\langle \mathbf{T}[\, \mathbf{E}[\, c_1 \parallel c_2 \,] \,],\, \sigma\rangle & \longmapsto & \langle \mathbf{T}[\, \langle\!\langle c_1, c_2 \rangle\!\rangle \mathbf{E}[\, \mathbf{skip} \,] \,],\, \sigma\rangle & \\
\langle \mathbf{T}[\, \langle\!\langle \mathbf{skip}, \mathbf{skip} \rangle\!\rangle c \,],\, \sigma\rangle & \longmapsto & \langle \mathbf{T}[\, c \,],\, \sigma\rangle &
\end{array}
$$

Figure 7. Interleaving semantics of concurrent programs

$$
\begin{array}{lll}
[\Lambda]\ \langle T,\, \sigma\rangle \longmapsto \langle T',\, \sigma'\rangle & \text{if } \exists T''.\ (T,T'') \in \Lambda \wedge \langle T'',\, \sigma\rangle \longmapsto \langle T',\, \sigma'\rangle \\
[\Lambda]\ \langle T,\, \sigma\rangle \longmapsto \mathsf{abort} & \text{if } \exists T'.\ (T,T') \in \Lambda \wedge \langle T',\, \sigma\rangle \longmapsto \mathsf{abort}
\end{array}
$$

Figure 8. Semantics parameterized over Λ

2. but permits the rewriting of non-synchronized sequential portions while preserving their *sequential* semantics.

The intuition is that programs in relaxed memory models should be well-synchronized to avoid unexpected behaviors. That is, accesses to shared memory should be performed through synchronized operations (**cons**, **dispose** and **atomic** c in our language), and non-synchronized (unordered) operations should only access thread-local or read-only memory (but note that the term "shared" and "local" are dynamic notions and their boundary does not have to be fixed). Therefore, the effect of a thread's non-synchronized code is not visible to other threads until the next synchronized point is reached. On the other hand, the behavior of the non-synchronized code will not be affected by other threads since the data it uses would not be updated by others. So we do not need to consider its interleaving with other threads.

The subsumption of c_1 by c_2 ($c_1 \preceq c_2$) is defined below. Here ($\_ \xrightarrow[\delta]{\mathbf{u}}{}^* \_$) represents zero or multiple steps of unordered transitions, where δ is the union of the footprints of individual steps. $\langle c,\, \sigma\rangle \Downarrow \langle c',\, \sigma'\rangle$ is a big-step transition of unordered operations. From the definition shown in Fig. 9, we know c' must be either **skip**, or a command starting with an ordered operation.

Definition 3.1. $c_1 \preceq_0 c_2$ always holds; $c_1 \preceq_{k+1} c_2$ holds if and only if, for all $j \leq k$, the following are true:

1. If $\langle c_1,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, then $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$;
2. If $\langle c_1,\, \sigma\rangle \Downarrow \langle c_1',\, \sigma'\rangle$, then either $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, or there exists c_2' such that $\langle c_2,\, \sigma\rangle \Downarrow \langle c_2',\, \sigma'\rangle$ and the following constraints hold:
   (a) if $c_1' = \mathbf{skip}$, then $c_2' = \mathbf{skip}$;

$$
\begin{array}{llll}
\langle c,\,\sigma\rangle & \xrightarrow[emp]{\mathbf{u}}{}^{0} & \langle c,\,\sigma\rangle & \text{always}\\
\langle c,\,\sigma\rangle & \xrightarrow[\delta]{\mathbf{u}}{}^{k+1} & \langle c',\,\sigma'\rangle & \text{if there exist } c'',\,\sigma'',\,\delta',\text{ and } \delta'' \text{ such that } \langle c,\,\sigma\rangle \xrightarrow[\delta']{\mathbf{u}} \langle c'',\,\sigma''\rangle,\\
 & & & \langle c'',\,\sigma''\rangle \xrightarrow[\delta'']{\mathbf{u}}{}^{k} \langle c',\,\sigma'\rangle \text{ and } \delta = \delta' \cup \delta''\\
\langle c,\,\sigma\rangle & \Downarrow_{\delta} & \langle c',\,\sigma'\rangle & \text{if } \langle c,\,\sigma\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*} \langle c',\,\sigma'\rangle,\ \neg(\langle c',\,\sigma'\rangle \xrightarrow{\mathbf{u}} \mathsf{abort}),\\
 & & & \text{and } \neg\exists c'', \sigma''.(\langle c',\,\sigma'\rangle \xrightarrow{\mathbf{u}} \langle c'',\,\sigma''\rangle)\\
\langle c,\,\sigma\rangle & \Downarrow & \langle c',\,\sigma'\rangle & \text{if there exists } \delta \text{ such that } \langle c,\,\sigma\rangle \Downarrow_{\delta} \langle c',\,\sigma'\rangle\\
\hline
\langle c,\,\sigma\rangle & \xrightarrow[emp]{}{}^{0} & \langle c,\,\sigma\rangle & \text{always}\\
\langle c,\,\sigma\rangle & \xrightarrow[\delta]{}{}^{k+1} & \langle c',\,\sigma'\rangle & \text{if there exist } c'',\,\sigma'',\,\delta',\text{ and } \delta'' \text{ such that } \langle c,\,\sigma\rangle \xrightarrow[\delta']{} \langle c'',\,\sigma''\rangle,\\
 & & & \langle c'',\,\sigma''\rangle \xrightarrow[\delta'']{}{}^{k} \langle c',\,\sigma'\rangle \text{ and } \delta = \delta' \cup \delta''
\end{array}
$$

Figure 9. Multi-step sequential transitions

(b) if $c_1' = \mathbf{E}_1[\, c_1'' \parallel c_1''' \,]$, there exist $\mathbf{E}_2$, c_2'' and c_2''' such that
  i. $c_2' = \mathbf{E}_2[\, c_2'' \parallel c_2''' \,]$;
  ii. $c_1'' \preceq_j c_2''$ and $c_1''' \preceq_j c_2'''$;
  iii. $\mathbf{E}_1[\,\textbf{skip}\,] \preceq_j \mathbf{E}_2[\,\textbf{skip}\,]$;

(c) if $c_1' = \mathbf{E}_1[\,\textbf{atomic}\ c_1''\,]$, there exist $\mathbf{E}_2$ and c_2'' such that
  i. $c_2' = \mathbf{E}_2[\,\textbf{atomic}\ c_2''\,]$;
  ii. $c_1'' \preceq_j c_2''$;
  iii. $\mathbf{E}_1[\,\textbf{skip}\,] \preceq_j \mathbf{E}_2[\,\textbf{skip}\,]$;

(d) if $c_1' = \mathbf{E}_1[\, c_1''\,]$, where c_1'' is a **cons** or **dispose** command, there exist $\mathbf{E}_2$ and c_2'' such that
  i. for all σ, if $\langle c_1'',\,\sigma\rangle \xrightarrow{\mathbf{o}} \mathsf{abort}$, then $\langle c_2'',\,\sigma\rangle \xrightarrow{\mathbf{o}} \mathsf{abort}$;
  ii. for all σ and σ', if $\langle c_1'',\,\sigma\rangle \xrightarrow{\mathbf{o}} \langle\textbf{skip},\,\sigma'\rangle$, then $\langle c_2'',\,\sigma\rangle \xrightarrow{\mathbf{o}} \langle\textbf{skip},\,\sigma'\rangle$;
  iii. $\mathbf{E}_1[\,\textbf{skip}\,] \preceq_j \mathbf{E}_2[\,\textbf{skip}\,]$.

3. If $\langle c_1,\,\sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}}{}^{*} \langle c_1',\,\sigma'\rangle$, then either $\langle c_2,\,\sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$, or there exist δ_2, c_2' and σ'' such that $\langle c_2,\,\sigma\rangle \xrightarrow[\delta_2]{\mathbf{u}}{}^{*} \langle c_2',\,\sigma''\rangle$ and $\delta_1 \subseteq \delta_2$;

We define $c_1 \preceq c_2$ as $\forall k.\ c_1 \preceq_k c_2$; and $c_1 \succeq c_2$ as $c_2 \preceq c_1$. □

Informally, we say c_1 is subsumed by c_2 if for all input states — afters performing a sequential big step — c_1 aborts only if c_2 aborts; or, if c_1 completes, then either c_2 aborts or there is a sequential big step taken by c_2 that ends in the same state. Also, if c_1 completes the big step and the execution terminates (**skip** case) or reaches a synchronization point (cases for thread fork and join, atomic blocks, **cons** and **dispose**), there must be a corresponding synchronization point at the end of the big step taken by c_2 and the remaining parts of c_1 and c_2 still satisfy the relation. We use indices in the definition since $\mathbf{E}_1[\,\textbf{skip}\,]$ in cases 2(b), 2(c) and 2(d) might be "larger" than c_1. The last condition requires that the footprint of c_1 is not larger than that of c_2 if c_2 does not abort. The subset relation between footprints is defined in Fig. 3.

Properties of subsumption. Observe that the big step is deterministic, therefore, if c_1 and c_2 are sequential programs and $c_1 \preceq c_2$, then for any input state we have one of the following possibilities:

1. c_2 aborts and c_1 may have any behaviors;
2. c_1 and c_2 complete a big step and reach the same state;
3. c_1 diverges and c_2 may have any behaviors.

Here we intend to use c_2 to represent the original program and c_1 the one after optimizations (by compilers or hardware). By the three cases above we know c_1 preserves the partial correctness of c_2 [Calcagno et al. 2007] (to handle total correctness, an extra condition must be added to Definition 3.1 to ensure that normal termination is preserved by subsumption). The last condition in Definition 3.1 is also necessary to ensure the transformation from c_2 to c_1 does not introduce new races. We give examples in Sec. 4 to show the expressiveness of the subsumption relation and how it models behaviors of programs in relaxed memory models.

Lemma 3.2 below states that the subsumption relation is preserved when c_1 completes a big step and c_2 does not abort.

Lemma 3.2. If $c_1 \preceq c_2$ and $\langle c_1,\,\sigma\rangle \Downarrow \langle c_1',\,\sigma'\rangle$, then either $\langle c_2,\,\sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ or there exists c_2' such that $\langle c_2,\,\sigma\rangle \Downarrow \langle c_2',\,\sigma'\rangle$ and $c_1' \preceq c_2'$.

The following two lemmas are useful if we view $c_1 \succeq c_2$ as a static compiler transformation from c_1 to c_2 (see examples in Sec. 4.6).

Lemma 3.3. The relation $\preceq$ is reflexive and transitive.

Lemma 3.3 shows that both the identity transformation and the composition of multiple transformations — given that they obey subsumption — do not violate subsumption. It is useful when composing smaller transformations into large complex sequences of transformations.

Lemma 3.4. If $c_1 \preceq c_2$, then, for all contexts $\mathcal{C}$, $\mathcal{C}[\,c_1\,] \preceq \mathcal{C}[\,c_2\,]$.

Lemma 3.4 ensures that local transformations that obey subsumption also hold in any larger context. This helps proving that a given transformation obeys subsumption in a modular fashion. Note that $\mathcal{C}$ does not have to be an execution context $\mathbf{E}$. It can be any context, i.e. a program with a hole in it.

3.3 Relaxed semantics

The subsumption relation can be lifted for thread trees.

Definition 3.5. We define the binary relation $\preceq_{\mathbf{t}}$ for thread trees.

$$
T_1 \preceq_{\mathbf{t}} T_2 \stackrel{\text{def}}{=} \begin{cases} c_1 \preceq c_2 & \text{if } T_1 = c_1 \text{ and } T_2 = c_2\\ c_1 \preceq c_2 \wedge T_1' \preceq_{\mathbf{t}} T_2' \wedge T_1'' \preceq_{\mathbf{t}} T_2'' & \text{if } T_1 = \langle\!\langle T_1', T_1''\rangle\!\rangle c_1 \text{ and } T_2 = \langle\!\langle T_2', T_2''\rangle\!\rangle c_2 \end{cases}
$$

We use $T_1 \succeq_{\mathbf{t}} T_2$ to represent $T_2 \preceq_{\mathbf{t}} T_1$. □

We obtain a relaxed operational semantics by instantiating Λ of our parameterized semantics with this relation. The resulting

stepping relation becomes

$$[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto \langle T',\, \sigma'\rangle\,.$$

This semantics performs a program transformation, following our subsumption relation, at each step. This resembles a dynamic compiler that modifies the program as it executes.

On the other hand, as we show in Lemma 3.6, the execution according to this semantics is equivalent to performing one single initial program transformation and then executing the target program using the interleaving semantics. This resembles a static compiler that modifies the program prior to execution. Similarly, Lemma 3.7 shows the abort case.

Lemma 3.6. $[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto^* \langle \textbf{skip},\, \sigma'\rangle$ iff there exists a T' such that $T \succeq_{\mathbf{t}} T'$ and $\langle T',\, \sigma\rangle \longmapsto^* \langle \textbf{skip},\, \sigma'\rangle$.

Lemma 3.7. $[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto^* \mathsf{abort}$ iff there exists a T' such that $T \succeq_{\mathbf{t}} T'$ and $\langle T',\, \sigma\rangle \longmapsto^* \mathsf{abort}$.

We will formulate and prove the DRF-guarantee of this relaxed semantics in Sec. 5, after we formally define data-race-freedom.

4. Examples

There are different aspects that characterize a particular memory model, including memory ordering constraints, support for general compiler transformations, write atomicity constraints, presence of write buffers, cache coherence protocols, availability of memory barriers, etc. In this section, we show how some of these aspects are reflected in our semantics. Our goal is to familiarize the reader with the $\succeq$ relation. The examples are written using the following naming convention: $v1$, $v2$, $v3$, etc, are variables that hold values; x, y, z, etc, are variables that hold memory addresses.

4.1 Data dependencies

Before we discuss the memory ordering of our model, we need to make it clear that we can support precise discovery of data dependencies. We do it by showing the following example:

$$[x] := 1; v1 := [y]$$

In this small program, the data dependency between the two statements exists only for those initial states where x and y are aliased. At first glance, our $\succeq$ definition is too restrictive since its definition quantifies over *all* input states. So it does not allow the following:

$$([x] := 1; v1 := [y]) \succeq (v1 := [y]\,;\, [x] := 1)$$

However, through the $\succeq$ relation, we can obtain the following transformation:

$$\begin{array}{c} [x] := 1; v1 := [y] \\ \succeq \\ \textbf{if } x{=}y \textbf{ then } ([x] := 1; v1 := [x]) \textbf{ else } (v1 := [y]\,;\, [x] := 1) \end{array}$$

where we insert a dynamic test to see if x is an alias of y. We also replace y by x in one branch where there is dependency, and reorder the statements in the other branch. Based on this example, one can convince himself that the relaxed semantics will allow the reordering memory accesses that do not have data dependencies at runtime. But, the reader should also be aware that the $\succeq$ relation does not violate data dependencies:

$$\neg([x] := 1;\, [x] := 2 \succeq [x] := 2;\, [x] := 1)$$

4.2 Memory ordering

From the example just shown, it is not hard to see that the $\succeq$ relation supports all 4 types of memory reordering (R,W $\rightarrow$ R,W). Examples of this can be seen below (we use the context **if** $x{=}y$ **then skip else** $[\,]$ but these are supported in any context where $x \neq y$ can be inferred):

- Reads with reads:

$$\begin{array}{c} \textbf{if } x{=}y \textbf{ then skip else } (v1 := [x]\,;\, v2 := [y]) \\ \succeq \\ \textbf{if } x{=}y \textbf{ then skip else } (v2 := [y]\,;\, v1 := [x]) \end{array}$$

- Reads with writes:

$$\begin{array}{c} \textbf{if } x{=}y \textbf{ then skip else } (v1 := [x]\,;\, [y] := 2) \\ \succeq \\ \textbf{if } x{=}y \textbf{ then skip else } ([y] := 2;\, v1 := [x]) \end{array}$$

- Writes with reads:

$$\begin{array}{c} \textbf{if } x{=}y \textbf{ then skip else } ([x] := 1;\, v2 := [y]) \\ \succeq \\ \textbf{if } x{=}y \textbf{ then skip else } (v2 := [y]\,;\, [x] := 1) \end{array}$$

- Writes after writes:

$$\begin{array}{c} \textbf{if } x{=}y \textbf{ then skip else } ([x] := 1;\, [y] := 2) \\ \succeq \\ \textbf{if } x{=}y \textbf{ then skip else } ([y] := 2;\, [x] := 1) \end{array}$$

