

Program Transformations in Weak Memory Models

Jaroslav Ševčík

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2008

Abstract

We analyse the validity of common optimisations on multi-threaded programs in two memory models—the DRF guarantee and the Java Memory Model. Unlike in the single-threaded world, even simple program transformations, such as common subexpression elimination, can introduce new behaviours in shared-memory multi-threading with an interleaved semantics. To validate such optimisations, most current programming languages define weaker semantics, called memory models, that aim to allow such transformations while providing reasonable guarantees. In this thesis, we consider common program transformations and analyse their safety in the two most widely used language memory models: (i) the DRF guarantee, which promises sequentially consistent behaviours for data race free programs, and (ii) the Java Memory Model, which is the semantics of multi-threaded Java. The DRF guarantee is the semantics of Ada and it has been proposed as the semantics of the upcoming revision of C++. Our key results are: (i) we prove that a large class of elimination and reordering transformations satisfies the DRF guarantee; (ii) we find that the Java language is more restrictive—despite the claims in the specification, the Java Memory Model does not allow some important program transformations, such as common subexpression elimination.

To establish the safety results, we develop a trace semantic framework and describe important program optimisations as relations on sets of traces. We prove that all our elimination and reordering transformations satisfy the DRF guarantee, i.e., the semantic transformations cannot introduce new behaviours for data race free programs. Moreover, we prove that all the transformations preserve data race freedom. This ensures safety of transformations composed from eliminations and reorderings. In addition to the DRF guarantee, we prove that for arbitrary programs, our transformations prevent values appearing “out-of-thin-air”—if a program does not contain constant c and does not perform any operation that could create c , then no transformation of the program can output c . We give an application of the semantic framework to a concrete language and prove safety of several simple syntactic transformations.

We employ similar semantic techniques to prove validity of several classes of transformations, such as the elimination of an overwritten write or reordering of independent memory accesses, in the Java Memory Model. To establish the

negative results for the Java Memory Model, we provide counterexamples showing that several common optimisations can introduce new behaviours and thus are not safe.

Acknowledgements

First of all, I would like to thank my supervisor, David Aspinall, for his guidance and support in academic matters. He provided an environment where I could concentrate on research without worrying about financial or administrative aspects of my studies.

I owe many thanks to my second supervisor, Colin Stirling, for the invaluable discussions during the early stages of my research.

Many thanks go to the members of the Mobility and Security research group for providing a wonderful environment. Namely, to Ian Stark for asking the right questions on my progress report panels, and to Bob Atkey for interesting conversations and organising the group meetings and talks.

I am grateful to Doug Lea for his encouragement, interest in my work and thought-provoking insights into the practical aspects of concurrency. I also wish to thank Jeremy Manson and Sarita Adve for their feedback on the Java Memory Model. I would like to thank Marieke Huisman and Gustavo Petri for inviting me to INRIA and for the valuable and exhausting discussions we had there.

I am very grateful to my friends and family for their support through the three years. Finally, I would like to thank my wife, for everything.

This work was supported by a PhD studentship awarded by the UK EPSRC, grant EP/C537068. I also acknowledge the support of the EU project MOBIUS (IST-15905).

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Jaroslav Ševčík)

Table of Contents

1	Introduction	1
1.1	DRF guarantee	2
1.2	Java Memory Model	4
1.3	Aims	5
1.4	Transformations by Example	6
1.4.1	Trace-preserving Transformations	6
1.4.2	Reordering	6
1.4.3	Redundant Read Elimination	7
1.4.4	Irrelevant Read Elimination	7
1.4.5	Irrelevant Read Introduction	8
1.4.6	Redundant Write Elimination	8
1.4.7	Roach-motel Reordering	9
1.4.8	Reordering with External Actions	9
1.5	Summary of Results and Outline of Contents	9
1.5.1	Overview	10
2	Preliminaries	13
2.1	Actions, Traces and Interleavings	13
2.1.1	Interleavings and Executions	18
2.1.2	Orders on Actions	20
2.1.3	Matchings	22
2.2	Data Race Freedom	25
2.2.1	Alternative DRF	27
3	Safety of Transformations	33
3.1	Trace-semantic Transformations	34
3.1.1	Eliminations	34

