Parameterized Memory Models and Concurrent Separation Logic

Rodrigo Ferreira[1], Xinyu Feng[2], and Zhong Shao[1]

[1] Yale University
{rodrigo.ferreira,zhong.shao}@yale.edu
[2] Toyota Technological Institute at Chicago
feng@ttic.edu

Abstract. 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 is weaker than many memory models and accounts for the majority of 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 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. However, they 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 [19], which requires that the instructions in each thread be executed following the program order.

A classic example to demonstrate this problem is Dekker's mutual exclusion algorithm [12] as shown below:

Initially $[x] = [y] = 0$ and $x \neq y$

$[x] := 1;$		$[y] := 1;$
$v_1 := [y];$	$\parallel$	$v_2 := [x];$
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

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 267–286, 2010.

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 is a well-known problem [5, 1].

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. A tutorial about the subject is given by Adve and Gharachorloo [1], and a detailed survey is given by Mosberger [22]. Formalization of memory models for languages such as Java [21, 11], C++ [4] and x86 multiprocessor machine code [24] 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 [2], in which data-race-free programs (i.e. well-synchronized programs) behave in a sequentially consistent manner. The DRF-guarantee is also known as the fundamental property [26] 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 [7] pointed out, most memory models are defined axiomatically by giving partial orders of events in the execution traces of programs. These 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*" [7]. This gap, we believe, partly explains why most program logics for concurrency verification are proved sound only in a sequentially consistent model, and their soundness in relaxed memory models is rarely discussed.

For instance, the soundness of concurrent separation logic (CSL) [23] in sequentially consistent models has been proved in various ways [9, 10, 14, 18], 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. [18] 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, 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

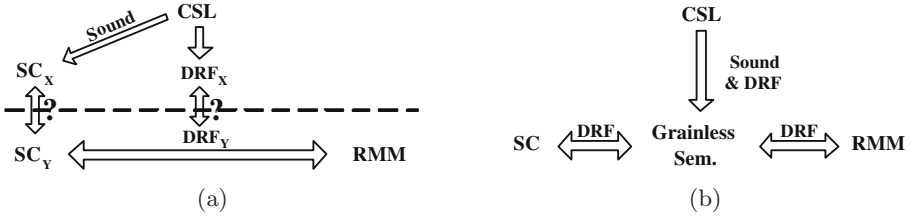

Fig. 1. (a) the gap between the language-side (above the dashed line) and the memory-model-side (below the line); we use subscripts X and Y to represent the different formulations in the two sides; (b) our solution: a new RMM and a grainless semantics. Here single arrows represent (informally) logical 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 6.2, 5.3 and 5.4 respectively.

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.

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 buffering with read bypassing, and those caused by the lack 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 that 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 [3, 20, 30] have developed techniques to prove observational equivalence or simulation relations, which may be used to further derive the subsumption relation. 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 [25, 8], 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

$$
\begin{array}{rrcl}
(\mathit{Expr}) & e & ::= & n \mid x \mid e_1 + e_2 \mid -e \mid \ldots \\
(\mathit{BExpr}) & b & ::= & \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid \ldots \\
(\mathit{Comm}) & c & ::= & x := e \mid x := [e] \mid [e] := e' \mid \mathbf{skip} \mid x := \mathbf{cons}(e_1, \ldots, e_n) \\
 & & & \mid \mathbf{dispose}(e) \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \\
 & & & \mid \mathbf{atomic}\ c \mid c_1 \parallel c_2
\end{array}
$$

The syntax of the language is shown above. 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. This allows us to focus on studying memory reads ($x := [e]$) and writes ($[e] := e'$). **cons** and **dispose** allocate and free memory respectively.

atomic c ensures that the execution of c is not interrupted by other threads. It can be viewed as a synchronization block in high-level languages. 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(ℓ, x, y)) operation:

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

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. 2. Program states (σ) consist of heaps and stores. A heap (h) is a partial mapping from memory locations to integers. A store (s)

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

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

Fig. 2. Runtime constructs and footprints

maps variables to integers. A thread tree (T) 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 to be executed after the two sub-trees both terminate.