4.3 Write buffer with read bypassing

A write buffer is a hardware feature that delays writes to memory in an attempt to overlap the latency of writes with subsequent code. The actual behavior obtained is that a processor might read its own writes earlier, i.e. before they are actually committed to memory. This can be supported by a simple program transformation as seen in the example below:

$$[x] := 1; v2 := [x] \succeq v2 := 1;\, [x] := 1$$

4.4 Redundancy introduction and elimination

Redundant memory reads and writes can be introduced and eliminated, as shown by the following examples:

$$\begin{array}{rl} v1 := [x]; v2 := 1 & \succeq v1 := [x]; v2 := [x]; v2 := 1 \\ v1 := [x]; v2 := [x] & \succeq v1 := [x]; v2 := v1 \\ [x] := v1; v2 := [x] & \succeq [x] := v1; v2 := v1 \\ [x] := v1 & \succeq [x] := 1;\, [x] := v1 \\ [x] := 1;\, [x] := v1 & \succeq [x] := v1 \end{array}$$

Furthermore, we can eliminate dead memory operations when that yields a smaller memory footprint:

$$v1 := [x]; v1 := 1 \succeq v1 := 1$$

Note that the reverse is not true. A program cannot increase its footprint given the $\succeq$ relation. Recall that $c_1 \succeq c_2$ requires c_1 to abort whenever c_2 aborts; therefore if the footprint of c_2 is larger than the footprint of c_1 there will be a case where c_2 aborts but c_1 does not, which is in conflict with the definition of $\succeq$.

$$\neg(v1 := 1 \succeq v1 := [x]; v1 := 1)$$

4.5 Write atomicity

Given the $\succeq$ relation, write atomicity is not preserved. This might not be clear at first, but can be shown in the example below (here $v1$ is a temporary, we assume it is reused later on, by the artifact of assigning an arbitrary value to it):

$$\begin{array}{c} [x] := 1; v1 := 42 \\ \succeq \\ v1 := [x];\, [x] := 1;\, [x] := v1;\, [x] := 1; v1 := 42 \end{array}$$

Here the write is replaced by 3 writes, oscillating between the original write and the write of the initial value into the memory location. In fact, it might oscillate with any value (not only the

initial value) as shown below:

$$[x]:=1 \\ \succeq \\ [x]:=1;\,[x]:=42;\,[x]:=1;\,[x]:=69;\,[x]:=1$$

therefore, a write can store arbitrary values to memory before completing; which in practice means that the memory value is undefined until the write completes.

4.6 Compiler optimizations

The $\succeq$ relation is general enough to support many sequential compiler optimizations. For instance, it is not hard to see that we can support instruction scheduling

$$v3:=v1{+}v2;\,v6:=v4{+}v5;\,v7:=v3{+}v6 \\ \succeq \\ v6:=v4{+}v5;\,v3:=v1{+}v2;\,v7:=v3{+}v6$$

algebraic transformations (here again we assume $v4$ is a temporary)

$$v4:=v1{+}v2;\,v5:=v4{+}v3;\,v4:=42 \\ \succeq \\ v4:=v2{+}v3;\,v5:=v1{+}v4;\,v4:=42$$

register allocation (we have to test for aliasing of z and w)

$$\begin{array}{l} v1:=[x];\,v2:=[y];\,v3:=v1{+}v2;\,[w]:=v3; \\ v1:=[z];\,v2:=[w];\,v3:=v1{+}v2;\,[w]:=v3 \\ \succeq \\ v1:=[x];\,v2:=[y];\,v3:=v1{+}v2; \\ (\textbf{if } z{=}w \textbf{ then } v1:=v3 \textbf{ else } v1:=[z]); \\ v2:=v3;\,v3:=v1{+}v2;\,[w]:=v3 \end{array}$$

and many others, including control transformations and redundancy elimination such as the ones already presented in Section 4.4.

4.7 Concurrent behaviors

Here we present some concurrent behaviors of the semantics yielded by $\succeq$. In all examples we assume a sequential interleaving of commands according to the standard semantics after considering a program transformation through the $\succeq$ relation. We also assume initial memory values are all 0. We start with the following example (not supported by Boudol and Petri [2009])

$$(v1:=[x];\,[y]:=1) \;\parallel\; (v2:=[y];\,[x]:=1)$$

in which we can perceive $v1{=}v2{=}1$ if $x{\neq}y$. It can be supported in our semantics by reordering the commands in the second thread,

$$v2:=[y];\,[x]:=1 \\ \succeq \\ \textbf{if } x{=}y \textbf{ then } (v2:=[x];\,[x]:=1) \textbf{ else } ([x]:=1;\,v2:=[y])$$

yielding a new program that produces the desired result through an interleaved scheduling. Similarly, we can support the classic crossover example:

$$([x]:=1;\,v1:=[x]) \;\parallel\; ([x]:=2;\,v2:=[x])$$

in which we can perceive $v1=2$ and $v2=1$. That is achieved by inserting a redundant write in the right hand side thread:

$$[x]:=2;\,v2:=[x] \succeq [x]:=2;\,v2:=[x];\,[x]:=2$$

Yet another similar example is the prescient write test:

$$(v1:=[x];\,[x]:=1) \;\parallel\; (v2:=[x];\,[x]:=v2)$$

where we could perceive $v1=v2=1$. That is also supported by inserting redundant writes and reads in the left hand side thread:

$$v1:=[x];\,[x]:=1 \\ \succeq \\ v1:=[x];\,[x]:=1;\,[x]:=v1;\,v1:=[x];\,[x]:=1$$

As one can seen, the semantics derived from $\succeq$ leads to possibly unwanted behaviors of raceful programs. First, a read from a shared location can return any value. For instance, there is a scheduling of the program below:

$$v1:=[x] \;\parallel\; [x]:=1$$

where $v1=33$ is allowed. That happens if we consider the following replacement of the right hand side thread:

$$[x]:=1 \succeq [x]:=33;\,[x]:=1$$

This is commonly referred to as "out-of-thin-air" behavior. A similar behavior happens when we have simultaneous write to the same location:

$$(v1:=1;\,[x]:=v1) \;\parallel\; [x]:=2$$

in this case, the final value of $[x]$ can also be arbitrary. For instance, it could be 3 if we replace the left hand side thread as below

$$v1:=1;\,[x]:=v1 \succeq [x]:=0;\,v1:=[x];\,v1:=v1{+}1;\,[x]:=v1$$

Another unwanted behavior happens when the implementations of mutual exclusions rely on memory ordering (such as the core of Dekker's algorithm presented earlier):

$$([x]:=1;\,v1:=[y]) \;\parallel\; ([y]:=1;\,v2:=[x])$$

In this case, we would not want the behavior $v1=v2=0$ to happen. However, it may happen if we consider the reordering of the two commands of the right hand side thread:

$$[y]:=1;\,v2:=[x] \\ \succeq \\ \textbf{if } x{=}y \textbf{ then } ([x]:=1;\,v2:=[x]) \textbf{ else } (v2:=[x];\,[y]:=1)$$

Note that we assumed initial values $[x]=[y]=0$ and $x\neq y$.

Many other examples of raceful code can be shown to have unwanted behaviors in such a relaxed execution. They are obtained by either reordering of memory operations or relying on the non-atomic undefined nature of raceful reads and writes. On the other hand, race-free programs do not have unwanted behaviors (see the DRF-guarantee in Section 5). In the example below:

$$\begin{pmatrix} v1:=[x]; \\ \textbf{if } v1{=}1 \textbf{ then } [y]:=1 \end{pmatrix} \;\parallel\; \begin{pmatrix} v2:=[y]; \\ \textbf{if } v2{=}1 \textbf{ then } [x]:=1 \end{pmatrix}$$

the only behavior allowed is $v1=v2=0$. Its data-race-freedom might not be obvious, but there are no sequentially consistent executions of this program that may reach the code within the branches (assuming $[x]=[y]=0$ and $x\neq y$ initially). So, the program never issues a memory write, therefore it is race-free. And if you consider the code of each one of the threads in isolation — through the $\succeq$ relation — it is impossible to insert a race when the initial state has $[x]=[y]=0$. That is guaranteed from the fact that the footprints of both threads are disjoint, and they can only decrease through the $\succeq$ relation.

4.8 Strong barrier

In our relaxed semantics, we can enforce both atomicity and ordering by using the **atomic** c command. In the following examples we use the macro MF (memory fence) as a syntactic sugar for **atomic skip**, a command that does nothing but enforces ordering.

The first example we analyze is about cache coherence. Cache coherence ensures that everybody agrees on the order of writes to the same location. Since the $\succeq$ relation does not preserve the atomicity of writes, coherence is not preserved by the semantics, as can be seen in the following example:

$$[x]:=1 \;\parallel\; [x]:=2 \;\parallel\; \begin{pmatrix} v1:=[x]; \\ \mathsf{MF}; \\ v2:=[x] \end{pmatrix} \;\parallel\; \begin{pmatrix} v3:=[x]; \\ \mathsf{MF}; \\ v4:=[x] \end{pmatrix}$$

in which the outcome $v1 = v4 = 1$ and $v2 = v3 = 2$ can be noticed once we rewrite the leftmost thread as

$$[x] := 1 \succeq [x] := 1; [x] := 1$$

Another related example is the independent-reads-independent-writes (IRIW) example shown below

$$[x] := 1 \;\|\; [y] := 1 \;\|\; \begin{pmatrix} v1 := [x]; \\ \mathsf{MF}; \\ v2 := [y] \end{pmatrix} \;\|\; \begin{pmatrix} v3 := [y]; \\ \mathsf{MF}; \\ v4 := [x] \end{pmatrix}$$

where the behavior $v1 = v3 = 1$ and $v2 = v4 = 0$ is permissible (again assuming $[x] = [y] = 0$ initially). That can be perceived in our semantics once we replace the leftmost thread through

$$[x] := 1 \succeq [x] := 1; [x] := 0; [x] := 1$$

Other similar examples shown by Boehm and Adve [2008] can also be supported. It might be intuitive that they happen because $\succeq$ does not enforce write atomicity.

4.9 Preventing "out-of-thin-air" reads

In the context of typed languages, such as Java, the DRF-guarantee is not sufficient to ensure type safety, as the type system does not enforce race-freedom. Therefore, behaviors such as the "out-of-thin-air" example must not be allowed as they can break the type system. That would compromise not only the language safety but also its security. Much of the complexity of the JMM comes from the fact that it should forbid such behaviors while still allowing all other type safe optimizations.

In our setting, from the examples shown, it is clear that the $\succeq$ relation is not suited for type safe languages. However, specific memory models can be obtained by simply constraining the $\succeq$ relation. In this case, we could enforce the preservation of types while still allowing for sequential optimizations by using the following type compatibility relation (assuming a typing judgment of the form $\Gamma \vdash c : \mathsf{unit}$ is available):

$$c_1 \diamond c_2 \stackrel{\text{def}}{=} \forall \Gamma.\ (\Gamma \vdash c_1 : \mathsf{unit}) \text{ iff } (\Gamma \vdash c_2 : \mathsf{unit})$$

It ensures that both c_1 and c_2 are well-typed in the same typing environments. If we use the relation $(\succeq \cap \diamond)$ in our parameterized semantics, we could achieve type safety without having to worry about specific issues such as the legality of "out-of-thin-air" executions. That seems natural, as the preservation of types is a property of program transformations performed at the high-level, by source-to-source compilers, and also by compilers that use typed intermediate languages. Therefore, in our perception we should not prevent "out-of-thin-air" reads from happening; instead we just make sure that, if they happen, they will not break type safety.

5. Grainless Semantics and DRF Guarantee

Reynolds [2004] proposed trace-based grainless semantics to avoid specifying the default level of atomicity in concurrent languages. The semantics is based on three principles:

1. *Operations have duration and can overlap with one another during execution.*
2. *If two overlapping operations touch the same location, the meaning of the program execution is "wrong".*
3. *If, from a given starting state, execution of a program can give "wrong", then no other possibilities need to be considered.*

A different grainless semantics was proposed by Brookes [2006], based on "footstep traces". Following similar ideas, here we give a grainless semantics to our language, which is operational instead of being trace-based denotational semantics. The semantics permits only data-race-free programs to execute, therefore it gives us a simple and operational formulation of data-race-freedom and allows us to prove DRF-guarantee of our relaxed semantics.

$$\begin{array}{lll} (\textit{ThrdTree}) & \widetilde{T} & ::= (c, \delta) \mid \langle\!\langle \widetilde{T}, \widetilde{T} \rangle\!\rangle c \\ (\textit{ThrdCtxt}) & \widetilde{\mathbf{T}} & ::= [\,] \mid \langle\!\langle \widetilde{\mathbf{T}}, \widetilde{T} \rangle\!\rangle c \mid \langle\!\langle \widetilde{T}, \widetilde{\mathbf{T}} \rangle\!\rangle c \end{array}$$

Figure 10. Instrumented thread trees and contexts

$$\begin{array}{rcl} \delta \smile \delta' & \stackrel{\text{def}}{=} & \delta.ws \cap (\delta'.rs \cup \delta'.ws) = \emptyset \wedge \delta.rs \cap \delta'.ws = \emptyset \\ \mathsf{wft}(\widetilde{T}, \delta) & \stackrel{\text{def}}{=} & \forall c, c', \delta', \widetilde{\mathbf{T}}'. \\ & & (\widetilde{\mathbf{T}}[\,(c, \delta)\,] = \widetilde{\mathbf{T}}'[\,(c', \delta')\,]) \wedge (\widetilde{\mathbf{T}} \neq \widetilde{\mathbf{T}}') \rightarrow \delta \smile \delta' \\ \lfloor T \rfloor & \stackrel{\text{def}}{=} & \begin{cases} (c, \mathit{emp}) & T = c \\ \langle\!\langle \lfloor T_1 \rfloor, \lfloor T_2 \rfloor \rangle\!\rangle c & T = \langle\!\langle T_1, T_2 \rangle\!\rangle c \end{cases} \end{array}$$

Figure 11. Auxiliary definitions

5.1 Grainless semantics

We first instrument thread trees with footprints of threads, as shown in Fig. 10. The sets *rs* and *ws* in the footprint δ record the memory locations that are being read and written by the corresponding thread. Recall that we assume threads only share read-only variables, therefore accesses of variables would not cause races and we do not record variables in footprints. Execution contexts $\widetilde{\mathbf{T}}$ in the instrumented trees are defined similarly as $\mathbf{T}$ in Sec. 2.

The footprint δ associated with each leaf node on $\widetilde{T}$ records the memory locations that are being accessed by this thread. To ensure the data-race-freedom, the footprint δ of the active thread at the context $\widetilde{\mathbf{T}}$ must be disjoint with the footprints of other threads. This requirement is defined in Fig. 11 as the wft (well-formed tree) condition. We also define $\lfloor T \rfloor$ to convert T to an instrumented thread tree with an initial footprint *emp* for each thread.

The grainless semantics is shown in Fig. 12, which refers to the sequential transitions defined in Figs. 6 and 9. In this semantics we execute unordered commands in a big step, as shown in the first rule (see Fig. 9 for the definition of $\langle c, \sigma\rangle \Downarrow_\delta \langle c', \sigma'\rangle$). It cannot be interrupted by other threads, therefore the environment cannot observe transitions of *the smallest granularity*. The footprint δ of this big step is recorded on the thread tree at the end, which means the transition has *duration* and the memory locations in δ are still in use (even though the state is changed to σ'). So when other threads execute, they cannot assume this step has finished and cannot issue conflicting memory operations.

cons and **dispose** (the third rule), atomic blocks (the sixth rule) and thread fork/join (the last two rules) are all *atomic* instead of being grainless. Comparing with the first rule, we can see the footprint at the end of the step is *emp*, showing that this step finishes and the memory locations in δ are no longer in use. Note the *emp* footprint also clears the footprint of the preceding unordered transition of this thread, therefore these atomic operations also serve as memory barriers that mark the end of the preceding unordered commands. The footprint on the left hand side is not used in these rules, so we use _ to omit it.

In all these rules, we check the wft condition to ensure that each step does not issue memory operations that are in conflict with those ongoing ones made by other threads. If the check fails, we reach the special race configuration and the execution stops (the fourth and seventh rules).