3.1.2	Reordering	38
3.2	Safety of Transformations	43
3.2.1	Elimination	45
3.2.2	Reordering	55
3.2.3	Out-of-thin-air	66
3.3	High-level Remarks about Roach Motel Reordering	68
4	Syntax to Semantics	69
4.1	Language	69
4.2	Elimination	74
4.3	Reordering	80
4.4	Out-of-thin-air	90
5	Java Memory Model	91
5.1	Introduction	91
5.2	JMM Operationally	92
5.2.1	JMM actions and orders	92
5.2.2	Committing semantics	96
5.2.3	Example	101
5.3	Invalid Transformations	102
5.3.1	Redundant Write after Read Elimination	103
5.3.2	Redundant Read after Read Elimination	103
5.3.3	Roach Motel Semantics	104
5.3.4	Irrelevant Read Introduction	105
5.3.5	Reordering with external actions	106
5.4	Valid Transformations	107
5.4.1	Proof of Validity	108
5.5	Practical Impact	114
6	Conclusion	117
6.1	Towards Applications	117
6.2	Related Work	118
6.2.1	Compilers	118
6.2.2	Other Models of Concurrency	120
6.2.3	Semantics of Concurrency	120
6.2.4	Hardware Memory Models	122

6.3	Future Work and Open Questions	123
6.3.1	Transformations	123
6.3.2	Richer Trace Semantics	126
6.3.3	Termination Properties	127
6.3.4	Formal Techniques	128
	Bibliography	129

Chapter 1

Introduction

When programmers write programs in languages for multi-threaded shared memory, they intuitively assume that the computer will execute their program *sequentially consistently* (Lamport, 1979). That is, it should appear as though some interleaving of the operations performed by the individual threads in the program was executed.

However, modern compilers and multi-processors perform optimisations that break this illusion of sequential consistency. Consider the program

```
initially requestReady = responseReady = data = 0
-----
Thread 1 | Thread 2
-----|-----
data := 1 | if (requestReady == 1) {
requestReady := 1 |     data := 2
if (responseReady == 1) |     responseReady := 1
    print data | }
-----|-----
```

and assume that all shared locations, i.e., `data`, `requestReady` and `responseReady`, are initialised to 0. While this program cannot output value 1 in any interleaving, optimising compilers¹ often reuse the constant 1 assigned to `data` in Thread 1 and replace the `print data` statement with `print 1`. This makes the output of 1 possible and breaks the illusion of sequential consistency. Perhaps surprisingly, this transformation is perfectly legal in many languages, including C/C++ and Java, although the specific details vary.

The C and C++ specifications do not specify behaviours of multi-threaded programs at all (Kernighan and Ritchie, 1988; Stroustrup, 2000). The Java language defines behaviours of concurrent programs, but the specification is delib-

¹For example, GCC 4.1.2 on the x86 architecture.

erately weaker than the interleaved semantics (Gosling et al., 2005a). Such weak semantics are often called weak memory models, or relaxed memory models.

It is notoriously known that reasoning about concurrent programs is hard even when assuming an interleaved semantics. Why would one use a weaker model? The main reason is performance. Many important compiler optimisations for sequential programs, including common subexpression elimination and various loop optimisations, are not valid under sequential consistency. In addition to the compiler optimisations, modern processors aggressively optimise memory accesses at runtime. Although the processors provide instructions for preventing the optimisations, these instructions are often the most expensive ones in the instruction set—they take hundreds of cycles in the current generation of processors. Moreover, prohibiting transformations that violate sequential consistency would affect single-threaded programs too, because separate compilation and dynamic loading of libraries make it almost impossible for a compiler to decide whether a program has multiple threads.

To resolve the conflict between performance and ease of reasoning, researchers suggested a weaker contract that allows many optimisations while providing a reasonable semantics: Instead of guaranteeing sequential consistency for all programs, specifications of several modern programming languages guarantee sequential consistency only for data race free programs (Gharachorloo et al., 1990; Adve and Aggarwal, 1993). Such a contract is called the DRF guarantee.

1.1 DRF guarantee

Intuitively, a program is data race free if there are no concurrent accesses to the same shared location from two different threads, such that at least one of the accesses is a write. More precisely, a program is *data race free* if there is no interleaving that contains two conflicting memory accesses from different threads such that there is no synchronisation between them. A synchronisation is either a *lock* or an *unlock*. Two memory accesses are *conflicting* if they access the same location and at least one of them is a write. We give a formal definition in Section 2.2. Here, we illustrate data race freedom on examples.