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

We give a contextual operational semantics for the language. The sequential context ($\mathbf{E}$) and thread context ($\mathbf{T}$) defined above show the places where the execution of primitive commands occurs. Sequential execution of threads is shown in Fig. 3. We use $[\![e]\!]_s$ to represent the evaluation of e with the store s. The definition is standard and 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. 2 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 $(\_ \longrightarrow^* \_)$.

$$\langle \mathbf{E}[\, x := [e]\,],\ (h,s)\rangle \xrightarrow[(\{\ell\},\emptyset)]{\mathbf{u}} \langle \mathbf{E}[\,\mathbf{skip}\,],\ (h,s')\rangle \quad \text{if } [\![e]\!]_s = \ell,\ h(\ell) = n,\ s' = s[x \rightsquigarrow n]$$

$$\langle \mathbf{E}[\, x := [e]\,],\ (h,s)\rangle \xrightarrow[emp]{\mathbf{u}} \mathsf{abort} \quad \text{otherwise}$$

$$\langle \mathbf{E}[\, [e] := e'\,],\ (h,s)\rangle \xrightarrow[(\emptyset,\{\ell\})]{\mathbf{u}} \langle \mathbf{E}[\,\mathbf{skip}\,],\ (h',s)\rangle \quad \text{if } [\![e]\!]_s = \ell,\ [\![e']\!]_s = n,\ \ell \in \mathsf{dom}(h), \text{ and } h' = h[\ell \rightsquigarrow n]$$

$$\langle \mathbf{E}[\, [e] := e'\,],\ (h,s)\rangle \xrightarrow[emp]{\mathbf{u}} \mathsf{abort} \quad \text{otherwise}$$

$$\langle \mathbf{E}[\, x := e\,],\ (h,s)\rangle \xrightarrow[emp]{\mathbf{u}} \langle \mathbf{E}[\,\mathbf{skip}\,],\ (h,s')\rangle \quad \text{if } [\![e]\!]_s = n \text{ and } s' = s[x \rightsquigarrow n]$$

$$\langle \mathbf{E}[\,\mathbf{skip}; c\,],\ \sigma\rangle \xrightarrow[emp]{\mathbf{u}} \langle \mathbf{E}[\, c\,],\ \sigma\rangle \quad \text{always}$$

$$\ldots \qquad \ldots$$

$$\langle \mathbf{E}[\,\mathbf{dispose}(e)\,],\ (h,s)\rangle \xrightarrow[(\emptyset,\{\ell\})]{\mathbf{o}} \langle \mathbf{E}[\,\mathbf{skip}\,],\ (h',s)\rangle \quad \text{if } [\![e]\!]_s = \ell,\ \ell \in \mathsf{dom}(h),\ h' = h \backslash \{\ell\}$$

$$\langle \mathbf{E}[\,\mathbf{dispose}(e)\,],\ (h,s)\rangle \xrightarrow[emp]{\mathbf{o}} \mathsf{abort} \quad \text{otherwise}$$

$$\langle \mathbf{E}[\, x := \mathbf{cons}(e_1,\ldots,e_k)\,],\ (h,s)\rangle \xrightarrow[(\emptyset,ws)]{\mathbf{o}} \langle \mathbf{E}[\,\mathbf{skip}\,],\ (h',s')\rangle$$
$$\text{if } ws = \{\ell,\ldots,\ell+k-1\},\ ws \cap \mathsf{dom}(h) = \emptyset,\ [\![e_i]\!]_s = n_i,\ s' = s[x \rightsquigarrow \ell] \text{ and } h' = h[\ell \rightsquigarrow n_1,\ldots,\ell+k-1 \rightsquigarrow n_k]$$

$$\langle c,\sigma\rangle \xrightarrow[\delta]{} \langle c',\sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{u}} \langle c',\sigma'\rangle \text{ or } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \langle c',\sigma'\rangle$$

$$\langle c,\sigma\rangle \xrightarrow[\delta]{} \mathsf{abort} \quad \text{if } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{u}} \mathsf{abort} \text{ or } \langle c,\sigma\rangle \xrightarrow[\delta]{\mathbf{o}} \mathsf{abort}$$