The second rule shows that the intermediate footprint δ' may be recorded on the thread tree, even if the big step transition has not

$$\langle \widetilde{\mathbf{T}}[\,(c,\_)\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,(c',\delta)\,],\sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \Downarrow_\delta \langle c',\sigma'\rangle \text{ and } \mathsf{wft}(\widetilde{\mathbf{T}},\delta)$$

$$\langle \widetilde{\mathbf{T}}[\,(c,\delta)\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,(c,\delta')\,],\sigma\rangle \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta']{\mathbf{u}}{}^{*} \langle c',\sigma'\rangle,\ \delta\subset\delta', \text{ and } \mathsf{wft}(\widetilde{\mathbf{T}},\delta')$$

$$\langle \widetilde{\mathbf{T}}[\,(c,\_)\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,(c',\mathit{emp})\,],\sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \langle c',\sigma'\rangle \text{ and } \mathsf{wft}(\widetilde{\mathbf{T}},\delta)$$

$$\langle \widetilde{\mathbf{T}}[\,(c,\_)\,],\sigma\rangle \Longrightarrow \mathsf{race} \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*} \langle c',\sigma'\rangle \text{ or } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \langle c',\sigma'\rangle, \text{and } \neg\,\mathsf{wft}(\widetilde{\mathbf{T}},\delta)$$

$$\langle \widetilde{\mathbf{T}}[\,(c,\_)\,],\sigma\rangle \Longrightarrow \mathsf{abort} \quad \text{if } \langle c,\sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort} \text{ or } \langle c,\sigma\rangle \xrightarrow{\mathbf{o}} \mathsf{abort}$$

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,],\_)\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{skip}\,],\mathit{emp})\,],\sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{}{}^{*} \langle \mathbf{skip},\sigma'\rangle \text{ and } \mathsf{wft}(\widetilde{\mathbf{T}},\delta)$$

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,],\_)\,],\sigma\rangle \Longrightarrow \mathsf{race} \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{}{}^{*} \langle c',\sigma'\rangle \text{ and } \neg\mathsf{wft}(\widetilde{\mathbf{T}},\delta)$$

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,],\_)\,],\sigma\rangle \Longrightarrow \mathsf{abort} \quad \text{if } \langle c,\sigma\rangle \longrightarrow^{*} \mathsf{abort}$$

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,c_1 \parallel c_2\,],\_)\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,\langle\!\langle (c_1,\mathit{emp}),(c_2,\mathit{emp})\rangle\!\rangle \mathbf{E}[\,\mathbf{skip}\,]\,],\sigma\rangle$$

$$\langle \widetilde{\mathbf{T}}[\,\langle\!\langle (\mathbf{skip},\_),(\mathbf{skip},\_)\rangle\!\rangle c\,],\sigma\rangle \Longrightarrow \langle \widetilde{\mathbf{T}}[\,(c,\mathit{emp})\,],\sigma\rangle$$

Figure 12. Grainless semantics

finished. This is necessary to characterize the following program as one with data-races:

$$(\mathbf{while}\ \mathsf{true}\ \mathbf{do}\ [x] := 4) \ \parallel\ (\mathbf{while}\ \mathsf{true}\ \mathbf{do}\ [x] := 3)$$

This program would violate the side condition wft of this rule, although both threads diverges. Note that the rule does not change the command c and the state σ. If we ignore the footprint, it simply adds some stuttering steps in the semantics. The side condition $\delta \subset \delta'$ ensures that the stuttering steps are not inserted arbitrarily. Here δ is either an intermediate footprint accessed earlier during this big-step transition, or the footprint accessed by the preceding big-step transition of this thread. In the second case, the last step must be an atomic operation and δ must be *emp* (see the explanation of atomic operations below).

The next rule shows that **cons** and **dispose** are atomic instead of being grainless. Comparing with the first rule, we can see the footprint after the step is *emp*, showing that this step finishes and memory locations in δ' are no longer used. However, in the wft condition, we still use δ' instead of *emp* to ensure the data-race-freedom. The fourth rule says the program has a data-race if the wft condition is violated.

The first rule for atomic blocks is similar to the rule for **cons** and **dispose**. Since the new footprint recorded in the thread tree is *emp*, it shows that atomic blocks are indeed atomic. The rules for thread fork and join are similar to their counterparts in Fig. 7. The two thread operations are also atomic.

Following Reynolds' Principles 2 and 3, both abort and race are viewed as bad program configurations. Execution of a program stops when it reaches one of them. Here we distinguish race from abort to define data-race-freedom. A thread tree T is race-free if and only if its execution in the grainless semantics never leads to race. By this definition, programs that abort may still be race-free. This allows us to discuss about race-free but unsafe programs, as shown in Theorem 5.3. In the formal definition below, we use $\lfloor T \rfloor$ to convert T to an instrumented thread tree in the grainless semantics, which is defined in Fig. 11.

Definition 5.1. $\langle T,\sigma\rangle$ racefree iff $\neg(\langle\lfloor T\rfloor,\sigma\rangle \Longrightarrow^{*} \mathsf{race})$; T racefree iff, for all σ, $\langle T,\sigma\rangle$ racefree.

We know the example we show above is not race-free. Below we show some more examples.

Example 5.2. Given the following programs,

(1) $[x] := 3 \ \parallel\ [x] := 4$

(2) $[x] := 3 \ \parallel\ \mathbf{atomic}\ \{[x] := 4\}$

(3) $[x] := 3 \ \parallel\ \mathbf{atomic}\ \{\mathbf{while}\ \mathsf{true}\ \mathbf{do}\ [x] := 4\}$

(4) $\mathbf{atomic}\ \{[x] := 3\} \ \parallel\ \mathbf{atomic}\ \{[x] := 4\}$

we know (4) is race-free, but (1), (2) and (3) are not. □

5.2 DRF-guarantee of the relaxed semantics

We can now formulate and prove the DRF-guarantee of the relaxed semantics presented in Sec. 3.3 . Theorem 5.3 says a race-free program configuration $\langle c,\sigma\rangle$ has the same observable behavior in both the relaxed semantics and the interleaving semantics: if it aborts in one semantics, it aborts in the other; if it never aborts, it reaches the same set of final states in both settings.

Theorem 5.3 (DRF-guarantee)**.** If $\langle T,\sigma\rangle$ racefree, then

1. $[\succeq_{\mathbf{t}}]\ \langle T,\sigma\rangle \longmapsto^{*} \mathsf{abort}$ iff $\langle T,\sigma\rangle \longmapsto^{*} \mathsf{abort}$.
2. If $\neg(\langle T,\sigma\rangle \longmapsto^{*} \mathsf{abort})$, then $[\succeq_{\mathbf{t}}]\ \langle T,\sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\sigma'\rangle$ iff $\langle T,\sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\sigma'\rangle$.

Proof. The proof is trivial by applying Lemmas 5.5 and 5.6. □

Below we also show an interesting corollary. It says that, if $c_1 \preceq c_2$ and we put them in any context $\mathcal{C}$, then the behavior of $\mathcal{C}[\,c_1\,]$ in the interleaving semantics is subsumed by the behavior of $\mathcal{C}[\,c_2\,]$, as long as there are no data-races.

Corollary 5.4. If $c_1 \preceq c_2$, and $\langle \mathcal{C}[\,c_2\,],\sigma\rangle$ racefree, then

1. If $\langle \mathcal{C}[\,c_1\,],\sigma\rangle \longmapsto^{*} \mathsf{abort}$ then $\langle \mathcal{C}[\,c_2\,],\sigma\rangle \longmapsto^{*} \mathsf{abort}$.
2. If $\neg(\langle \mathcal{C}[\,c_2\,],\sigma\rangle \longmapsto^{*} \mathsf{abort})$, and $\langle \mathcal{C}[\,c_1\,],\sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\sigma'\rangle$, then $\langle \mathcal{C}[\,c_2\,],\sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\sigma'\rangle$.

Proof. The proof is trivial given Theorem 5.3 and Lemma 3.4. □

The proof of the DRF-guarantee depends on two important lemmas. Lemma 5.5 shows the equivalence between the interleaving semantics and the grainless semantics for race-free programs. Lemma 5.6 shows the equivalence between the grainless semantics and the relaxed semantics. Therefore, we can derive the DRF-guarantee using the grainless semantics as a bridge.

Lemma 5.5. If $\langle T,\sigma\rangle$ racefree, then

1. $\langle T,\sigma\rangle \longmapsto^{*} \mathsf{abort}$ iff $\langle\lfloor T\rfloor,\sigma\rangle \Longrightarrow^{*} \mathsf{abort}$.
2. $\langle T,\sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\sigma'\rangle$ iff $\langle\lfloor T\rfloor,\sigma\rangle \Longrightarrow^{*} \langle(\mathbf{skip},\_),\sigma'\rangle$;

Lemma 5.6. If $\langle T,\sigma\rangle$ racefree, then

1. $[\succeq_{\mathbf{t}}]\ \langle T,\sigma\rangle \longmapsto^{*} \mathsf{abort}$ iff $\langle\lfloor T\rfloor,\sigma\rangle \Longrightarrow^{*} \mathsf{abort}$.

2. if $\neg(\langle T, \sigma\rangle \longmapsto^* \mathsf{abort})$, then
$[\succeq_{\mathbf{t}}]\ \langle T, \sigma\rangle \longmapsto^* \langle \mathbf{skip}, \sigma'\rangle$ iff $\langle \lfloor T \rfloor, \sigma\rangle \Longrightarrow^* \langle (\mathbf{skip}, \_), \sigma'\rangle$.

To prove Lemma 5.5, we use the following two lemmas. Lemma 5.7 shows that an unordered operation can be reordered with other operations as long as they do not have data dependencies. Lemma 5.8 says the data-race-freedom is preserved by the interleaving semantics.

Lemma 5.7. If $\delta_1 \smile \delta_2$, and c_1 and c_2 only share read-only variables, then $(\exists\sigma'.\ \langle c_1, \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1', \sigma'\rangle \wedge \langle c_2, \sigma'\rangle \xrightarrow[\delta_2]{} \langle c_2', \sigma''\rangle)$ iff $(\exists\sigma'.\ \langle c_2, \sigma\rangle \xrightarrow[\delta_2]{} \langle c_2', \sigma'\rangle \wedge \langle c_1, \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1', \sigma''\rangle)$.

Lemma 5.8. If $\langle T, \sigma\rangle$ racefree and $\langle T, \sigma\rangle \longmapsto \langle T', \sigma'\rangle$, then $\langle T', \sigma'\rangle$ racefree.

Before proving Lemma 5.6, we first lift our $\preceq_{\mathbf{t}}$ relation for the grainless semantics.

Definition 5.9. We lift $\preceq_{\mathbf{t}}$ for instrumented thread trees.

1. $(c_1, \delta_1) \preceq_{\mathbf{t}} (c_2, \delta_2)$ iff $c_1 \preceq c_2$ and $\delta_1 \subseteq \delta_2$;
2. $\langle\!\langle \widetilde{T}_1', \widetilde{T}_1'' \rangle\!\rangle c_1 \preceq_{\mathbf{t}} \langle\!\langle \widetilde{T}_2', \widetilde{T}_2'' \rangle\!\rangle c_2$ iff $c_1 \preceq c_2$, $\widetilde{T}_1' \preceq_{\mathbf{t}} \widetilde{T}_2'$ and $\widetilde{T}_1'' \preceq_{\mathbf{t}} \widetilde{T}_2''$.

Note that here we only require the footprint in $\widetilde{T}_1$ is a subset of the corresponding footprint in $\widetilde{T}_2$. This is because $\widetilde{T}_1$ only accesses a subset of memory used by $\widetilde{T}_2$, as shown by the following lemma.

Lemma 5.10. If $c_1 \preceq c_2$ and $\langle c_1, \sigma\rangle \Downarrow_{\delta_1} \langle c_1', \sigma'\rangle$, then either $\langle c_2, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exist c_2' and δ_2 such that $\langle c_2, \sigma\rangle \Downarrow_{\delta_2} \langle c_2', \sigma'\rangle$, $\delta_1 \subseteq \delta_2$, and $c_1' \preceq c_2'$.

The proof of Lemma 5.6 uses Lemma 3.6 and Lemma 3.7. It is also based on the following lemma and corollary. Lemma 5.11 essentially says that, if $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$ and $\widetilde{T}_2$ does not race or abort, then they lead to the same state after each step in the grainless semantics. It can be derived from Lemma 5.10.

Lemma 5.11. If $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$, then

1. if $\langle \widetilde{T}_1, \sigma\rangle \Longrightarrow \mathsf{abort}$ or race, then $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow \mathsf{abort}$ or race;
2. if $\langle \widetilde{T}_1, \sigma\rangle \Longrightarrow \langle \widetilde{T}_1', \sigma'\rangle$, then $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow \mathsf{abort}$ or race, or there exists $\widetilde{T}_2'$, such that $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow \langle \widetilde{T}_2', \sigma'\rangle$ and $\widetilde{T}_1' \preceq_{\mathbf{t}} \widetilde{T}_2'$.

Corollary 5.12. If $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$, then

1. if $\langle \widetilde{T}_1, \sigma\rangle \Longrightarrow^* \mathsf{abort}$ or race, then $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow^* \mathsf{abort}$ or race;
2. if $\langle \widetilde{T}_1, \sigma\rangle \Longrightarrow^* \langle (\mathbf{skip}, \delta), \sigma'\rangle$, then $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow^* \mathsf{abort}$ or race, or $\langle \widetilde{T}_2, \sigma\rangle \Longrightarrow^* \langle (\mathbf{skip}, \_), \sigma'\rangle$.

Here we only show the key lemmas used to prove the DRF-guarantee.

6. Soundness of CSL

We prove the soundness of CSL in our relaxed semantics by first proving it is sound in the grainless semantics. The CSL we use here is mostly standard [O'Hearn 2007, Brookes 2007]. To make the paper self-contained, we show the assertions and their semantics in Fig. 13, and some selected logic rules in Fig 14. Below we give a brief overview of the logic.

The logic consists of sequential and concurrent rules. The first four rules in Fig. 14 are sequential (more rules are omitted here). They are just standard sequential separation logic rules [Ishtiaq and O'Hearn 2001, Reynolds 2002]. The semantics is standard and is defined below. Soundness of the rules is shown by Lemma 6.2.

(*Assertion*) $p, q, r, I ::= b \mid \mathsf{emp} \mid e_1 \mapsto e_2 \mid p * q \mid p \Rightarrow q \mid \forall x.p \mid \ldots$

$(h, s) \models b$	iff	$[\![b]\!]_s = \mathbf{true}$
$(h, s) \models \mathsf{emp}$	iff	$\mathsf{dom}(h) = \emptyset$
$(h, s) \models e_1 \mapsto e_2$	iff	$\mathsf{dom}(h) = \{[\![e_1]\!]_s\}$ and $h([\![e_1]\!]_s) = [\![e_2]\!]_s$
$(h, s) \models p * q$	iff	there exist h_1 and h_2 such that $h = h_1 \uplus h_2$, $(h_1, s) \models p$ and $(h_2, s) \models q$ where $\uplus$ means the union of two heaps with disjoint domains

$\ldots$

Figure 13. CSL assertions and their semantics

Definition 6.1. $\models \{p\}c\{q\}$ iff, for all σ such that $\sigma \models p$, $\neg(\langle c, \sigma\rangle \longrightarrow^* \mathsf{abort})$, and, if $\langle c, \sigma\rangle \xrightarrow[\delta]{}{}^* \langle \mathbf{skip}, \sigma'\rangle$, then $\sigma' \models q$.

Lemma 6.2. If $\vdash \{p\}\, c\, \{q\}$ then $\models \{p\}c\{q\}$.

The judgment $I \vdash \{p\}\, c\, \{q\}$ for concurrent rules informally says that the state can be split implicitly into a shared part and a local part; the local part can be accessed only by c; p and q are pre- and post-conditions for the local state; the shared part can be accessed by both c and its environment, but only in atomic blocks; accesses of the shared state must preserve its invariant I.

We do not explain details of the rules. The LOCALR rule is similar to the local resource rule by Brookes [2007]. This rule and the FRAME-I rule are due to Parkinson et al. [2007]. They also show that the standard frame rule (over local resources) can be derived from the two rules. Here we implicitly require that I be precise, i.e. for any state there is at most one sub-state satisfying I.