Example 1.1. Figure 1.1 shows two programs that use shared variables x and y , and thread-local registers $r1$ and $r2$. The program A is not data race free: for example, the accesses to y are not separated by any synchronisation in the

		initially $x = y = 0$	
		lock m1	lock m2
		r1:=x	r2:=y
		unlock m1	unlock m2
		lock m2	lock m1
		y:=r1	x:=1
		unlock m2	unlock m1

A. (allowed)

B. (prohibited)

Is it possible to get $r1 = r2 = 1$ at the end of an execution?

Figure 1.1: Examples of legal and illegal executions.

interleaving $r1:=x$, $r2:=y$, $y:=r1$, $x:=1$. The DRF guarantee semantics does not specify behaviours of such programs and any behaviour is valid. In particular, it is possible obtain value 1 in register $r2$ under the DRF guarantee, although this is an impossible result under sequential consistency.

On the other hand, if we protect variable x with monitor $m1$ and variable y with monitor $m2$, as shown by program B from Figure 1.1, we make the program data race free. Therefore, the DRF guarantee implies that $r2$ must be 0, because this is the only possible value under sequential consistency.

Why is the DRF guarantee a good interface between programmers, compilers and hardware? On the programmers' side, it is a good practice to write data race free programs and there are numerous run-time and compile-time tools that aid programmers in detecting and removing data races. For example, Bacon et al. (2000), Flanagan and Freund (2000) and Boyapati and Rinard (2001) designed and implemented expressive type systems that guarantee data race freedom. Naik et al. (2006) developed a powerful static analysis for finding data races without any annotations. Finally, there are several runtime tools for detecting data races (Savage et al., 1997; Elmas et al., 2007). One might argue that not all programmers wish to write data race free programs, because they utilise efficient lock-free data structures with data races (Herlihy and Shavit, 2008). To cater to the needs of such advanced programmers, languages provide a *volatile*² shared-memory lo-

²This modifier in Java has a different meaning from the C++ *volatile*. See Example 2.6 in Chapter 2 for details.

cation modifier and allow data races on volatile locations. More precisely, if all data races in a program happen only on volatile locations then the compiler still guarantees the illusion of sequential consistency. For example, marking the locations `x` and `y` in the program A from Figure 1.1 as volatile would prevent the result `r2 = 1` under the DRF guarantee.

Although the DRF guarantee provides a strong semantics for data race free programs, it allows arbitrary behaviours for programs with data races. For inherently unsafe languages, such as C++, this may be an acceptable specification. Indeed, the upcoming version of C++ adopts the DRF guarantee as its memory model (Boehm and Adve, 2008). Whereas C++ places the burden of maintaining data race freedom on programmers, Ada precludes data races³ in its type system (Ada95, 1995). Therefore, the DRF guarantee coincides with sequential consistency in Ada. Java (Gosling et al., 2005a) takes a different approach—it defines a rather complex semantics of multi-threaded programs, called the Java Memory Model (JMM), so that the DRF guarantee is one of the consequences of the JMM (Manson et al., 2005). We discuss the motivations for the JMM in the next section.

From the perspective of compilers and computer hardware, the DRF guarantee is a good contract, because it allows many common optimisations. This was first observed in the computer architecture community (Gharachorloo et al., 1990; Adve and Hill, 1990; Adve and Aggarwal, 1993) and it is believed that most compiler transformations preserve the DRF guarantee, but we are not aware of any previous formal study of the DRF guarantee for program transformations performed by compilers. Such a study of program transformations is the main contribution of this thesis.

1.2 Java Memory Model

The Java Memory Model defines the semantics of Java programs with threads (Gosling et al., 2005a). The semantics guarantees sequential consistency for data race free programs. However, the behaviours of programs with data races cannot be left undefined, because Java promises to enforce strong security properties for

³In fact, Ada 83 allowed data races and was perhaps the first language to adopt the DRF guarantee as its memory model (Ledgard, 1983) in its type system. The Ada 95 revision of the language enforces data race freedom statically.

arbitrary programs. For example, Java applets in web pages should not have any access to the file system of its host computer even when they contain data races.

Interestingly, the Java specification does not give any precise statement of the guarantee for programs with data races. Instead, it shows some disallowed behaviours on examples, and it ensures that these behaviours are indeed prevented by the JMM. This guarantee is called the *out-of-thin-air* guarantee, because the examples show a program and argue that some value cannot appear “out-of-thin-air” in one of the program’s local variables. For example, consider the program

$$\frac{\text{initially } x = y = 0}{\begin{array}{l|l} r1 := x & r2 := y \\ y := r1 & x := r2 \end{array}}$$

and note that no value other than 0 appears in the program. Moreover, the program does not contain any arithmetic that could create any new values. As a result, the registers `r1` and `r2` should always contain value 0. Any other value would be created “out-of-thin-air”.