Fig. 3. Sequential footprint semantics

$$\langle \mathbf{T}[\, c\,],\ \sigma\rangle \longmapsto \langle \mathbf{T}[\, c'\,],\ \sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \longrightarrow \langle c',\sigma'\rangle$$

$$\langle \mathbf{T}[\, c\,],\ \sigma\rangle \longmapsto \mathsf{abort} \quad \text{if } \langle c,\sigma\rangle \longrightarrow \mathsf{abort}$$

$$\langle \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,],\ \sigma\rangle \longmapsto \langle \mathbf{T}[\,\mathbf{E}[\,\mathbf{skip}\,]\,],\ \sigma'\rangle \quad \text{if } \langle c,\sigma\rangle \longrightarrow^* \langle \mathbf{skip},\ \sigma'\rangle$$

$$\langle \mathbf{T}[\,\mathbf{E}[\,\mathbf{atomic}\ c\,]\,],\ \sigma\rangle \longmapsto \mathsf{abort} \quad \text{if } \langle c,\sigma\rangle \longrightarrow^* \mathsf{abort}$$

$$\langle \mathbf{T}[\,\mathbf{E}[\, c_1 \parallel c_2\,]\,],\ \sigma\rangle \longmapsto \langle \mathbf{T}[\,\langle\!\langle c_1, c_2\rangle\!\rangle \mathbf{E}[\,\mathbf{skip}\,]\,],\ \sigma\rangle \quad \text{always}$$

$$\langle \mathbf{T}[\,\langle\!\langle \mathbf{skip}, \mathbf{skip}\rangle\!\rangle c\,],\ \sigma\rangle \longmapsto \langle \mathbf{T}[\, c\,],\ \sigma\rangle \quad \text{always}$$

Fig. 4. Interleaving semantics of concurrent programs

Figure 4 defines the interleaving semantics of concurrent programs. Following Vafeiadis and Parkinson [29], 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 are no atomic blocks or parallel compositions in c.

Next we give a simple example to show the use of contexts and thread trees.

Example 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 $\langle\!\langle c_1, c_2\rangle\!\rangle(\mathbf{skip}; c')$. Then the $\mathbf{T}'$ for the next step can be either $\langle\!\langle [\,], c_2\rangle\!\rangle(\mathbf{skip}; c')$ or $\langle\!\langle c_1, [\,]\rangle\!\rangle(\mathbf{skip}; c')$.

$$[\Lambda]\ \langle T, \sigma\rangle \longmapsto \langle T', \sigma'\rangle \quad \text{if } \exists T''.\ (T, T'') \in \Lambda \wedge \langle T'', \sigma\rangle \longmapsto \langle T', \sigma'\rangle$$

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

Fig. 5. Semantics parameterized over Λ

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 5 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:

$$\Lambda \in \mathcal{P}(\mathit{ThrdTree} \times \mathit{ThrdTree})$$

The semantics follows the interleaving semantics in Fig. 4, except that the current thread tree can be replaced at any given step by another thread tree related through the Λ relation. Λ is supposed to provide a set of thread trees that are equivalent to the current thread tree with some notion of equivalence. This Λ-based semantics chooses nondeterministically which command will execute.

Naturally, different instantiations of Λ yield different semantics. As one can see, this semantics is trivially equivalent to the interleaving semantics shown in Fig. 4 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;
2. but permits the rewriting of non-synchronized sequential portions while preserving their *sequential* semantics.