To prove the soundness of CSL, we first formulate in Definition 6.5 the program invariant enforced by the logic rules. Some auxiliary constructs used in the formulation are defined in Definitions 6.3 and 6.4. Here $\sigma \parallel_{(I,X)} \sigma'$ means the difference between σ and σ' must be within the variable set X and the shared sub-states specified by I. $\sigma \models \delta \uplus I$ says that the footprint δ is a subset of the heap in σ and it has no overlap with the shared part specified by I (therefore δ belongs to the local state).

Definition 6.3. $(h, s) \parallel_{(I,X)} (h', s')$ iff there exist h_1, h_1' and h_2 such that $h = h_1 \uplus h_2$, $h' = h_1' \uplus h_2$, $(h_1, s) \models I$, $(h_1', s') \models I$, and $\forall x \notin X.\ s(x) = s'(x)$.

Definition 6.4. $(h, s) \models \delta \uplus I$ iff, for all h_1 and h_2, if $h_1 \uplus h_2 = h$ and $(h_1, s) \models I$, then $(\delta.rs \cup \delta.ws) \subseteq \mathsf{dom}(h_2)$.

Definition 6.5.
$I \models \langle \widetilde{T}, \sigma\rangle \rhd_0 q$ always holds. $I \models \langle \widetilde{T}, \sigma\rangle \rhd_{k+1} q$ holds iff the following are true:

1. $\sigma \models I * \mathsf{true}$;
2. $\neg(\langle \widetilde{T}, \sigma\rangle \Longrightarrow \mathsf{abort})$ and $\neg(\langle \widetilde{T}, \sigma\rangle \Longrightarrow \mathsf{race})$;
3. if $\widetilde{T} = (\mathbf{skip}, \delta)$, then $\sigma \models I * q$;
4. for all $\widetilde{\mathbf{T}}$, c and δ, if $\widetilde{T} = \widetilde{\mathbf{T}}[\,(c, \delta)\,]$, then $\sigma \models \delta \uplus I$;
5. if $\langle \widetilde{T}, \sigma\rangle \Longrightarrow \langle \widetilde{T}', \sigma'\rangle$, then $\forall j \leq k.\ I \models \langle \widetilde{T}', \sigma'\rangle \rhd_j q$;
6. if X does not contain free variables in $\widetilde{T}$ and q, and $\sigma \parallel_{(I,X)} \sigma'$, then $\forall j \leq k.\ I \models \langle \widetilde{T}, \sigma'\rangle \rhd_j q$.

$I \models \langle \widetilde{T}, \sigma\rangle \rhd q$ if and only if $\forall k.\ I \models \langle \widetilde{T}, \sigma\rangle \rhd_k q$.

Informally, $I \models \langle \widetilde{T}, \sigma\rangle \rhd_k q$ requires that I holds over a sub-heap of σ; the next step would not race or abort; $I * q$ holds if we are at the end of execution; the footprint of each thread is part of σ but has no overlap with I; and all these invariants are preserved up to k steps made by either $\widetilde{T}$ itself or by its environment. The

$$\frac{}{\vdash \{e_1 \mapsto \_\}\ [e_1] := e_2\ \{e_1 \mapsto e_2\}}\ (\text{ST}) \qquad \frac{}{\vdash \{e \mapsto \_\}\ \mathbf{dispose}(e)\ \{\mathsf{emp}\}}\ (\text{DISPOSE}) \qquad \frac{\vdash \{p\}\,c\,\{q\}}{\vdash \{p * r\}\,c\,\{q * r\}}\ (\text{FRM-S})$$

$$\frac{}{\vdash \{x = x' \wedge \mathsf{emp}\}\ x := \mathbf{cons}(e_1, \ldots, e_n)\ \{(x \mapsto [x'/x]e_1) * \cdots * (x{+}k{-}1 \mapsto [x'/x]e_k)\}}\ (\text{CONS})$$

$$\frac{\vdash \{p\}\,c\,\{q\}}{I \vdash \{p\}\,c\,\{q\}}\ (\text{ENV}) \qquad \frac{\vdash \{p * I\}\,c\,\{q * I\}}{I \vdash \{p\}\ \mathbf{atomic}\ c\,\{q\}}\ (\text{ATOM}) \qquad \frac{I \vdash \{p_1\}\,c_1\,\{q_1\} \quad I \vdash \{p_2\}\,c_2\,\{q_2\}}{I \vdash \{p_1 * p_2\}\,c_1 \,\|\, c_2\,\{q_1 * q_2\}}\ (\text{PAR})$$

where c_2 does not update free var. in p_1, c_1 and q_1, and conversely.

$$\frac{I \vdash \{p\}\,c_1\,\{r\} \quad I \vdash \{r\}\,c_2\,\{q\}}{I \vdash \{p\}\,c_1; c_2\,\{q\}}\ (\text{SEQ}) \qquad \frac{I * I' \vdash \{p\}\,c\,\{q\}}{I \vdash \{p * I'\}\,c\,\{q * I'\}}\ (\text{LOCALR}) \qquad \frac{I \vdash \{p\}\,c\,\{q\}}{I * I' \vdash \{p\}\,c\,\{q\}}\ (\text{FRAME-I})$$

Figure 14. Selected CSL Rules

following lemma shows that $I \vdash \{p\}\,c\,\{q\}$ indeed ensures the invariant $I \models \langle (c, \delta),\ \sigma\rangle \rhd q$, as long as σ satisfies the precondition and δ is part of the initial local state.

Lemma 6.6. If $I \vdash \{p\}\,c\,\{q\}$, $\sigma \models I * p$, and $\sigma \models \delta \uplus I$, then $I \models \langle (c, \delta),\ \sigma\rangle \rhd q$.

The proof of this lemma follows standard techniques, i.e. we need to first prove the locality [Yang and O'Hearn 2002, Calcagno et al. 2007] of each primitive commands. Details of the proofs are shown in Appendix D.

We define semantics of the judgment $I \models \{p\}c\{q\}$ below, based on the grainless semantics. The soundness of CSL rules is shown by Lemma 6.8. The proof trivially follows from Lemma 6.6.

Definition 6.7. $I \models \{p\}c\{q\}$ iff, for all σ and δ such that $\sigma \models I * p$ and $\sigma \models \delta \uplus I$, we have $\neg\,(\langle (c, \delta),\ \sigma\rangle \Longrightarrow^* \mathsf{abort})$, $\neg\,(\langle (c, \delta),\ \sigma\rangle \Longrightarrow^* \mathsf{race})$, and, if $\langle (c, \delta),\ \sigma\rangle \Longrightarrow^* \langle (\mathbf{skip}, \_),\ \sigma'\rangle$, then $\sigma' \models I * q$.

Lemma 6.8. If $I \vdash \{p\}\,c\,\{q\}$, then $I \models \{p\}c\{q\}$.

Finally we give semantics to $I \vdash \{p\}\,c\,\{q\}$ based on our relaxed semantics, and show the soundness in Theorem 6.10.

Definition 6.9. $I \models_{[\Lambda]} \{p\}c\{q\}$ iff, for all σ such that $\sigma \models I * p$, $\neg\,([\Lambda]\ \langle c,\ \sigma\rangle \longmapsto^* \mathsf{abort})$, and, if $[\Lambda]\ \langle c,\ \sigma\rangle \longmapsto^* \langle \mathbf{skip},\ \sigma'\rangle$, then $\sigma' \models I * q$.

Theorem 6.10. If $I \vdash \{p\}\,c\,\{q\}$, then $I \models_{[\succeq_{\mathtt{t}}]} \{p\}c\{q\}$.

Proof. Trivial by applying Lemma 5.6. □

Extensions of CSL. The original CSL [O'Hearn 2007] only supports a coarse classification of resource ownership. It does not support simultaneous read by different threads. Bornat et al. [2005] extended CSL with fractional permissions [Boyland 2003] to distinguish exclusive total accesses (read, write and disposal) and shared read-only accesses.

$$\begin{array}{lll} (\textit{Perm}) & \pi & \in\ (0, 1] \\ (\textit{Assertion}) & p, q, r, I & ::= \ \ldots \mid e_1 \overset{\pi}{\mapsto} e_2 \mid \ldots \end{array}$$

The permission π in the new assertion $e_1 \overset{\pi}{\mapsto} e_2$ is a rational number. $\pi = 1$ means total access; $0 < \pi < 1$ means shared access. The original assertion $e_1 \mapsto e_2$ can be viewed as a shorthand notation for $e_1 \overset{1}{\mapsto} e_2$.

We can prove that CSL with fractional permissions is also sound with respect to the grainless semantics, but the model of heaps needs to be changed to a partial mapping from locations to a pair of values and permissions. We also need to refine our Definition 6.4 and require that $\delta.ws$ belong to a subset of h_2 that contain only permissions for total accesses. The proof should be similar to the proof for standard CSL.

Since our grainless semantics is a mix of big-step and small-step semantics, and there is no interleaving between threads when unordered commands are executed, intuitively proving the soundness of CSL-family logics could only be simpler in this semantics than in the interleaving semantics. Therefore we are confident that other extensions of CSL, such as the support of storable locks [Gotsman et al. 2007, Hobor et al. 2008] and the combinations of CSL with Rely-Guarantee reasoning [Vafeiadis and Parkinson 2007, Feng 2009], can also be proved sound with respect to the grainless semantics. Then their soundness in our relaxed semantics can be easily derived from Lemma 5.6. We would like to prove our hypothesis in our future work.

7. Discussions and Related Work

Relaxed memory models. The literature on memory models is vast. We cannot give a detailed overview due to space constraints. Below we just discuss some closely related work.

The RAO model by Saraswat et al. [2007] consists of a family of transformations (IM, CO, AU, LI, PR and DX). Unlike our subsumption relation which gives only an abstract and extensional formulation of semantics preservation between sequential threads, each of them defines a very specific class of transformations. We suspect that our model is weaker (not necessarily strictly weaker) than the RAO model. IM, CO and DX are obvious specializations of our subsumption relation with extra constraints. Although we only support intra-thread local transformations, we can define a more relaxed version of PR: $c \succeq$ **if** q **then** c' **else** c, assuming c' has the same behaviors with c if q holds over the initial state. AU enforces a specific scheduling. We allow all possible scheduling in our relaxed semantics. LI is an inter-thread transformation. It is unclear how it relates to our subsumption relation, but the examples [Saraswat et al. 2007] involving LI (e.g., the cross-over example) can be supported following the pattern with which we reproduce the prescient-write example in Sec. 4.

In this paper, we do not investigate the precise connection to the Java Memory Model (JMM). For the moment, we assume that the work by Saraswat et al. [2007] is consistent with JMM. Our semantics is operational and not based upon the happens-before model. We believe it provides a weaker memory model with the DRF-guarantee, and supports compiler optimizations that JMM does not, such as the one described by Cenciarelli et al. [2007]. However, there are two key issues if we want to apply our model to Java, i.e. preventing the "out-of-thin-air" behaviors and supporting partial barriers. The first one can be addressed by adding constraints similar to Saraswat's DX-family transformations in our subsumption relation. The second one can be solved by allowing transformations to go beyond partial barriers. We will show the solution in an upcoming paper.

Boudol and Petri [2009] presented an operational approach to relaxed memory models. Their weak semantics made explicit use of write buffers to simulate the effects of memory caching during execution, which was more concrete and constructive than most memory model descriptions. However, only a restricted set of reorderings was observable in their semantics, while our semantics is much weaker and supports all four types of memory reordering. Also, since our formalization of memory models is based on program transformations, our semantics has better support of compiler optimizations. The connection between their semantics and program logics such as CSL is unclear either.

Sevcik [2008] analyzed the impact of common optimizations in two relaxed memory models, establishing their validity and showing counter examples; some of our examples were inspired by his work. Gao and Sarkar [2000] introduced Location Consistency (LC), probably the weakest memory model described in the literature; we stand by their view that memory models should be more relaxed and not based necessarily on cache consistence.

Grainless semantics. Besides being operational instead of trace-based semantics, there are some other differences between our grainless semantics and semantics by Reynolds [2004] and Brookes [2006]. Each operation in Reynolds' semantics is modeled as a pair of actions labeled with "**start**" and "**fin**" respectively, but actions are not coalesced. We model duration of operations by recording their footprints on the thread trees. We also coalesce non-atomic operations into a "big-step" transition, which is similar to Brookes' semantics. On the other hand, we do not coalesce operations from different threads, as Brookes did in his semantics.

Reynolds did not discuss about memory allocation and disposal operations. Brookes treated them as non-atomic operations. We treat them as atomic, otherwise the following program may generate a race in our semantics:

$$x := \mathbf{cons}(3) \parallel \mathbf{dispose}(y)$$

The two operations may have the same footprint if **dispose** is executed first and then the memory location is recycled and assigned to x by **cons**. Although it is possible to relax this atomicity requirement by making the footprint of **cons** to be *emp* (since **cons** always generates fresh memory locations), doing this would make our proof of the DRF-guarantee much harder because Lemma 5.7 is broken in the above scenario. As discussed in Sec. 2, we believe the decision is reasonable because **cons** and **dispose** do share resources and need to be synchronized in their real-world implementations. Even viewed abstractly, they share the set of fresh locations. Also note that treating them as built-in atomic operations does not affect the soundness of CSL. Like other non-atomic operations, they can be executed either inside or outside of atomic blocks.

Oracle semantics for CSL. In their oracle semantics, Hobor et al. [2008] gave a coroutine interleaving model for concurrent programs, in which context switching only occur at concurrent operations. This is similar to our grainless semantics. Both their semantics and our grainless semantics permit only race-free programs to execute, but this is enforced in different ways. We require that the footstep of each thread is compatible with other threads' at each step of execution; while the oracle semantics maintains disjoint local worlds for each thread and ensures that thread-local operations can only occur in a local world. The goal of their work was to give an operational semantics that bridges sequential optimizations (and relaxed memory models) with CSL-verified concurrent programs that is guaranteed to be race-free, which is similar to the goal of our paper. However, there was no formalization of memory models and optimizations, so the claim was not formally proved.

8. Conclusions

We present a simple operational semantics to formalize memory models. The semantics is parameterized on a binary relation over programs. By instantiating the parameter with a specific relation $\succeq_{\mathbf{t}}$, we have obtained a memory model that is weaker than many existing ones. Since the relation is weaker than observational equivalence of sequential programs, this memory model also captures many sequential optimizations that usually preserve semantic equivalence. We then propose an operational grainless semantics, which allows us to define data-race-freedom and prove the DRF-guarantee of our relaxed memory model. We also proved the soundness of CSL in relaxed memory models, using the grainless semantics as a bridge between CSL and the relaxed semantics.

In our future work, we would like to extend our framework to support partial barriers. This can be achieved by extend the $\succeq$ relation with transformations that go beyond partial barriers. It is also interesting to formally verify the correctness of sequential optimization algorithms in a concurrent setting. Given this framework, it is sufficient to prove that the algorithms implement a subset of the $\succeq$ relation. As mentioned before, we also want to apply this approach to languages with other important language features, such as function calls, dynamic locks, and dynamic thread creations.

References

S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

S. Adve and M. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6): 613–624, Jun. 1993.

S. Adve and M. Hill. Weak ordering — a new definition. In *17th ISCA*, pages 2–14, Seattle, Washington, May 1990.

N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *31st POPL*, pages 14–25, Jan. 2004.

H. Boehm and S. Adve. The foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, Tucson, Arizona, Jun. 2008.

H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI*, pages 261–268, Chicago, Jun. 2005.

R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pages 259–270, Jan. 2005.

G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *36th POPL*, pages 392–403, Savannah, Georgia, USA, Jan. 2009.

J. Boyland. Checking interference with fractional permissions. In *10th International Symposium on Static Analysis*, pages 55–72, 2003.

S. Brookes. A semantics for concurrent separation logic. *Theoretical Comp. Sci.*, 375(1–3):227–270, May 2007.

S. Brookes. A grainless semantics for parallel programs with shared mutable data. *Electronic Notes in Theoretical Computer Science*, 155:277–307, May 2006.

C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd LICS*, pages 366–378, July 2007.

P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, pages 331–346, Mar. 2007.

E. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.

M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th ISCA*, pages 434–442, Tokyo, Jun. 1986.

X. Feng. Local rely-guarantee reasoning. In *36th POPL*, pages 315–327, Jan. 2009.

X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.

G. Gao and V. Sarkar. Location consistency – a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8): 798–813, Aug. 2000.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH News*, 18(3), Jun. 1990.

J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Committee, Mar. 1989.

A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *5th APLAS*, pages 19–37, Dec. 2007.