The main idea behind the Java Memory Model’s definition of allowable behaviours of a program is the causal order on the program’s data races. To show that a program may have a certain behaviour, one must provide the outcomes of its data races with a preorder on the data races, and justify that the outcome of each data race only depends on the outcomes of the earlier data races in the preorder. The preorder essentially represents a causality order on the program’s data races. The precise definition is quite subtle. We explain the details of the justification procedure in Chapter 5. One of the contributions of this thesis is the analysis of safety of program transformations in the JMM: we show that several important optimisations, including the common subexpression elimination, are not safe in Java.

1.3 Aims

Our main goal is to analyse safety of important program transformations in the DRF guarantee. A transformation is *safe* in the DRF guarantee if it cannot introduce any new behaviours. More precisely, a transformation from a data race free program P to a program P' is safe if any sequentially consistent execution of P' has the same behaviour as some execution of P . Note that safe transformations can remove behaviours. Moreover, we do not consider termination

behaviours to be observable. Therefore, our transformations can change termination behaviours. We leave the study of termination behaviours for future work. Our second goal is to show that the transformations provide some basic guarantees even for programs with data races. These guarantees should prevent at least the behaviours described in the previous section, i.e., values should not appear out-of-thin-air, if we execute the transformed program sequentially consistently. Finally, we will analyse safety of the transformations in the Java Memory Model.

1.4 Transformations by Example

In this section we describe the transformations that we have considered in our analysis. Our transformations are local, i.e., they should be valid in any context. We use variables `x` and `y` for shared memory locations and `r1`, `r2` for thread-local registers. We believe that the transformations shown here cover most cases of optimisations encountered in real compilers.

1.4.1 Trace-preserving Transformations

It is useful to view programs as sets of possible sequences, or traces, of memory-related operations, such as memory accesses, locks and unlocks. Any transformation that does not change the set of traces should be valid. E.g., if both branches of a conditional—whose guard does not examine memory—contain the same code, it is valid to eliminate the conditional, as illustrated by Figure 1.2.

$$\begin{array}{lcl}
 \text{if } (r1==1) & & \\
 \quad \{x:=1;y:=1\} & \Leftrightarrow & x:=1 \\
 \text{else } \{x:=1;y:=1\} & & y:=1
 \end{array}$$

Figure 1.2: Trace preserving transformation.

1.4.2 Reordering

Reordering of independent statements is an important transformation that swaps two consecutive non-synchronisation memory accesses. It is often performed in hardware (Intel, 2007, 2002; Sparc International, 2000), or in a compiler’s loop optimiser (Kennedy and Allen, 2002; Click, 1995). Although Manson et al. claim

this transformation to be valid in the JMM (Manson et al., 2005, Theorem 1), Cenciarelli et al. (2007) found a counterexample to this. Figure 1.3 illustrates reordering of independent statements in one branch of an `if`-statement.

<pre>r1:=z; if (r1==1) {x:=1;y:=1} else {y:=1;x:=1}</pre>	→	<pre>r1:=z; if (r1==1) {x:=1;y:=1} else {x:=1;y:=1}</pre>
---	---	---

Figure 1.3: Reordering.

1.4.3 Redundant Read Elimination

Elimination of a redundant read is a transformation that replaces a read immediately preceded by a read or a write to the same variable by the value of that read/write. This transformation is often performed as a part of common subexpression elimination or constant propagation optimisations in compilers. For example, the two examples of transformations in Figure 1.4 reuse the value of `x` stored in register `r1` instead of re-reading `x`.

<pre>r1 := x r2 := x if (r1==r2) y := 1</pre>	→	<pre>r1 := x r2 := r1 if (r1==r2) y := 1</pre> <p style="text-align: center;">(read after read)</p>	<pre>x := r1 r2 := x if (r1==r2) y := 1</pre>	→	<pre>x := r1 r2 := r1 if (r1==r2) y := 1</pre> <p style="text-align: center;">(read after write)</p>
---	---	---	---	---	--

Figure 1.4: Reusing a value for a read.