The intuition is that programs should be well-synchronized to avoid unexpected behaviors in relaxed memory models. 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 either since

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

$\langle c, \sigma\rangle$	$\xrightarrow[emp]{}{}^{0}$	$\langle c, \sigma\rangle$	always
$\langle c, \sigma\rangle$	$\xrightarrow[\delta]{}{}^{k+1}$	$\langle c', \sigma'\rangle$	if there exist c'', σ'', δ', and δ'' such that $\delta = \delta' \cup \delta''$, $\langle c, \sigma\rangle \xrightarrow[\delta']{} \langle c'', \sigma''\rangle$ and $\langle c'', \sigma''\rangle \xrightarrow[\delta'']{}{}^{k} \langle c', \sigma'\rangle$

Fig. 6. Multi-step sequential transitions

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

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 — after performing a sequential big step — c_1 aborts only if c_2 aborts; or, if c_1 completes, then c_2 either aborts or takes a big step that ends in the same state. Also, if c_1 at the end of the big step terminates (**skip** case) or reaches a synchronization point (cases for thread fork and join, atomic blocks, **cons** and **dispose**), there must be a corresponding termination or synchronization point at the end of the big step taken by c_2 and the remaining parts (if any) of c_1 and c_2 still satisfy the relation. We use indices in the definition since $\mathbf{E}_1[\,\mathbf{skip}\,]$ in the 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. 2.

Properties of subsumption. Suppose c_1 and c_2 are sequential programs consisting of unordered operations only, and $c_1 \preceq c_2$. For any input state we have 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 [10] (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 Sect. 4 to show the expressiveness of the subsumption relation and how it models behaviors of programs in relaxed memory models. More properties about the relation are shown by the following two lemmas.

Lemma 3.2. The relation $\preceq$ is reflexive and transitive.

Lemma 3.3. If $c_1 \preceq c_2$, then, for all contexts $\mathcal{C}$, $\mathcal{C}[\,c_1\,] \preceq \mathcal{C}[\,c_2\,]$.

Here $\mathcal{C}$ can be any context, i.e. a program with a hole in it. It does not have to be **E** or **T**.

3.3 Relaxed Semantics

The subsumption relation can be lifted for thread trees.

Definition 3.4. We define the binary relation $\preceq_{\mathbf{t}}$ for thread trees.

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

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

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

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

On the other hand, as we show in Lemma 3.5, 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 3.6 shows the abort case.

Lemma 3.5. $[\succeq_{\mathbf{t}}]\ \langle T, \sigma\rangle \longmapsto^* \langle \mathbf{skip}, \sigma'\rangle$ iff there exists a T' such that $T \succeq_{\mathbf{t}} T'$ and $\langle T', \sigma\rangle \longmapsto^* \langle \mathbf{skip}, \sigma'\rangle$.

Lemma 3.6. $[\succeq_{\mathbf{t}}]\ \langle T, \sigma\rangle \longmapsto^* \mathsf{abort}$ 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 Sect. 5, after we formally define data-race-freedom.

4 Examples

There are different aspects that characterize a particular memory model. In this section, we show how they are reflected in our semantics. The examples are shown with the following naming convention: v_1, v_2, v_3, etc, are thread-local variables that hold values; x, y, z, etc, are variables that hold memory addresses.

Data dependencies. At first glance, the definition of $\succeq$ is too restrictive since it quantifies over *all* input states. It does not allow

$$(\,[x] := 1; v_1 := [y]\,) \succeq (v_1 := [y];\, [x] := 1)\,,$$

where the data dependency of the two statements depends on the runtime values of x and y. However, the $\succeq$ relation allows the following transformation:

$$[x] := 1; v_1 := [y] \succeq \mathbf{if}\ x = y\ \mathbf{then}\ (\,[x] := 1; v_1 := [x]\,)\ \mathbf{else}\ (v_1 := [y];\, [x] := 1),$$

where we insert a dynamic test to see if x is an alias of y. So we do allow reordering of memory accesses that do not have data dependencies at runtime.