A. Hobor, A. Appel, and F. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, pages 353–367, Mar. 2008.

S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, Jan. 2001.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), Sep. 1979.

X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *33rd POPL*, pages 42–54, Jan. 2006.

J. Manson, W. Pugh, and S. Adve. The Java memory model. In *32nd POPL*, pages 378–391, Long Beach, California, Jan. 2005.

D. Mosberger. Memory consistency models. *Operating Systems Review*, 27 (1):18–26, Jan. 1993.

P. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Comp. Sci.*, 375(1–3):271–307, May 2007.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *22nd TPHOLS*, pages 391–407, Munich, Germany, Aug. 2009.

M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *34th POPL*, pages 297–302, Nice, France, Jan. 2007.

J. Reynolds. Towards a grainless semantics for shared-variable concurrency. In *FSTTCS*, pages 37–48, Chennai, India, Dec. 2004.

J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, Copenhagen, Jul. 2002.

V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *12th PPoPP*, San Jose, Mar. 2007.

J. Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, School of Informatics, University of Edinburgh, 2008.

The SPARC Architecture Manual, Version 8. Revision SAV080SI9308. SPARC International Inc, 1992.

V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, Lisbon, Sep. 2007.

H. Yang. Relational separation logic. *Theoretical Computer Science*, 375 (1-3):308–334, 2007.

H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, pages 402–416, Grenoble, France, Apr. 2002.

A. Total/Partial Store Order

We give another non-trivial instantiation of Λ in our parameterized semantics, which yields the Total Store Ordering (TSO) model implemented by the SPARCv8 architecture [SPA 1992]. TSO allows write-to-read reordering. It enforces cache-coherence, but allows a thread to read its own writes earlier.

We define $\succeq_{\text{TSO}}$, an instantiation of Λ, in Fig. 15. The first rule shows the reordering of a write with a subsequent read. The **else** branch shows the reordering when there is no data dependency. The **then** branch allows a thread to read its own write earlier. Here $fv(e)$ is the set of free variables in e. The other rules (except the last one) show how to propagate the reordering to the subsequent code. Remember that the transformation may occur at any step during the execution in our parameterized semantics, so we only need to consider the statements starting with a write operation, and the write might be postponed indefinitely until an ordered operation is reached.

In real architectures, the reordering is caused by write buffering instead of swapping the two instructions. We do not model the write buffer here since our goal is not to faithfully model what happens in hardware. Instead, we just want to give an extensional model for programmers. To see the adequacy of our rules, we can view the right hand side of the first rule as a simplification of the following code, which simulates the write buffering [Owens et al. 2009] more directly:

local *tmp*, *buf* **in**
  tmp := e_1; *buf* := e';
  if *tmp* = e_2 **then** x := *buf* **else** $x := e_2$;
  [*tmp*] := *buf*
end

Here the local variable *buf* can be viewed as a write buffer. Also note that the side condition of this rule can be eliminated if we also simulate the hardware support of register renaming (like our use of *tmp* above).

Remark A.1. The $\succeq_{\text{TSO}}$ relation is a subset of the $\succeq$ relation

Figure 16 presents the $\succeq_{\text{PSO}}$ relation. It builds upon the $\succeq_{\text{TSO}}$ relation, but with extra rules to allow write-to-write reorderings.

Remark A.2. The $\succeq_{\text{PSO}}$ relation is a subset of the $\succeq$ relation

B. Proving Properties of Subsumption

Lemma B.1. If $c_1 \preceq c_2$ and $\langle c_1, \sigma\rangle \Downarrow \langle c_1', \sigma'\rangle$, then either $\langle c_2, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c_2' such that $\langle c_2, \sigma\rangle \Downarrow \langle c_2', \sigma'\rangle$ and $c_1' \preceq c_2'$.

Proof. From $c_1 \preceq c_2$, we can assume $c_1 \preceq_1 c_2$. From condition 2 of Definition **??** we know that either $\langle c_2, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ (left hand side of the goal) or there exists c_2' such that $\langle c_2, \sigma\rangle \Downarrow \langle c_2', \sigma'\rangle$ and constraints (a) to (d) hold for c_2' and index 0. If we instantiate the existential in the right hand side of the goal using c_2', remains to show $c_1' \preceq c_2'$, i.e. for all k, $c_1' \preceq_k c_2'$. If $k = 0$, that is trivial. When $k > 0$, again, from $c_1 \preceq c_2$, we can assume $c_1 \preceq_k c_2$. From condition 2 of Definition **??** we know that either $\langle c_2, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c_2'' such that $\langle c_2, \sigma\rangle \Downarrow \langle c_2'', \sigma'\rangle$ and constraints (a) to (d) hold for c_2'' and all index j, $j \leq k-1$. Given that $\Downarrow$ is deterministic, we have $c_2' = c_2''$. Since $k > 0$ We unfold the definition of $c_1' \preceq_k c_2'$ in the goal; we need to show that for all $j \leq k-1$ both conditions of Definition **??** hold. Condition 1, holds trivially given that $\langle c_1', \sigma'\rangle$ is the final state of a big step and cannot abort. Condition 2, holds directly from the fact that only the current configuration $\langle c_1', \sigma'\rangle$ can be the final configuration of a big step; therefore we instantiate the c_2' in the goal with the current c_2' (and we know that if can perform a big step to itself since it is the target of a big step) and we can use constraints (a) to (d) for c_2' and index j to conclude. □

Lemma B.2. The relation $\preceq$ is reflexive.

Proof. Reflexitivity means for all c, $c \preceq c$. From $c \preceq c$, by definition, we need to show that for all k, $c \preceq_k c$. We perform strong induction over k. Base case holds trivially. In the inductive case, condition 1 of Definition **??** holds trivially. From condition 2, we assume $\langle c, \sigma\rangle \Downarrow \langle c', \sigma'\rangle$, and we show the right hand side case by instantiating c' with current c', and using the induction hypothesis to establish that $c' \preceq_j c'$ for all $j \leq k-1$. □

Corollary B.3. The relation $\preceq_{\mathbf{t}}$ is reflexive.

Lemma B.4. The relation $\preceq$ is transitive.

$$\mathbf{E}[\,[e_1]:=e';x:=[e_2]\,] \succeq_{\text{TSO}} \mathbf{E}[\begin{pmatrix}\textbf{if } (e_1{=}e_2)\\ \textbf{then } (x:=e';\,[e_1]:=x)\\ \textbf{else } (x:=[e_2];\,[e_1]:=e')\end{pmatrix}] \qquad \text{if } x \notin fv(e_1)\cup fv(e')$$

$$\mathbf{E}[\,[e]:=e'_1;x:=e'_2\,] \succeq_{\text{TSO}} \mathbf{E}[x:=e'_2;\,[e]:=e'_1\,] \qquad \text{if } x \notin fv(e)\cup fv(e'_1)$$

$$\mathbf{E}[\,[e]:=e';\textbf{skip}\,] \succeq_{\text{TSO}} \mathbf{E}[\textbf{skip};\,[e]:=e'\,] \qquad \text{always}$$

$$\mathbf{E}[\,[e_1]:=e'_1;\,[e_2]:=e'_2\,] \succeq_{\text{TSO}} c' \qquad \text{if } \exists c''.\ \mathbf{E}[\,[e_2]:=e'_2\,] \succeq_{\text{TSO}} c'' \wedge\ ([e_1]:=e'_1;c'') \succeq_{\text{TSO}} c'$$

$$\mathbf{E}[\begin{pmatrix}[e]:=e';\\ \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\end{pmatrix}] \succeq_{\text{TSO}} \mathbf{E}[\begin{pmatrix}\textbf{if } b \textbf{ then } ([e]:=e';\ c_1)\\ \quad\textbf{else } ([e]:=e';\ c_2)\end{pmatrix}] \qquad \text{always}$$

$$\mathbf{E}[\,[e]:=e';\textbf{while } b \textbf{ do } c\,] \succeq_{\text{TSO}} \mathbf{E}[\begin{pmatrix}\textbf{if } b\\ \textbf{then } ([e]:=e';c;\textbf{while } b \textbf{ do } c)\\ \textbf{else } [e]:=e'\end{pmatrix}] \qquad \text{always}$$

$$c \succeq_{\text{TSO}} c \qquad \text{always}$$

Figure 15. TSO

$$c \succeq_{\text{PSO}} c' \qquad \text{if } c \succeq_{\text{TSO}} c'$$

$$\mathbf{E}[\,[e_1]:=e'_1;\,[e_2]:=e'_2\,] \succeq_{\text{PSO}} c' \qquad \text{if } \exists c''.\ \mathbf{E}[\,[e_2]:=e'_2\,] \succeq_{\text{PSO}} c'' \wedge ([e_1]:=e'_1;c'') \succeq_{\text{PSO}} c'$$

$$\mathbf{E}[\,[e_1]:=e'_1;\,[e_2]:=e'_2\,] \succeq_{\text{PSO}} \mathbf{E}[\begin{pmatrix}\textbf{if } (e_1{=}e_2)\\ \textbf{then } ([e_1]:=e'_1;\,[e_1]:=e'_2)\\ \textbf{else } ([e_2]:=e'_2;\,[e_1]:=e'_1)\end{pmatrix}] \qquad \text{always}$$

Figure 16. PSO

Proof. Transitivity means for all c_1, c_2, and c_3, if $c_1 \preceq c_2$, and $c_2 \preceq c_3$, then $c_1 \preceq c_3$. From $c_1 \preceq c_3$, by definition, we need to show that for all k, $c_1 \preceq_k c_3$. We assume $c_1 \preceq_k c_2$, and $c_2 \preceq_k c_3$. We perform induction over k. Base case holds trivially. In the inductive case, we unfold the definition of $c_1 \preceq_k c_2$ in the goal. Then, for all $j \leq k-1$, we need to show conditions 1 and 2 of Definition **??**. For condition 1, assuming $\langle c_1,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ we need to show $\langle c_3,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$. From $c_1 \preceq_k c_2$, and $\langle c_1,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, we know that $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$. From $c_2 \preceq_k c_3$, and $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, we know that $\langle c_3,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$. For condition 2, assuming $\langle c_1,\, \sigma\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$ we need to show that either $\langle c_3,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c'_3 such that $\langle c_3,\, \sigma\rangle \Downarrow \langle c'_3,\, \sigma'\rangle$ and conditions (a) to (d) hold for index j. From $c_1 \preceq_k c_2$, and $\langle c_1,\, \sigma\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$, we know that either $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c'_2 such that $\langle c_2,\, \sigma\rangle \Downarrow \langle c'_2,\, \sigma'\rangle$ and conditions (a) to (d) hold for index j. In the first case, from $c_2 \preceq_k c_3$, and $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, we can conclude that $\langle c_3,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$. In the second case, from $c_2 \preceq_k c_3$, and $\langle c_2,\, \sigma\rangle \Downarrow \langle c'_2,\, \sigma'\rangle$, we know that either $\langle c_3,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c'_3 such that $\langle c_3,\, \sigma\rangle \Downarrow \langle c'_3,\, \sigma'\rangle$ and conditions (a) to (d) hold for index j; which was our goal. □

Corollary B.5. The relation $\preceq_{\mathbf{t}}$ is transitive.

Lemma B.6. If $c_1 \preceq c_2$, then, for all contexts $\mathcal{C}$, $\mathcal{C}[\,c_1\,] \preceq \mathcal{C}[\,c_2\,]$.

Proof. By structural induction over context $\mathcal{C}$, then we have to consider the following cases:

- If $c_1 \preceq c_2$, then $c_1; c \preceq c_2; c$ From $c_1; c \preceq c_2; c$, by definition, we need to show that for all k, $c_1; c \preceq_k c_2; c$. If $k = 0$ that is trivial. If $k > 0$, then we unfold Definition **??** in the goal and we need to show that both conditions 1 and 2 hold for all $j \leq k-1$. For condition 1, we assume $\langle c_1; c,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, from that we can derive $\langle c_1,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, therefore using $c_1 \preceq c_2$ we know that $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$, and we can establish that $\langle c_2; c,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$. For condition 2, we assume $\langle c_1; c,\, \sigma\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$ and we need to show that either $\langle c_2; c,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c'_2 such that $\langle c_2; c,\, \sigma\rangle \Downarrow \langle c'_2,\, \sigma'\rangle$ and conditions (a) to (d) hold for c'_2 and index j. From $\langle c_1; c,\, \sigma\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$ we know that either $\langle c_1,\, \sigma\rangle \Downarrow \langle \textbf{skip},\, \sigma''\rangle$ and $\langle c,\, \sigma''\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$ or there exists c''_1 such that $\langle c_1,\, \sigma\rangle \Downarrow \langle c''_1,\, \sigma'\rangle$ and $c'_1 = c''_1; c$. In both cases, from condition 2 of $c_1 \preceq c_2$, we know that either $\langle c_2,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ or there exists c'_2 such that $\langle c_2,\, \sigma\rangle \Downarrow \langle c'_2,\, \sigma'''\rangle$ and conditions (a) to (d) hold for c'_2 and index j. In the case that c_2 aborts, we can establish that $\langle c_2; c,\, \sigma\rangle \xrightarrow{\mathbf{u}}{}^* \mathsf{abort}$ and conclude. Otherwise, considering $\langle c_1,\, \sigma\rangle \Downarrow \langle \textbf{skip},\, \sigma''\rangle$ we know then that $c'_2 = \textbf{skip}$ and $\sigma''' = \sigma''$, composing that with $\langle c,\, \sigma''\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$ we have $\langle c_2; c,\, \sigma\rangle \Downarrow \langle c'_1,\, \sigma'\rangle$, we instantiate the c'_2 in the goal with c'_1 and conditions (a) to (d) hold trivially using Lemma B.2. Considering the case where $\langle c_1,\, \sigma\rangle \Downarrow \langle c''_1,\, \sigma'\rangle$ and $c'_1 = c''_1; c$, we know that $c''_1 \preceq_j c'_2$ and $\sigma''' = \sigma'$. We can then apply the induction hypothesis to establish $c''_1; c \preceq_j c'_2; c$ according to constraints (b) to (d) using Lemma B.2 when necessary.
- If $c_1 \preceq c_2$, then $c; c_1 \preceq c; c_2$ Similar to the previous case.
- If $c_1 \preceq c_2$, then **if** b **then** c_1 **else** $c \preceq$ **if** b **then** c_2 **else** c Follows the closely the definition of $c_1 \preceq c_2$ for states where condition b hold. Trivial otherwise.
- If $c_1 \preceq c_2$, then **if** b **then** c **else** $c_1 \preceq$ **if** b **then** c **else** c_2. Similar to the previous case.
- If $c_1 \preceq c_2$, then **while** b **do** $c_1 \preceq$ **while** b **do** c_2. We extract the index k from the goal. We do induction over k. The base case holds trivially. In the inductive case we unfold the Definition **??**.