1.4.4 Irrelevant Read Elimination

A read statement can also be removed if the value of the read is not used. For example, `r1:=x;r1:=1` can be replaced by `r1:=1`, because the register `r1` is overwritten by the value 1 immediately after reading shared variable `x`, and thus the value read is irrelevant for the continuation for the program. An example of this transformation is dead code elimination because of dead variables.

1.4.5 Irrelevant Read Introduction

Irrelevant read introduction is the inverse transformation to the irrelevant read elimination. It might seem that this transformation is not an optimisation, because it introduces new computations on some program paths, but processor hardware or an aggressive compiler might introduce irrelevant reads speculatively (Murphy et al., 2008). For example, the first transformation in Figure 1.5 introduces irrelevant read of `x` in the `else` branch of the conditional (assuming that `r2` is not used in the rest of the program). In terms of traces, this is equivalent to reading `x` speculatively, as demonstrated by the program on the right.

<pre> if (r1==1) {r2:=x; y:=r2} r2:=...</pre>	→	<pre> if (r1==1) {r2:=x; y:=r2} else r2:=x r2:=...</pre>	⇔	<pre> r2:=x; if (r1==1) y:=r2 r2:=...</pre>
--	---	---	---	---

Figure 1.5: Introducing an irrelevant read.

1.4.6 Redundant Write Elimination

This transformation eliminates a write in two cases: (i) if it follows a read of the same value, or (ii) if it precedes another write to the same variable. For example, in the first transformation in Figure 1.6 the write `x:=1` can be eliminated, because in all traces where the write occurs, it always follows a read of `x` with value 1. The other transformation in Figure 1.6 shows the elimination of a previous overwritten write. This transformation is often included in peephole optimisations (Aho et al., 1986).

<pre> r := x if (r == 1) x := 1</pre>	→	<pre> r := x</pre>	→	<pre> x := 1 x := 3</pre>
(write after read)				(write before write)

Figure 1.6: Redundant write elimination.

1.4.7 Roach-motel Reordering

Intuitively, increasing synchronisation should limit a program’s behaviours. In the limit, if a program is fully synchronised, i.e., data race free, the DRF guarantee promises only sequentially consistent behaviours. One way of increasing synchronisation is moving normal memory accesses into synchronised blocks, see Figure 1.7. Although compilers do not perform this transformation explicitly, it may be performed by underlying hardware if a synchronisation action uses only one memory fence to prevent the code inside a synchronised section from being reordered to outside of the section.

<code>x:=1</code>		<code>lock(m)</code>		<code>lock(m)</code>
<code>lock(m)</code>			→	<code>x:=1</code>
<code>y:=1</code>				<code>y:=1</code>
<code>unlock(m)</code>				<code>unlock(m)</code>

Figure 1.7: Roach motel—memory accesses can be moved in, but not out.

1.4.8 Reordering with External Actions

As well as reordering memory operations with one another, one may consider richer reorderings, for example, reordering memory operations with external operations. This seems more likely to alter the behaviour of a program, but it is valid for data race free programs under sequential consistency. For example, the exchange of printing a constant with a memory write: `x:=1;print 1` → `print 1;x:=1`.

1.5 Summary of Results and Outline of Contents

In this thesis, we study how common compiler transformations comply with the DRF guarantee and the Java Memory Model. Table 1.1 contains a summary of the transformations that we have analysed. For comparison, the first column of the table shows which transformations are valid under sequential consistency. The second column gives an overview of our results for safety of the transformations with respect to the DRF guarantee. We prove these results, except for the read

Table 1.1: Validity of transformations in the DRF guarantee and in the JMM.

Transformation	SC	DRF guarantee	JMM
Trace-preserving transformations	✓	✓	✓
Reordering normal memory accesses	×	✓	×
Redundant read after read elimination	✓	✓	×
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	?	×
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	✓	×
Roach-motel reordering	×(✓ for locks)	✓	×
External action reordering	×	✓	×

introduction, in Chapters 2, 3 and 4. The third column summarises our results⁴ for the Java Memory Model. We give a counterexample for each negative case and we prove safety of all the valid transformations in Chapter 5.

1.5.1 Overview

We give a brief synopsis of each chapter here:

In **Chapter 2**, we introduce the trace semantics of concurrent shared-memory programs. We view multi-threaded programs as sets of traces, which are sequences of memory operations, i.e., reads and writes, synchronisation operations, such as locks, unlocks, and observable operations, such as output. We define the executions of such programs as interleavings of these traces such that each read has the same value as the most recent write to the same variable. The observable behaviour of an execution is the sequence of all observable operations in the execution. The set of observable behaviours of a program is the set of observable behaviours of its executions. We say that a program is data race free if it cannot access one shared variable from two different threads concurrently, where at least one of the threads performs a write.