Memory reordering. It is easy to see that the $\succeq$ relation supports all four types of memory reordering (R,W $\rightarrow$ R,W). In the example below,

$$(v_1 := [x]; [y] := 1) \quad \| \quad (v_2 := [y]; [x] := 1) \quad ,$$

we can get $v_1 = v_2 = 1$ if $x \neq y$. This can be achieved by reordering the commands in the second thread (not supported by Boudol and Petri [7]),

$$v_2 := [y]; [x] := 1 \succeq \textbf{if } x = y \textbf{ then } (v_2 := [x]; [x] := 1) \textbf{ else } ([x] := 1; v_2 := [y]) .$$

Write atomicity. Write atomicity is not preserved by the $\succeq$ relation. In the classic cross-over example below,

$$([x] := 1; v_1 := [x]) \quad \| \quad ([x] := 2; v_2 := [x]) \quad ,$$

we can get $v_1 = 2$ and $v_2 = 1$. This is achieved by adding a redundant write in the right hand side thread: $[x] := 2; v_2 := [x] \succeq [x] := 2; v_2 := [x]; [x] := 2$. This simulates the duration between the beginning and the end of the write. We may also store arbitrary values to memory before completing. For instance, the program below allows $v_1 = 33$ at the end.

$$v_1 := [x] \quad \| \quad [x] := 1$$

It happens with the following transformation of the right hand side thread:

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

which means the memory value is undefined until the write completes. This is commonly referred to as "out-of-thin-air" behavior. A similar behavior shows up when we have simultaneous writes to the same location:

$$(v_1 := 1; [x] := v_1) \quad \| \quad [x] := 2 \quad .$$

In this case, the final value of $[x]$ could be arbitrary. It could be 3 if we do the following transformation of the left hand side thread:

$$v_1 := 1; [x] := v_1 \succeq [x] := 0; v_1 := [x]; v_1 := v_1 + 1; [x] := v_1 .$$

Strong barrier. In the relaxed semantics, we can enforce both atomicity and ordering by using **atomic** c. A memory fence MF can be implemented by **atomic skip**. The following examples show that MF is not sufficient to enforce program orders when there is no cache coherence. In the example below,

$$[x] := 1 \quad \| \quad [x] := 2 \quad \| \quad \begin{pmatrix} v_1 := [x]; \\ \mathsf{MF}; \\ v_2 := [x] \end{pmatrix} \quad \| \quad \begin{pmatrix} v_3 := [x]; \\ \mathsf{MF}; \\ v_4 := [x] \end{pmatrix}$$

We can get the outcome $v_1 = v_4 = 1$ and $v_2 = v_3 = 2$ by rewriting the leftmost thread: $[x] := 1 \succeq [x] := 1; [x] := 1$.

See also the independent-reads-independent-writes (IRIW) example:

$$[x] := 1 \quad \| \quad [y] := 1 \quad \| \quad \begin{pmatrix} v_1 := [x]; \\ \mathsf{MF}; \\ v_2 := [y] \end{pmatrix} \quad \| \quad \begin{pmatrix} v_3 := [y]; \\ \mathsf{MF}; \\ v_4 := [x] \end{pmatrix}$$

where the behavior $v_1 = v_3 = 1$ and $v_2 = v_4 = 0$ is permissible if we rewrite the leftmost thread through $[x] := 1 \succeq [x] := 1; [x] := 0; [x] := 1$.

Race-free programs. Race-free programs do not have unexpected behaviors in our semantics (see the DRF-guarantee in Sect. 5). In the example below:

$$\begin{pmatrix} v_1 := [x]; \\ \textbf{if } v_1 = 1 \textbf{ then } [y] := 1 \end{pmatrix} \quad \| \quad \begin{pmatrix} v_2 := [y]; \\ \textbf{if } v_2 = 1 \textbf{ then } [x] := 1 \end{pmatrix}$$

the only behavior allowed is $v_1 = v_2 = 0$. Because the two conditional statements cannot be reached (assuming $[x] = [y] = 0$ and $x \neq y$ initially), the program never issues a memory write. So the program is race-free. Also, transformations allowed by the $\succeq$ relation cannot introduce races by inserting redundant writes. This is guaranteed by the fact that the footprints of both threads are disjoint, and they cannot increase after transformations.