For condition 1, we do strong induction over the number of steps taken by **while** b **do** c_1 to reach abort. The base case is not possible as we can always unfold the loop. In the inductive case, we know that we can unfold the loop and from there we reach an abort state. We know condition b holds for current σ because otherwise we leave the loop and never abort. therefore we know that $\langle c_1; \textbf{while } b \textbf{ do } c_1,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{n-2} \mathsf{abort}$. Similarly, we know that $\langle c2; \textbf{while } b \textbf{ do } c_2,\ \sigma\rangle$ can be reached in 2 steps, so it remains to show that $\langle c2; \textbf{while } b \textbf{ do } c_2,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$. Since, sequential small steps are deterministic we have two cases: either $\langle c_1,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{j-2} \mathsf{abort}$ or it completes $\langle c_1,\ \sigma\rangle \Downarrow_{j_1} \langle \textbf{skip},\ \sigma'\rangle$ and $\langle \textbf{skip}; \textbf{while } b \textbf{ do } c_1,\ \sigma'\rangle \xrightarrow{\mathbf{u}}{}^{j_2} \mathsf{abort}$ where $j-2 = j_1 + j_2$ (c_1 cannot reach a barrier as that would prevent the aborted execution). In the first case, from $c_1 \preceq c_2$, we know that $\langle c_2,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ and we can derive the goal trivially. In the second case, from $c_1 \preceq c_2$ we know that either $\langle c_2,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ or $\langle c_2,\ \sigma\rangle \Downarrow \langle \textbf{skip},\ \sigma'\rangle$. If, c_2 aborts, we conclude since we can derive trivially the goal. Otherwise, we show that $\langle \textbf{while } b \textbf{ do } c_2,\ \sigma'\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ using the induction hypothesis, which composed with the big step give us the goal. For condition 2, given that $\langle \textbf{while } b \textbf{ do } c_1,\ \sigma\rangle \Downarrow \langle c_1',\ \sigma'\rangle$ we need to show that either $\langle \textbf{while } b \textbf{ do } c_2,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ or exists c_2' such that $\langle \textbf{while } b \textbf{ do } c_2,\ \sigma\rangle \Downarrow \langle c_2',\ \sigma'\rangle$ and conditions (a) to (d) hold for c_2' and all index $j \leq k$–1. We do strong induction over the number of steps taken by **while** b **do** c_1 to complete a big step. The base case is not possible as we can always unfold the loop. In the inductive case, we know that we can unfold the loop and from there perform a big step. We have two cases to consider. If the condition b is false, the loop completes with **skip**, the same can be show for **while** b **do** c_2, and we conclude trivially. If the condition b is true we know the big step continues from $c_1; \textbf{while } b \textbf{ do } c_1$, similar happens to **while** b **do** c_2. Now we have two cases to consider. If c_1 completes a big step, ending in configuration $\langle \textbf{skip},\ \sigma'\rangle$ from $c_1 \preceq c_2$ we know that either c_2 aborts, in which case we know $c_2; \textbf{while } b \textbf{ do } c_2$ also aborts, or it completes a big step in which we can continue the proof again for **while** b **do** c_1 and **while** b **do** c_2 using the induction hypothesis given that the number of steps has decreased. If c_1 completes a big step, endind in a barrier configuration $\langle c_1',\ \sigma'\rangle$ from $c_1 \preceq c_2$ we know that either c_2 aborts, in which case we know $c_2; \textbf{while } b \textbf{ do } c_2$ also aborts, or it completes a big step arriving in a similar barrier. The proof continues by looking at conditions (b) to (d) and using the subsumption relations obtaines in addition to applying the first induction hypothesis to establish **while** b **do** $c_1 \preceq_j$ **while** b **do** c_2.

- If $c_1 \preceq c_2$, then $c_1 \parallel c \preceq c_2 \parallel c$. From $c_1 \parallel c \preceq c_2 \parallel c$, by definition, we need to show that for all k, $c_1 \parallel c \preceq_k c_2 \parallel c$. If $k = 0$ that is trivial. If $k > 0$, then we unfold Definition **??** in the goal and we need to show that both conditions 1 and 2 hold for all $j \leq k-1$. Condition 1 holds trivially since $\langle c_1 \parallel c,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \mathsf{abort}$ is never satisfied. For condition 2, we know that $\langle c_1 \parallel c,\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \langle c_1 \parallel c,\ \sigma\rangle$ and we need to deal with constraint (b) where we need to show $c_1 \preceq_j c_2$, $c \preceq_j c$, **skip** $\preceq_j$ **skip** ($\mathbf{E} = [\,]$). $c_1 \preceq_j c_2$ we obtain from $c_1 \preceq c_2$. $c \preceq_j c$ and **skip** $\preceq_j$ **skip** we obtain using Lemma B.2.
- If $c_1 \preceq c_2$, then $c \parallel c_1 \preceq c \parallel c_2$. Similar to the previous case.
- If $c_1 \preceq c_2$, then **atomic** $c_1 \preceq$ **atomic** c_2. Similar to the previous case.

□

Lemma B.7. If $\langle c,\ \sigma\rangle \xrightarrow{\mathbf{u}} \langle c',\ \sigma'\rangle$, and $c''' \preceq_{\mathbf{t}} c'$, then there exists c'' such that $c'' \preceq c$ and $\langle c'',\ \sigma\rangle \xrightarrow{\mathbf{u}}{}^{*} \langle c''',\ \sigma'\rangle$.

Proof. We look at the step taken by $\langle c,\ \sigma\rangle$ into $\langle c',\ \sigma'\rangle$, it can either be the execution of a command c_1 that modifies the state (e.g. $c_1 = [e] := e'$) or a control command that does not modify the state. In the first case, we instantiate c'' with $c_1; c'$ and we know that configuration $\langle c',\ \sigma'\rangle$ can be reached in 2 steps having $\langle \textbf{skip}; c',\ \sigma'\rangle$ as intermediate configuration. In the second case, we instantiate c'' with c', which is already the final configuration, therefore no additional stepping is necessary. □

Corollary B.8. If $\langle T,\ \sigma\rangle \longmapsto \langle T',\ \sigma'\rangle$, and $T''' \preceq_{\mathbf{t}} T'$, then there exists T'' such that $T'' \preceq_{\mathbf{t}} T$ and $\langle T'',\ \sigma\rangle \longmapsto^{*} \langle T''',\ \sigma'\rangle$.

Lemma B.9. $[\succeq_{\mathbf{t}}]\ \langle T,\ \sigma\rangle \longmapsto^{*} \langle \textbf{skip},\ \sigma'\rangle$ iff there exists a T' such that $T \succeq_{\mathbf{t}} T'$ and $\langle T',\ \sigma\rangle \longmapsto^{*} \langle \textbf{skip},\ \sigma'\rangle$.

Proof. We only show left to right direction, the other is trivial. We prove by induction over the number of steps for T to reach **skip**. It is trivial for 0 steps since $T = \textbf{skip}$ and $\sigma = \sigma'$; we let $T' = \textbf{skip}$ and show **skip** $\succeq_{\mathbf{t}}$ **skip** using Corollary B.3. For $k+1$ steps, there exists T'', T''' and σ'' such that $T''' \preceq_{\mathbf{t}} T$, $\langle T''',\ \sigma\rangle \longmapsto \langle T'',\ \sigma''\rangle$ and $[\succeq_{\mathbf{t}}]\ \langle T'',\ \sigma''\rangle \longmapsto^{k} \langle \textbf{skip},\ \sigma'\rangle$. By induction hypothesis, we know there exists T'''' such that $T'''' \preceq_{\mathbf{t}} T''$ and $\langle T'''',\ \sigma''\rangle \longmapsto^{*} \langle \textbf{skip},\ \sigma'\rangle$. By Lemma B.8 we know there exists T''''' such that $T''''' \preceq_{\mathbf{t}} T'''$ and $\langle T''''',\ \sigma\rangle \longmapsto^{*} \langle T'''',\ \sigma''\rangle$. Therefore we have $\langle T''''',\ \sigma\rangle \longmapsto^{*} \langle \textbf{skip},\ \sigma'\rangle$. Since $T''' \preceq_{\mathbf{t}} T$, $T''''' \preceq_{\mathbf{t}} T'''$, using Corollary B.5, we also have $T''''' \preceq_{\mathbf{t}} T$. □

C. Prooving the DRF Guarantee

The key lemmas to establish the DRF-guaranntee (Theorem 5.3) are Lemmas C.5 and C.6. Another important lemma is the commutativity property show in Lemma C.1.

Lemma C.1. If $\delta_1 \smile \delta_2$, and c_1 and c_2 only share read-only variables, then $(\exists\sigma'.\ \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma'\rangle \wedge \langle c_2,\ \sigma'\rangle \xrightarrow[\delta_2]{} \langle c_2',\ \sigma''\rangle)$ iff $(\exists\sigma'.\ \langle c_2,\ \sigma\rangle \xrightarrow[\delta_2]{} \langle c_2',\ \sigma'\rangle \wedge \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma''\rangle)$.

Corollary C.2. If $\delta_1 \smile \delta_2$, and c_1 and c_2 only share read-only variables, then $(\exists\sigma'.\ \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma'\rangle \wedge \langle c_2,\ \sigma'\rangle \xrightarrow[\delta_2]{}{}^{*} \langle c_2',\ \sigma''\rangle)$ iff $(\exists\sigma'.\ \langle c_2,\ \sigma\rangle \xrightarrow[\delta_2]{}{}^{*} \langle c_2',\ \sigma'\rangle \wedge \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma''\rangle)$.

Corollary C.3. If $\delta_1 \smile \delta_2$, and c_1 and c_2 only share read-only variables, then $(\exists\sigma'.\ \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma'\rangle \wedge \langle c_2,\ \sigma'\rangle \Downarrow_{\delta_2} \langle c_2',\ \sigma''\rangle)$ iff $(\exists\sigma'.\ \langle c_2,\ \sigma\rangle \Downarrow_{\delta_2} \langle c_2',\ \sigma'\rangle \wedge \langle c_1,\ \sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1',\ \sigma''\rangle)$.

Lemma C.4. If $\langle T,\ \sigma\rangle$ racefree and $\langle T,\ \sigma\rangle \longmapsto \langle T',\ \sigma'\rangle$, then $\langle T',\ \sigma'\rangle$ racefree.

Proof. From the definition of $\langle T,\ \sigma\rangle$ racefree we have $\neg(\langle \lfloor T \rfloor,\ \sigma\rangle \Longrightarrow^{*} \mathsf{race})$; and from the definition of $\langle T',\ \sigma'\rangle$ racefree we have $\neg(\langle \lfloor T' \rfloor,\ \sigma'\rangle \Longrightarrow^{*} \mathsf{race})$. If we remove the negations, from $\langle T,\ \sigma\rangle \longmapsto \langle T',\ \sigma'\rangle$, and $\langle \lfloor T' \rfloor,\ \sigma'\rangle \Longrightarrow^{*} \mathsf{race}$, we need to show $\langle \lfloor T \rfloor,\ \sigma\rangle \Longrightarrow^{*} \mathsf{race}$. By inversion of $\langle T,\ \sigma\rangle \longmapsto \langle T',\ \sigma'\rangle$, we have the following cases:

1. $T = \mathbf{T}[\,c\,]$, $T' = \mathbf{T}[\,c'\,]$, and $\langle c,\ \sigma\rangle \xrightarrow{\mathbf{u}} \langle c',\ \sigma'\rangle$. We know then that from $\langle \lfloor \mathbf{T}[\,c'\,] \rfloor,\ \sigma'\rangle \Longrightarrow^{*} \mathsf{race}$, and $\langle c,\ \sigma\rangle \xrightarrow[\delta]{\mathbf{u}} \langle c',\ \sigma'\rangle$, we need to establish

$$\langle \lfloor \mathbf{T}[\,c\,] \rfloor,\ \sigma\rangle \Longrightarrow^{*} \mathsf{race}$$

We know there exists $\widetilde{\mathbf{T}}$ such that

- $\lfloor \mathbf{T}[\,c\,] \rfloor = \widetilde{\mathbf{T}}[\,(c, \mathit{emp})\,]$;
- $\lfloor \mathbf{T}[\,c'\,] \rfloor = \widetilde{\mathbf{T}}[\,(c', \mathit{emp})\,]$;

We do induction over the number of steps k before c' reaches the race. In the base case, we have two options for race:

- $\widetilde{\mathbf{T}}[\,(c', \mathit{emp})\,] = \widetilde{\mathbf{T}}'[\,(c'', \delta')\,]$ where $\langle c'',\, \sigma'\rangle \xrightarrow[\delta'']{\mathbf{u}}{}^{*} \langle c''',\, \sigma''\rangle$ or $\langle c'',\, \sigma'\rangle \xrightarrow[\delta'']{\mathbf{o}} \langle c''',\, \sigma''\rangle$, and $\neg\,\mathsf{wft}(\widetilde{\mathbf{T}}', \delta'')$
- $\widetilde{\mathbf{T}}[\,(c', \mathit{emp})\,] = \widetilde{\mathbf{T}}'[\,(\mathbf{E}[\,\mathbf{atomic}\ c''\,], \delta')\,]$ where $\langle c'',\, \sigma'\rangle \xrightarrow[\delta'']{}{}^{*} \langle c''',\, \sigma''\rangle$, and $\neg\,\mathsf{wft}(\widetilde{\mathbf{T}}', \delta'')$

For both cases, if $\widetilde{\mathbf{T}} = \widetilde{\mathbf{T}}'$, then $c' = c''$ (first case) or $c' = \mathbf{E}[\,\mathbf{atomic}\ c''\,]$ (second case). In the first case, we can establish the goal by knowing that (a) $\langle c,\, \sigma\rangle \xrightarrow[\delta''']{\mathbf{u}}{}^{*} \langle c''',\, \sigma''\rangle$ and $\neg\,\mathsf{wft}(\widetilde{\mathbf{T}}', \delta''')$ where $\delta''' = \delta \cup \delta''$; or (b) $\langle c,\, \sigma\rangle \Downarrow_{\delta} \langle c',\, \sigma'\rangle$ and $\mathsf{wft}(\widetilde{\mathbf{T}}, \delta)$ to be composed with $\langle \widetilde{\mathbf{T}}[\,(c', \delta)\,],\, \sigma'\rangle \implies^{*} \mathsf{race}$; or (c) $\langle c,\, \sigma\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*} \langle c',\, \sigma'\rangle$ and $\neg\,\mathsf{wft}(\widetilde{\mathbf{T}}, \delta)$. In the second case, atomic command, it is simmilar to allocation just shown. For both cases, if $\widetilde{\mathbf{T}} \neq \widetilde{\mathbf{T}}'$, we can just update $\widetilde{\mathbf{T}}'$ to use (c, emp) instead of (c', emp) and apply the stuttering rule (second rule) to increase the footprint from *emp* to δ if that results in $\mathsf{wft}(\widetilde{\mathbf{T}}, \delta)$, otherwise there is a race is defined. In the inductive case, we look into the first step of the grainless multistep that leads to a race. If it comes from the same thread we just merge the steps similar to what was shown in the base case. If they come from different threads either they are non conflicting operations that can be flipped using Corollary C.2 or Corollary C.3 and we apply the induction hypothesis before composing the steps back together, or they are conflicting where a race is defined.

2. Memory allocation/disposal case is similar to atomic case that follows.
3. $T = \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,]$, $T' = \mathbf{T}[\,\mathbf{E}[\,\mathbf{skip}\,]\,]$, and $\langle c,\, \sigma\rangle \longrightarrow^{*} \langle \mathbf{skip},\, \sigma'\rangle$. We know then that from $\langle \lfloor \mathbf{T}[\,\mathbf{E}[\,\mathbf{skip}\,]\,] \rfloor,\, \sigma'\rangle \implies^{*} \mathsf{race}$, and $\langle c,\, \sigma\rangle \longrightarrow^{*} \langle \mathbf{skip},\, \sigma'\rangle$, we need to establish

$$\langle \lfloor \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,] \rfloor,\, \sigma\rangle \implies^{*} \mathsf{race}$$

We know there exists $\widetilde{\mathbf{T}}$ such that

- $\lfloor \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,] \rfloor = \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,], \mathit{emp})\,]$;
- $\lfloor \mathbf{T}[\,\mathbf{E}[\,\mathbf{skip}\,]\,] \rfloor = \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{skip}\,], \mathit{emp})\,]$;

From $\langle c,\, \sigma\rangle \longrightarrow^{*} \langle \mathbf{skip},\, \sigma'\rangle$ we know there exists δ' such that $\langle c,\, \sigma\rangle \xrightarrow[\delta']{}{}^{*} \langle \mathbf{skip},\, \sigma'\rangle$. Then we consider two cases:

(a) if $\mathsf{wft}(\widetilde{\mathbf{T}}, \delta')$, from the grainless semantics we obtain

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,], \delta)\,],\, \sigma\rangle \implies \langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{skip}\,], \mathit{emp})\,],\, \sigma'\rangle$$

where we assume $\delta = \mathit{emp}$, which combined with

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{skip}\,], \mathit{emp})\,],\, \sigma'\rangle \implies^{*} \mathsf{race}$$

give us the goal.

(b) if $\neg\mathsf{wft}(\widetilde{\mathbf{T}}, \delta')$, considering

$$\langle c,\, \sigma\rangle \xrightarrow[\delta']{}{}^{*} \langle c',\, \sigma'\rangle$$

where $c' = \mathbf{skip}$, we obtain

$$\langle \widetilde{\mathbf{T}}[\,(\mathbf{E}[\,\mathbf{atomic}\ c\,], \mathit{emp})\,],\, \sigma\rangle \implies^{*} \mathsf{race}$$

direct from its definition.

4. Thread spawning case is similar to thread join case that follows.
5. $T = \mathbf{T}[\,\langle\!\langle \mathbf{skip}, \mathbf{skip} \rangle\!\rangle c\,]$, $T' = \mathbf{T}[\,c\,]$, and $\sigma' = \sigma$. We know then that from $\langle \lfloor \mathbf{T}[\,c\,] \rfloor,\, \sigma\rangle \implies^{*} \mathsf{race}$ we need to establish $\langle \lfloor \mathbf{T}[\,\langle\!\langle \mathbf{skip}, \mathbf{skip} \rangle\!\rangle c\,] \rfloor,\, \sigma\rangle \implies^{*} \mathsf{race}$. We know there exists $\widetilde{\mathbf{T}}$ such that

- $\lfloor \mathbf{T}[\,\langle\!\langle \mathbf{skip}, \mathbf{skip} \rangle\!\rangle c\,] \rfloor = \widetilde{\mathbf{T}}[\,\langle\!\langle (\mathbf{skip}, \mathit{emp}), (\mathbf{skip}, \mathit{emp}) \rangle\!\rangle c\,]$;
- $\lfloor \mathbf{T}[\,c\,] \rfloor = \widetilde{\mathbf{T}}[\,(c, \mathit{emp})\,]$;

Then from $\langle \widetilde{\mathbf{T}}[\,(c, \mathit{emp})\,],\, \sigma\rangle \implies^{*} \mathsf{race}$, and from the grainless semantics

$$\langle \widetilde{\mathbf{T}}[\,\langle\!\langle (\mathbf{skip}, \delta_1), (\mathbf{skip}, \delta_2) \rangle\!\rangle c\,],\, \sigma\rangle \implies \langle \widetilde{\mathbf{T}}[\,(c, \mathit{emp})\,],\, \sigma\rangle$$

(where $\delta_1 = \mathit{emp}$ and $\delta_2 = \mathit{emp}$) we can establish

$$\langle \widetilde{\mathbf{T}}[\,\langle\!\langle (\mathbf{skip}, \mathit{emp}), (\mathbf{skip}, \mathit{emp}) \rangle\!\rangle c\,],\, \sigma\rangle \implies^{*} \mathsf{race}$$

□

Lemma C.5. If $\langle T,\, \sigma\rangle$ racefree, then

1. $\langle T,\, \sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\, \sigma'\rangle$ iff $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$;
2. $\langle T,\, \sigma\rangle \longmapsto^{*} \mathsf{abort}$ iff $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \mathsf{abort}$.

Proof. We show the proof for the right direction of 1, i.e. $\langle T,\, \sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\, \sigma'\rangle$ implies $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$. We do induction over the number of execution steps. The base case is trivial. Now suppose $\langle T,\, \sigma\rangle \longmapsto^{k+1} \langle \mathbf{skip},\, \sigma'\rangle$. We know there exists T_1 and σ_1 such that $\langle T,\, \sigma\rangle \longmapsto \langle T_1,\, \sigma_1\rangle$ and $\langle T_1,\, \sigma_1\rangle \longmapsto^{k} \langle \mathbf{skip},\, \sigma'\rangle$. By Lemma C.4 we know $\langle T_1,\, \sigma_1\rangle$ racefree. Then by the induction hypothesis we know $\langle \lfloor T_1 \rfloor,\, \sigma_1\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$. By $\langle T,\, \sigma\rangle \longmapsto \langle T_1,\, \sigma_1\rangle$ we know there are the following cases:

- $T = \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,]$, $T_1 = \mathbf{T}[\,\mathbf{E}[\,\mathbf{skip}\,]\,]$, and $\langle c,\, \sigma\rangle \longrightarrow^{*} \langle \mathbf{skip},\, \sigma_1\rangle$. Then we know $\langle \lfloor T \rfloor,\, \sigma\rangle \implies \langle \lfloor T_1 \rfloor,\, \sigma_1\rangle$. By $\langle \lfloor T_1 \rfloor,\, \sigma_1\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$ we have $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$.
- $\langle T,\, \sigma\rangle \longmapsto \langle T_1,\, \sigma_1\rangle$ is one of the rest of ordered executions, i.e. **cons**, **dispose**, fork or join operations. The proof is similar to the first case.
- $T = \mathbf{T}[\,c\,]$, $\langle c,\, \sigma\rangle \xrightarrow{\mathbf{u}} \langle c',\, \sigma_1\rangle$, and $T_1 = \mathbf{T}[\,c'\,]$. We do induction over the number of steps taken by $\langle \lfloor T_1 \rfloor,\, \sigma_1\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$ The base case is trivial and we can just derive a big step from $\langle c,\, \sigma\rangle \xrightarrow{\mathbf{u}} \langle c',\, \sigma_1\rangle$. In the inductive step we have two prossibilities. If the big step is performed by the same thread we merge the operations into a single grainless step (or two if the step is performed by a ordered operation). If the big step is performed by a different thread we have two options. If the footprints of both steps are not conflicting we use either Corollary C.2 or Corollary C.3 to flip the operations, and we apply the inductive hypothesis prior to composing the steps into a single step. If the footprints of both steps are conflicting, then a race is defined which contradics our premisse of race freedom.

□

Lemma C.6. If $\langle T,\, \sigma\rangle$ racefree, then

1. $[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto^{*} \mathsf{abort}$ iff $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \mathsf{abort}$.
2. if $\neg(\langle T,\, \sigma\rangle \longmapsto^{*} \mathsf{abort})$, then $[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\, \sigma'\rangle$ iff $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$.

Proof. The left directions (i.e. the right-hand side of "iff" implies the left-hand side) of both sub-goals are trivial: the mixed step semantics can be viewed as a special scheduling of the left-hand side. We show the right direction of 2, i.e. $[\succeq_{\mathbf{t}}]\ \langle T,\, \sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\, \sigma'\rangle$ implies $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$. By Lemma B.9 we know there exists T' such that $T' \preceq_{\mathbf{t}} T$ and $\langle T',\, \sigma\rangle \longmapsto^{*} \langle \mathbf{skip},\, \sigma'\rangle$. Since $\langle T,\, \sigma\rangle$ racefree and $T' \preceq_{\mathbf{t}} T$, by Corollary C.14 we know $\langle T',\, \sigma\rangle$ racefree. Then by Lemma C.5 we know $\langle \lfloor T' \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$. Since $T' \preceq_{\mathbf{t}} T$, we have $\lfloor T' \rfloor \preceq_{\mathbf{t}} \lfloor T \rfloor$. By Corollary C.14 we have $\langle \lfloor T \rfloor,\, \sigma\rangle \implies^{*} \langle (\mathbf{skip}, \_),\, \sigma'\rangle$. □

Lemma C.7. If $\langle c,\,(h,s)\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*}\langle c',\,\sigma'\rangle$, then $(\delta.rs \cup \delta.ws) \subseteq \mathsf{dom}(h)$.

Lemma C.8. If $\langle c,\,(h,s)\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*}\langle c',\,(h',s')\rangle$, $h = h_1 \uplus h_2$ and $\mathsf{dom}(h_1) = (\delta.rs \cup \delta.ws)$, then there exists h_1' such that $h' = h_1' \uplus h_2$ and $\langle c,\,(h_1,s)\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{*}\langle c',\,(h_1',s')\rangle$.

Lemma C.9. For all k, if $\langle c,\,(h,s)\rangle \xrightarrow[\delta]{\mathbf{u}}{}^{k}\langle c',\,\sigma\rangle$, $h \sharp h'$ and $(\delta.rs \cup \delta.ws) - \mathsf{dom}(h') \neq \emptyset$, then $\langle c,\,(h',s)\rangle \xrightarrow{\mathbf{u}}{}^{*}\,\mathsf{abort}$.
Here $h \sharp h' \stackrel{\text{def}}{=} \forall \ell.\ \ell \in (\mathsf{dom}(h) \cap \mathsf{dom}(h')) \rightarrow h(\ell) = h'(\ell)$.

Lemma C.10. If $c_1 \preceq c_2$ and $\langle c_1,\,\sigma\rangle \Downarrow_{\delta_1} \langle c_1',\,\sigma'\rangle$, then either $\langle c_2,\,\sigma\rangle \xrightarrow{\mathbf{u}}{}^{*}\,\mathsf{abort}$ or there exist c_2' and δ_2 such that $\langle c_2,\,\sigma\rangle \Downarrow_{\delta_2} \langle c_2',\,\sigma'\rangle$, $\delta_1 \subseteq \delta_2$, and $c_1' \preceq c_2'$.

Proof sketch. By Lemma B.1 we know that either $\langle c_2,\,\sigma\rangle \xrightarrow{\mathbf{u}}{}^{*}\,\mathsf{abort}$ or there exist c_2' and δ_2 such that $\langle c_2,\,\sigma\rangle \Downarrow_{\delta_2} \langle c_2',\,\sigma'\rangle$ and $c_1' \preceq c_2'$. Now we prove $\delta_1 \subseteq \delta_2$, if $\langle c_2,\,\sigma\rangle \Downarrow_{\delta_2} \langle c_2',\,\sigma'\rangle$. Let $\sigma = (h,s)$. Suppose $(\delta_1.rs \cup \delta_1.ws) - (\delta_2.rs \cup \delta_2.ws) \neq \emptyset$. By Lemma C.7, we know $(\delta_1.rs \cup \delta_1.ws \cup \delta_2.rs \cup \delta_2.ws) \subseteq \mathsf{dom}(h)$. So there exists h' such that $h' \subset h$ and $\mathsf{dom}(h') = (\delta_2.rs \cup \delta_2.ws)$. By Lemma C.9 we know $\langle c_1,\,(h',s)\rangle \xrightarrow{\mathbf{u}}{}^{*}\,\mathsf{abort}$. By Lemma C.8 we know there exists σ'' such that $\langle c_2,\,(h',s)\rangle \Downarrow_{\delta_2} \langle c_2',\,\sigma''\rangle$. This is in conflict with $c_1 \preceq c_2$, which requires that $\langle c_2,\,(h',s)\rangle \xrightarrow{\mathbf{u}}{}^{*}\,\mathsf{abort}$. □

Lemma C.11. If c_1 does not contain **atomic** blocks and parallel compositions, $c_1 \preceq c_2$, then

1. if $\langle c_1,\,\sigma\rangle \longrightarrow^{*} \mathsf{abort}$, then $\langle c_2,\,\sigma\rangle \longrightarrow^{*} \mathsf{abort}$;
2. if $\langle c_1,\,\sigma\rangle \xrightarrow[\delta_1]{}{}^{*}\langle \mathbf{skip},\,\sigma'\rangle$, then either $\langle c_2,\,\sigma\rangle \longrightarrow^{*} \mathsf{abort}$ or there exists δ_2 such that $\langle c_2,\,\sigma\rangle \xrightarrow[\delta_2]{}{}^{*}\langle \mathbf{skip},\,\sigma'\rangle$ and $\delta_1 \subseteq \delta_2$.

Proof. If c_1 starts with unordered commands, the proof follows from Lemma C.10. For other cases (c_1 starts with **cons** or **dispose**), the proof is trivial. □

Lemma C.12. If $(c_1,\delta_1) \preceq_{\mathbf{t}} (c_2,\delta_2)$, then

1. if $\langle (c_1,\delta_1),\,\sigma\rangle \Longrightarrow \mathsf{abort}$, then $\langle (c_2,\delta_2),\,\sigma\rangle \Longrightarrow \mathsf{abort}$;
2. if $\langle (c_1,\delta_1),\,\sigma\rangle \Longrightarrow \langle \widetilde{T}_1,\,\sigma'\rangle$, then either $\langle (c_2,\delta_2),\,\sigma\rangle \Longrightarrow \mathsf{abort}$ or there exists $\widetilde{T}_2$ such that $\langle (c_2,\delta_2),\,\sigma\rangle \Longrightarrow \langle \widetilde{T}_2,\,\sigma'\rangle$ and $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$.

Proof. If c_1 starts with unordered commands, the proof follows from Lemma C.10. If c_1 starts with an **atomic** block, we apply Lemma C.11. For other cases (c_1 starts with **cons**, **dispose** or parallel composition), the proof is trivial. □

Corollary C.13. If $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$, then

1. if $\langle \widetilde{T}_1,\,\sigma\rangle \Longrightarrow \mathsf{abort}$ or race, then $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow \mathsf{abort}$ or race;
2. if $\langle \widetilde{T}_1,\,\sigma\rangle \Longrightarrow \langle \widetilde{T}_1',\,\sigma'\rangle$, then $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow \mathsf{abort}$ or race, or there exists $\widetilde{T}_2'$, such that $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow \langle \widetilde{T}_2',\,\sigma'\rangle$ and $\widetilde{T}_1' \preceq_{\mathbf{t}} \widetilde{T}_2'$.

Corollary C.14. If $\widetilde{T}_1 \preceq_{\mathbf{t}} \widetilde{T}_2$, then

1. if $\langle \widetilde{T}_1,\,\sigma\rangle \Longrightarrow^{*} \mathsf{abort}$ or race, then $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow^{*} \mathsf{abort}$ or race;
2. if $\langle \widetilde{T}_1,\,\sigma\rangle \Longrightarrow^{*} \langle (\mathbf{skip},\delta),\,\sigma'\rangle$, then $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow^{*} \mathsf{abort}$ or race, or $\langle \widetilde{T}_2,\,\sigma\rangle \Longrightarrow^{*} \langle (\mathbf{skip},\_),\,\sigma'\rangle$.

D. Proving the Soundness of CSL

Here we prove Lemma 6.6, which is the major lemma that we use to drive the soundness of CSL with respect to the grainless semantics (i.e. Lemma 6.8). The proof follows the standard techniques to establish the soundness of separation logic and CSL, except that we need extra efforts to close the syntactic gap between the commands c (used in logic rules) and the thread threes $\widetilde{T}$ (used in the operational semantics). In this section, we always assume that I is precise.

We first show the locality of each primitive operations and concurrent transitions below.

Lemma D.1 (Locality)**.** If $\neg(\langle c,\,(h,s)\rangle \longrightarrow \mathsf{abort})$, then $\neg(\langle c,\,(h \uplus h',s)\rangle \longrightarrow \mathsf{abort})$; and for all $\langle c,\,(h \uplus h',s)\rangle \xrightarrow[\delta]{} \langle c_1,\,\sigma_1\rangle$, there exists h_1 such that $\sigma_1.h = h_1 \uplus h'$ and $\langle c,\,(h,s)\rangle \xrightarrow[\delta]{} \langle c_1,\,(h_1,\sigma_1.s)\rangle$.

Lemma D.2 (Seq-Locality)**.**
If $\neg(\langle c,\,(h,s)\rangle \longrightarrow^{*} \mathsf{abort})$, then $\neg(\langle c,\,(h \uplus h',s)\rangle \longrightarrow^{*} \mathsf{abort})$; and for all $\langle c,\,(h \uplus h',s)\rangle \xrightarrow[\delta]{}{}^{*}\langle c_1,\,\sigma_1\rangle$, there exists h_1 such that $\sigma_1.h = h_1 \uplus h'$ and $\langle c,\,(h,s)\rangle \xrightarrow[\delta]{}{}^{*}\langle c_1,\,(h_1,\sigma_1.s)\rangle$.

Lemma D.3 (Par-Locality)**.** If $\neg\,(\langle \widetilde{T},\,(h,s)\rangle \Longrightarrow \mathsf{abort}$ or $\mathsf{race})$, then $\neg\,(\langle \widetilde{T},\,(h \uplus h',s)\rangle \Longrightarrow \mathsf{abort}$ or $\mathsf{race})$, and if $\langle \widetilde{T},\,(h \uplus h',s)\rangle \Longrightarrow \langle \widetilde{T}',\,\sigma_1\rangle$, then there exists h_1 such that $\sigma_1.h = h_1 \uplus h'$ and $\langle \widetilde{T},\,(h,s)\rangle \Longrightarrow \langle \widetilde{T}',\,(h_1,\sigma_1.s)\rangle$.

This lemma below shows that the thread tree $\widetilde{T} = (\mathbf{skip}, emp)$ preserves the invariant.

Lemma D.4. If $(h,s) \models I * q$, then for all k we know $I \models \langle (\mathbf{skip}, emp),\,(h,s)\rangle \rhd_k q$ holds for all k.

Proof. Trivial, by induction over k and Definition 6.5. □

The proof of Lemma 6.6 is done by induction over the derivation of $I \vdash \{p\}\,c\,\{q\}$.

The* ENV *rule. If the ENV rule is the last rule applied to derive $I \vdash \{p\}\,c\,\{q\}$, we know $\vdash \{p\}\,c\,\{q\}$. By Lemma 6.2 we know $\models \{p\}c\{q\}$. We first prove the following lemma. Then our goal is proved as Lemma D.6.