Compiler optimizations (and obfuscations). Redundant memory reads and writes can be introduced and eliminated, as shown by the following examples:

$$\begin{aligned} v_1 := [x]; v_2 := 1 &\succeq v_1 := [x]; v_2 := [x]; v_2 := 1 \\ v_1 := [x]; v_2 := [x] &\succeq v_1 := [x]; v_2 := v_1 \\ [x] := v_1 &\succeq [x] := 1; [x] := v_1 \\ [x] := 1; [x] := v_1 &\succeq [x] := v_1 \end{aligned}$$

Furthermore, we can eliminate dead memory operations and reduce the memory footprint: $v_1 := [x]; v_1 := 1 \succeq v_1 := 1$. Note that the reverse is not true: $\neg(v_1 := 1 \succeq v_1 := [x]; v_1 := 1)$. A transformation cannot increase the footprint.

Now we can reproduce the prescient-write example:

$$(v_1 := [x]; [x] := 1) \quad \| \quad (v_2 := [x]; [x] := v_2)$$

where we could have $v_1 = v_2 = 1$ by rewriting the left hand side thread:

$$v_1 := [x]; [x] := 1 \succeq v_1 := [x]; [x] := 1; [x] := v_1; v_1 := [x]; [x] := 1\,.$$

Other optimizations, including instruction scheduling, register allocation, algebraic transformations and control transformations, can also be supported. More examples can be found in the technical report [15].

Total store ordering. 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 [28]. TSO allows write-to-read reordering. It enforces cache-coherence, but allows a thread to read its own writes earlier.

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

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

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

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

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

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

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

Fig. 7. TSO

We define $\succeq_{\text{TSO}}$, an instantiation of Λ, in Fig. 7. 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 [24] 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 1. $\succeq_{\text{TSO}}$ is a subset of $\succeq$.

Partial Store Ordering (PSO). In our technical report [15], we define $\succeq_{\text{PSO}}$, another instantiation of Λ that yields the PSO model [28]. It is defined by simply adding a couple of rules to $\succeq_{\text{TSO}}$ to support write-to-write reordering.

5 Grainless Semantics and DRF Guarantee

Following Reynolds [25] and Brookes [8], here we give a grainless semantics to our language, which is operational instead of being a 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 the DRF-guarantee of our relaxed semantics.

5.1 Grainless Semantics

Below we first instrument thread trees with footprints for threads. Execution contexts $\widetilde{\mathbf{T}}$ in the instrumented trees are defined similarly to $\mathbf{T}$ in Sect. 2.

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

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. 8 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. 9, which refers to the sequential transitions defined in Figs. 3 and 6. In this semantics we execute unordered commands in a big step, as shown in the first rule (see Fig. 6 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.

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

Fig. 8. Auxiliary definitions

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

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

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

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

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

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

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

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

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

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

Fig. 9. Grainless semantics

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, which has not been explained yet, allows an intermediate footprint δ' to be recorded on the thread tree before the big step transition of unordered commands finishes. This is necessary to characterize the following program as one with data-races:

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

The first rule does not apply here because both threads diverge, but we can apply the second rule to record the write set $\{x\}$ on the thread tree and then apply the fourth rule to detect the race. Note that this rule does not change the command c or the state σ. If we ignore the footprint, it simply adds stuttering steps in the semantics. The side condition $\delta \subset \delta'$ (defined in Fig. 2) ensures 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 transition of this thread. In the second case, the last step must be an atomic operation and δ must be *emp*.

Following Reynolds' principles for grainless semantics [25], both abort and race are viewed as bad program configurations. We distinguish race from abort to define data-race-freedom. A thread tree T is race-free if its execution never

leads to race. By this definition, programs that abort may still be race-free. This allows us to talk about race-free but unsafe programs, as shown in Theorem 5.2.

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.

Example 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

Theorem 5.2 formulates the DRF-guarantee of the relaxed semantics. It says a race-free program configuration has the same observable behaviors in both the relaxed semantics and the interleaving semantics: if it aborts in one semantics, it aborts in the other; if it never aborts (which means it is "safe"), it reaches the same set of final states in both settings. We need the premise in the second case because the subsumption relation allows us to transform an unsafe program into a safe one. Therefore a program that reaches $\langle \mathbf{skip}, \sigma'\rangle$ in the relaxed semantics may abort and never terminate at σ' in the interleaving semantics.