Lemma D.5. For all k, if $\langle c,\,\sigma\rangle \xrightarrow[\delta]{}{}^{k}\langle \mathbf{skip},\,\sigma'\rangle$, and $\langle c,\,\sigma\rangle \Downarrow_{\delta'} \langle c',\,\sigma''\rangle$, then there exists δ'' such that $\langle c',\,\sigma''\rangle \xrightarrow[\delta'']{}{}^{*}\langle \mathbf{skip},\,\sigma'\rangle$.

Proof. By induction over k. The base case is trivial. If $k = j+1$, we know there exists c_1, σ_1, δ_1 and δ_2 such that $\langle c,\,\sigma\rangle \xrightarrow[\delta_1]{} \langle c_1,\,\sigma_1\rangle$ and $\langle c_1,\,\sigma_1\rangle \xrightarrow[\delta_2]{}{}^{j}\langle \mathbf{skip},\,\sigma'\rangle$. If c starts with **cons** or **dispose**, we know $c' = c$ and $\sigma'' = \sigma$. The proof is trivial. Otherwise, we know $\langle c,\,\sigma\rangle \xrightarrow[\delta_1]{\mathbf{u}} \langle c_1,\,\sigma_1\rangle$ and $\langle c_1,\,\sigma_1\rangle \Downarrow_{\delta_1'} \langle c',\,\sigma''\rangle$ for some δ_1'. Then the goal follows trivially from the induction hypothesis. □

Lemma D.6. For all k, if

- $(h_1,s) \models I$;
- $(h_2,s) \models q$;
- $\neg(\langle c,\,(h_2,s)\rangle \longrightarrow^{*} \mathsf{abort})$;
- for all h_2' and σ', if $\langle c,\,(h_2,s)\rangle \xrightarrow[\delta']{}{}^{*}\langle \mathbf{skip},\,(h_2',s')\rangle$ then $(h_2',s') \models q$;
- $\sigma \models \delta \uplus I$;

then $I \models \langle (c,\delta),\,(h_1 \uplus h_2,s)\rangle \rhd_k q$.

Proof. We prove by induction by k. The base case is trivial. Now we consider the case that $k = j+1$. By Definition 6.5, we need to show all the 6 conditions hold. The first four conditions are obvious. Condition 6 is also trivial. Condition 5 is proved by applying the locality property (Lemma D.2), Lemma D.5, and the induction hypothesis. □

The* PAR *rule. If the PAR rule is the last rule applied to derive $I \vdash \{p\}\, c\, \{q\}$, we know p is in the form of $p_1 * p_2$, q is in the form of $q_1 * q_2$, c is in the form of $c_1 \parallel c_2$, $I \vdash \{p_1\}\, c_1\, \{q_1\}$ and $I \vdash \{p_2\}\, c_2\, \{q_2\}$. Then by induction hypothesis we know that, for all σ and δ,

- if $\sigma \models I * p_1$ and $\sigma \models \delta \uplus I$, then $I \models \langle (c_1, \delta),\, \sigma\rangle \rhd q_1$;
- if $\sigma \models I * p_2$ and $\sigma \models \delta \uplus I$, then $I \models \langle (c_2, \delta),\, \sigma\rangle \rhd q_2$.

We first prove the following lemma. Our goal is proved as Lemma D.8.

Lemma D.7. For all k, if $h = h_0 \uplus h_1 \uplus h_2$, $(h_0, s) \models I$, $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_k q_1$, $I \models \langle \widetilde{T}_2,\, (h_0 \uplus h_2, s)\rangle \rhd_k q_2$, $\widetilde{T}_1$ does not update free variables in p_2 and $\widetilde{T}_2$, and conversely, then $I \models \langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle \mathbf{skip},\, (h, s)\rangle \rhd_k q_1 * q_2$.

Proof. We prove by induction over k. It is trivial when $k = 0$. Suppose the lemma holds when $k = j$. We prove it holds when $k = j{+}1$. By Definition 6.5 we need to prove the 6 conditions. Proofs for Condition 1 and 3 are trivial. The 4th condition can be derived from $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ and $I \models \langle \widetilde{T}_2,\, (h_0 \uplus h_2, s)\rangle \rhd_{j+1} q_2$.

To prove Condition 2, it is obvious to see

$$\neg(\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle \mathbf{skip},\, (h, s)\rangle \Longrightarrow \mathsf{abort})\,.$$

We only need to prove $\neg(\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle \mathbf{skip},\, (h, s)\rangle \Longrightarrow \mathsf{race})$. By $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ and $I \models \langle \widetilde{T}_2,\, (h_0 \uplus h_2, s)\rangle \rhd_{j+1} q_2$, we know that all the footprints in $\widetilde{T}_1$ are subsets of $\mathsf{dom}(h_1)$, and all those in $\widetilde{T}_2$ are subsets of $\mathsf{dom}(h_2)$. Suppose $\widetilde{T}_1$ executes next step. The footprint for this new step must be within $\mathsf{dom}(h_0 \uplus h_1)$. We know it does not interfere with threads in $\widetilde{T}_2$. By $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ we also know it does not interfere with other threads in $\widetilde{T}_1$. Therefore $\neg(\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle \mathbf{skip},\, (h, s)\rangle \Longrightarrow \mathsf{race})$.

We now prove the 5th condition, i.e., if $\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle \mathbf{skip},\, \sigma\rangle \Longrightarrow \langle \widetilde{T}',\, \sigma'\rangle$, then $\forall i \le j.\ I \models \langle \widetilde{T}',\, \sigma'\rangle \rhd_j q_1 * q_2$. By inspecting the stepping relation, we know there are three possible cases.

First, $\langle \widetilde{T}_1,\, \sigma\rangle \Longrightarrow \langle \widetilde{T}_1',\, \sigma'\rangle$. Then $\widetilde{T}' = \langle\!\langle \widetilde{T}_1', \widetilde{T}_2\rangle\!\rangle \mathbf{skip}$. Suppose $\sigma' = (h', s')$. By $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ and Lemma D.3 we know that there exists h_0' and h_1' such that $h' = h_0' \uplus h_1' \uplus h_2$, $(h_0', s') \models I$, $\forall i \le j.\ I \models \langle \widetilde{T}_1',\, (h_0' \uplus h_1', s')\rangle \rhd_i q_1$, and there exists X such that X does not contain the free variables in $\widetilde{T}_2$ and q_2, and $(h_0' \uplus h_2, s) \parallel_{(I,X)} (h_0' \uplus h_2, s')$. By $I \models \langle \widetilde{T}_2,\, (h_0 \uplus h_2, s)\rangle \rhd_{j+1} q_2$ we know $\forall i \le j.\ I \models \langle \widetilde{T}_2,\, (h_0' \uplus h_2, s')\rangle \rhd_i q_2$. Then we prove our goal by the induction hypothesis.

Second, $\langle \widetilde{T}_2,\, \sigma\rangle \Longrightarrow \langle \widetilde{T}_2',\, \sigma'\rangle$. The proof is similar to above.

Third, $\widetilde{T}_1$ and $\widetilde{T}_2$ are both $(\mathbf{skip}, \_)$. Then $\widetilde{T}' = (\mathbf{skip}, \mathit{emp})$ and $\sigma' = \sigma$. By $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ and $I \models \langle \widetilde{T}_2,\, (h_0 \uplus h_2, s)\rangle \rhd_{j+1} q_2$ we know $(h, s) \models I * q_1 * q_2$. Then the proof simply follows from Lemma D.4.

Next we prove the 6th condition. Suppose $(h, s) \parallel_{(I,X)} (h', s')$, where X does not contain free variables in $\widetilde{T}_1$, $\widetilde{T}_2$, q_1 and q_2. Therefore we know there exists h_0' such that $h' = h_0' \uplus h_1 \uplus h_2$, and $(h_0', s') \models I$. By $I \models \langle \widetilde{T}_1,\, (h_0 \uplus h_1, s)\rangle \rhd_{j+1} q_1$ we know $\forall i \le j.\ I \models \langle \widetilde{T}_1,\, (h_0' \uplus h_1, s')\rangle \rhd_i q_1$. Similarly we have $\forall i \le j.\ I \models \langle \widetilde{T}_2,\, (h_0' \uplus h_2, s')\rangle \rhd_i q_2$. The our goal follows trivially from the induction hypothesis. □

Lemma D.8. If $h = h_0 \uplus h_1 \uplus h_2$, $(h_0, s) \models I$, $I \models \langle (c_1, \mathit{emp}),\, (h_0 \uplus h_1, s)\rangle \rhd q_1$, and $I \models \langle (c_2, \mathit{emp}),\, (h_0 \uplus h_2, s)\rangle \rhd q_2$, then for all k $I \models \langle (c_1 \parallel c_2, \_),\, (h, s)\rangle \rhd_k q_1 * q_2$.

Proof. We do induction over k. The base case is trivial. Suppose $k = j + 1$. By Definition 6.5 we need to prove the 6 conditions. The proofs for the first 4 conditions are trivial. Condition 6 trivially follows from $I \models \langle (c_1, \mathit{emp}),\, (h_0 \uplus h_1, s)\rangle \rhd q_1$, $I \models \langle (c_2, \mathit{emp}),\, (h_0 \uplus h_2, s)\rangle \rhd q_2$, and the induction hypothesis.

By the operational semantics we have $\langle (c_1 \parallel c_2, \_),\, (h, s)\rangle \Longrightarrow \langle\!\langle (c_1, \mathit{emp}), (c_2, \mathit{emp})\rangle\!\rangle \mathbf{skip},\, (h, s)\rangle$. Then by Lemma D.7 we know Condition 5 holds. □

The* SEQ *rule. If the SEQ rule is the last rule applied to derive $I \vdash \{p\}\, c\, \{q\}$, we know c is in the form of $c_1; c_2$, and there exists r such that $I \vdash \{p\}\, c_1\, \{r\}$ and $I \vdash \{r\}\, c_2\, \{q\}$. Then by induction hypothesis we know that, for all σ and δ,

- if $\sigma \models I * p$ and $\sigma \models \delta \uplus I$, then $I \models \langle (c_1, \delta),\, \sigma\rangle \rhd r$;
- if $\sigma \models I * r$ and $\sigma \models \delta \uplus I$, then $I \models \langle (c_2, \delta),\, \sigma\rangle \rhd q$.

We first prove the following auxiliary lemma. Then our goal, as shown in Lemma D.10, is simply the first sub-goal of the lemma below. Here we prove an extra sub-goal (the second one) because, to prove each of them, we need the induction hypothesis of the other.

Lemma D.9. If $I \models \langle (c', \delta'),\, \sigma'\rangle \rhd q$ holds for all σ' and δ' such that $\sigma' \models I * p$ and $\sigma' \models \delta' \uplus I$, then, for all k, the following are true:

1. for all c, if $I \models \langle (c, \delta),\, \sigma\rangle \rhd_k p$, then $I \models \langle (c; c', \delta),\, \sigma\rangle \rhd_k q$.
2. for all c, if $I \models \langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle c,\, \sigma\rangle \rhd_k p$, then $I \models \langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle (c; c'),\, \sigma\rangle \rhd_k q$.

Proof. By induction over k. The base case is always trivial. Suppose $k = j + 1$.

To prove the first sub-goal, we need to prove all the 6 conditions in Definition 6.5. The first four conditions are trivial. The 6th condition can be derived from $I \models \langle (c, \delta),\, \sigma\rangle \rhd_k p$ and the induction hypothesis.

Then we prove the 5th condition. By $\langle c; c',\, \sigma\rangle \Longrightarrow \langle \widetilde{T},\, \sigma'\rangle$, we know there are the following possible cases:

First, $\langle (c, \delta),\, \sigma\rangle \Longrightarrow \langle (c'', \delta''),\, \sigma''\rangle$ and $c'' \neq \mathbf{skip}$. Then we know $\widetilde{T} = (c''; c', \delta'')$. The proof simply follows from $I \models \langle (c, \delta),\, \sigma\rangle \rhd_k p$ and the induction hypothesis.

Second, $\langle (c, \delta),\, \sigma\rangle \Longrightarrow \langle (\mathbf{skip}, \delta''),\, \sigma''\rangle$. By $I \models \langle (c, \delta),\, \sigma\rangle \rhd_k p$ we know $\sigma'' \models I * p$ and $\sigma'' \models \delta'' \uplus I$. Therefore we have $I \models \langle (c', \delta''),\, \sigma''\rangle \rhd q$. Then we know there exist c_1, σ_1, δ_1 and δ_1' such that $\langle (c', \delta''),\, \sigma''\rangle \Downarrow_{\delta_1'} \langle (c_1, \delta_1),\, \sigma_1\rangle$. Since $\langle (c, \delta),\, \sigma\rangle \Longrightarrow \langle (\mathbf{skip}, \delta''),\, \sigma''\rangle$, we know $\langle (c, \delta),\, \sigma\rangle \Longrightarrow \langle (c_1, \delta_1),\, \sigma_1\rangle$. That is, $\widetilde{T} = (c_1, \delta_1)$ and $\sigma' = \sigma_1$. Then our goal follows from $I \models \langle (c', \delta''),\, \sigma''\rangle \rhd q$ and $\langle (c', \delta''),\, \sigma''\rangle \Downarrow_{\delta_1'} \langle (c_1, \delta_1),\, \sigma_1\rangle$.

Third, $\langle (c, \delta),\, \sigma\rangle \Longrightarrow \langle\!\langle (c_1, \delta_1), (c_2, \delta_2)\rangle\!\rangle c'',\, \sigma''\rangle$. Then we know $\widetilde{T} = \langle\!\langle (c_1, \delta_1), (c_2, \delta_2)\rangle\!\rangle (c''; c')$ and $\sigma' = \sigma''$. Also, by $I \models \langle (c, \delta),\, \sigma\rangle \rhd_k p$ we know $I \models \langle\!\langle (c_1, \delta_1), (c_2, \delta_2)\rangle\!\rangle (c''; c'),\, \sigma''\rangle \rhd_j p$. Our goal trivially follows from the induction hypothesis of the *second* sub-goal.

Now we have finished the proof of the first sub-goal. We can now prove the second one. Again, here we only show the proof for the 5th condition. By $\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle (c; c'),\, \sigma\rangle \Longrightarrow \langle \widetilde{T},\, \sigma'\rangle$, we know there are three possible cases.

First, $\langle \widetilde{T}_1,\, \sigma\rangle \Longrightarrow \langle \widetilde{T}_1',\, \sigma'\rangle$. Therefore $\widetilde{T} = \langle\!\langle \widetilde{T}_1', \widetilde{T}_2\rangle\!\rangle (c; c')$. Our goal follows trivially from $I \models \langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle c,\, \sigma\rangle \rhd_k p$ and the induction hypothesis.

Second, $\langle \widetilde{T}_2,\, \sigma\rangle \Longrightarrow \langle \widetilde{T}_2',\, \sigma'\rangle$. The proof is similar to above.

Third, $\langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle c,\, \sigma\rangle \Longrightarrow \langle (c, \mathit{emp}),\, \sigma'\rangle$. Therefore $\widetilde{T} = (c; c', \mathit{emp})$. By $I \models \langle\!\langle \widetilde{T}_1, \widetilde{T}_2\rangle\!\rangle c,\, \sigma\rangle \rhd_k p$ we can prove $I \models \langle (c, \mathit{emp}),\, \sigma'\rangle \rhd_j p$. Therefore our goal trivially follows from the induction hypothesis of the *first* sub-goal. □

Lemma D.10. If

- $I \models \langle (c_1, \delta),\, \sigma\rangle \rhd r$;

- for all all σ' and δ', if $\sigma' \models I * r$ and $\sigma' \models \delta' \uplus I$, then $I \models \langle (c_1, \delta'),\ \sigma' \rangle \rhd q$;

then for all k we have $I \models \langle (c_1; c_2, \delta),\ \sigma \rangle \rhd_k q$

Proof. This is simply the first sub-goal of Lemma D.9. □

Other rules. As we have shown above, the proofs for the PAR rule and the SEQ rule are a bit tricky because we need to close the gap between the syntax of c and $\widetilde{T}$ when we reach the fork and join of threads, and the gap between small-step transitions and big-step ones when we handle sequential compositions. The way we handle other rules are standard, given the locality properties. We omit the proofs here.