Theorem 5.2 (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 $\langle T, \sigma\rangle \longmapsto^* \langle \mathbf{skip}, \sigma'\rangle$.

The proof trivially follows from two important lemmas. Lemma 5.3 shows the equivalence between the interleaving semantics and the grainless semantics for race-free programs. Lemma 5.4 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 (see Fig. 1 (b)).

Lemma 5.3. 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 5.4. 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 $\langle\lfloor T\rfloor, \sigma\rangle \Longrightarrow^* \langle(\mathbf{skip}, \_), \sigma'\rangle$.

Details about the proofs can be found in the technical report [15].

$$\frac{\vdash \{p * I\}\, c\, \{q * I\}}{I \vdash \{p\}\, \textbf{atomic}\ c\, \{q\}}\ (\textsc{atom}) \qquad \frac{I \vdash \{p_1\}\, c_1\, \{q_1\} \quad I \vdash \{p_2\}\, c_2\, \{q_2\}}{I \vdash \{p_1 * p_2\}\, c_1 \parallel c_2\, \{q_1 * q_2\}}\ (\textsc{par})$$

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

Fig. 10. Selected CSL Rules

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 [23, 9]. It consists of sequential and concurrent rules. The sequential part ($\vdash \{p\}\, c\, \{q\}$) is standard sequential separation logic rules. The concurrent rules allow us to derive a judgment of the form $I \vdash \{p\}\, c\, \{q\}$. It 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. Figure 10 shows two of the most important rules of CSL.

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 6.2.

Definition 6.1. $I \models \{p\}c\{q\}$ iff, for all σ and δ such that $\sigma \models I * p$ and $\sigma \models \delta \uplus I$, we have (1) $\neg(\langle (c, \delta), \sigma\rangle \Longrightarrow^* \mathsf{abort})$ and $\neg(\langle (c, \delta), \sigma\rangle \Longrightarrow^* \mathsf{race})$, and, (2) if $\langle (c, \delta), \sigma\rangle \Longrightarrow^* \langle (\textbf{skip}, \_), \sigma'\rangle$, then $\sigma' \models I * q$.

Here $\sigma \models I * p$ means σ satisfies the assertion $I * p$, and $\sigma \models \delta \uplus I$ means the set of memory locations in δ is disjoint with the domain of the sub-heap (in σ) that satisfies I. The formal definitions are given in the technical report [15].

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

The proof of this lemma follows standard techniques, i.e. we need to first prove the locality [31, 10] of each primitive commands. We show details of the proofs in our technical report [15]. Next we give semantics to $I \vdash \{p\}\, c\, \{q\}$ based on our relaxed semantics, and show the soundness in Theorem 6.4.

Definition 6.3. $I \models_{[\Lambda]} \{p\}c\{q\}$ iff, for all σ such that $\sigma \models I * p$, we have (1) $\neg([\Lambda]\ \langle c, \sigma\rangle \longmapsto^* \mathsf{abort})$, and, (2) if $[\Lambda]\ \langle c, \sigma\rangle \longmapsto^* \langle \textbf{skip}, \sigma'\rangle$, then $\sigma' \models I * q$.

Theorem 6.4. If $I \vdash \{p\}\, c\, \{q\}$, then $I \models_{[\succeq_t]} \{p\}c\{q\}$.

Proof. Trivial by applying Lemmas 5.4 and 6.2. □

Extensions of CSL. Bornat et al. [6] extended CSL with fractional permissions to distinguish exclusive total accesses and shared read-only accesses. We can prove 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. The proof should be similar to the proof for standard CSL. We believe other extensions of CSL, such as CSL with storable locks [17, 18] and the combination of CSL with Rely-Guarantee reasoning [29, 13], can also be proved sound with respect to the grainless semantics. Then their soundness in our relaxed semantics can be derived easily from Lemma 5.4. We would like to verify our hypothesis in our future work.

7 Related Work and Conclusions

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. [26] 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: $c \succeq \textbf{if } q \textbf{ then } c' \textbf{ else } c$, 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 [26] involving LI (e.g., the cross-over example) can be supported following the pattern with which we reproduce the prescient-write example in Sect. 4.

In this paper, we do not investigate the precise connection to the Java Memory Model (JMM [21]). 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. [11]. 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 [7] 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 reordering 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 [27] 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 [16] 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.

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.

Acknowledgments. We thank anonymous referees for their comments on this paper. Rodrigo Ferreira and Zhong Shao are supported in part by NSF grants CCF-0811665, CNS-0915888, and CNS-0910670. Xinyu Feng is supported in part by National Natural Science Foundation of China (grant No. 90818019).

References

[1] Adve, S., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
[2] Adve, S., Hill, M.: A unified formalization of four shared-memory models. IEEE Transactions on Parallel and Distributed Systems 4(6), 613–624 (1993)
[3] Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: 31st POPL, January 2004, pp. 14–25 (2004)
[4] Boehm, H., Adve, S.: The foundations of the C++ concurrency memory model. In: PLDI, Tucson, Arizona, June 2008, pp. 68–78 (2008)
[5] Boehm, H.-J.: Threads cannot be implemented as a library. In: PLDI, Chicago, June 2005, pp. 261–268 (2005)
[6] Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: 32nd POPL, January 2005, pp. 259–270 (2005)
[7] Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: 36th POPL, Savannah, Georgia, USA, January 2009, pp. 392–403 (2009)
[8] Brookes, S.: A grainless semantics for parallel programs with shared mutable data. Electronic Notes in Theoretical Computer Science 155, 277–307 (2006)
[9] Brookes, S.: A semantics for concurrent separation logic. Theoretical Comp. Sci. 375(1-3), 227–270 (2007)

[10] Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: 22nd LICS, July 2007, pp. 366–378 (2007)
[11] Cenciarelli, P., Knapp, A., Sibilio, E.: The Java memory model: Operationally, denotationally, axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
[12] Dijkstra, E.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages, pp. 43–112. Academic Press, London (1968)
[13] Feng, X.: Local rely-guarantee reasoning. In: 36th POPL, January 2009, pp. 315–327 (2009)
[14] Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)
[15] Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic (extended version). Technical Report YALEU/DCS/TR-1422, Department of Computer Science, Yale University (2009), `http://flint.cs.yale.edu/publications/rmm.html`
[16] Gao, G., Sarkar, V.: Location consistency – a new memory model and cache consistency protocol. IEEE Transactions on Computers 49(8), 798–813 (2000)
[17] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
[18] Hobor, A., Appel, A., Nardelli, F.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
[19] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28(9) (September 1979)
[20] Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: 33rd POPL, January 2006, pp. 42–54 (2006)
[21] Manson, J., Pugh, W., Adve, S.: The Java memory model. In: 32nd POPL, Long Beach, California, January 2005, pp. 378–391 (2005)
[22] Mosberger, D.: Memory consistency models. Operating Systems Review 27(1), 18–26 (1993)
[23] O'Hearn, P.: Resources, concurrency, and local reasoning. Theoretical Comp. Sci. 375(1-3), 271–307 (2007)
[24] Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: 22nd TPHOLS, Munich, Germany, August 2009, pp. 391–407 (2009)
[25] Reynolds, J.: Toward a grainless semantics for shared-variable concurrency. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 35–48. Springer, Heidelberg (2004)
[26] Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: 12th PPoPP, San Jose (March 2007)
[27] Sevcik, J.: Program Transformations in Weak Memory Models. PhD thesis, School of Informatics, University of Edinburgh (2008)
[28] SPARC International Inc. The SPARC Architecture Manual, Version 8. Revision SAV080SI9308 (1992)
[29] Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
[30] Yang, H.: Relational separation logic. Theoretical Computer Science 375(1-3), 308–334 (2007)
[31] Yang, H., O'Hearn, P.: A semantic basis for local reasoning. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 402–416. Springer, Heidelberg (2002)