In **Chapter 3**, we prove safety of semantic elimination and reordering transformations for data race free programs. We define the semantic transformations to be relations on semantic programs, i.e., on sets of traces. A transformation from

⁴We have reported on this work in Ševčík and Aspinall (2008).

P to P' is safe if data race freedom of P implies that any observable behaviour of P' is an observable behaviour of P . In other words, safe transformations cannot introduce new behaviours for data race free programs. We define two classes of transformations: elimination and reordering. The former includes all the cases of eliminations from Section 1.4, the latter covers all the reordering examples. In addition to the proof of safety of any composition of the elimination and reordering transformations, we prove that the transformations cannot introduce values out-of-thin-air.

Chapter 4 instantiates the semantic framework for a set of syntactic transformations in a simple imperative language. After specifying the syntax, we define the trace semantics of our language and give a formal definition of several syntactic elimination and reordering transformations. We prove that the syntactic transformations are indeed instances of the semantic transformations. We use our results from Chapter 3 to conclude that the syntactic transformations are safe.

In **Chapter 5** we analyse safety of the transformations in the Java Memory Model. We show that several important classes of transformations, such as redundant read elimination or roach motel reordering, are illegal. Since Sun's Java Hotspot virtual machine performs some of these transformations, it does not conform with the Java Language Specification.

Chapter 6 concludes with a discussion of related work and with an overview of open questions.

There is a linear dependency among Chapters 2, 3 and 4—Chapter 2 defines the trace semantics used in Chapters 3 and 4, and Chapter 4 builds on the semantic transformations defined in Chapter 3. Chapter 5 is independent of the other chapters since we redefine the basic concepts from the ground up due to subtle differences between our notation and the notation of the Java Memory Model.

Chapter 2

Preliminaries

This chapter introduces the concepts that we use to reason about safety of transformations in multi-threaded programs formally. To describe the meaning of programs, we employ a trace semantics, where the traces are sequences of atomic memory operations. In addition to the standard read, write and synchronisation operations, the traces also include external operations, such as printing, because ultimately we wish to utilise the semantics to prove that our transformations do not introduce new external behaviours. Our framework is not bound to any specific language—we view programs as sets of traces. Later, in Chapter 4, we provide an example of a connection between a simple imperative language and our trace semantics.

We will start the chapter with an introduction to memory operations and traces. After the introduction to traces we will make precise the intuitive concept of interleavings, sequential consistency and data race freedom. The interleavings determine some important partial orders on actions, such as program order, synchronisation order and happens-before order. Similarly to the Java Memory Model, we use the happens-before order to define data race freedom and we will prove that it is equivalent to a more intuitive definition.

2.1 Actions, Traces and Interleavings

Before we define traces, we make precise our assumptions about the primitive entities that describe the shared memory.

Definition 2.1. A *memory* is a tuple $\langle L, M, T_\theta, L_v, V, \tau, D \rangle$, where

- L is a set of locations,

- M is a set of synchronisation monitors,
- T_θ is a set of thread identifiers,
- $L_v \subseteq L$ is the set of volatile locations,
- V is a set of values,
- τ is a function that assigns a type to each location, i.e., it is a function from L to the power set of V ,
- D is a function mapping a location to the default value for that location. We require that $D(l) \in \tau(l)$ for all $l \in L$. Intuitively, the default value of a location is the value, which we obtain if we read the location without any previous write.

With the exception of concrete examples and concrete language instantiation (Chapter 4), we assume an arbitrary fixed memory $\langle L, M, T_\theta, L_v, V, \tau, D \rangle$. We will say that location $l \in L$ is *volatile* if $l \in L_v$. Similarly to the Java Memory Model (see Chapter 5), the volatile locations provide stronger guarantees than the normal locations.

In examples, we will use letters \mathbf{x} , \mathbf{y} , and \mathbf{z} for distinct normal locations, and \mathbf{u} , \mathbf{v} for distinct volatile locations. The type of locations in examples will be natural numbers with default value 0. For the synchronisation monitors, we will use identifiers \mathbf{m}_1 , \mathbf{m}_2 , etc. Thread identifiers are natural numbers in examples.

The traces are finite sequences of operations called *actions*.

Definition 2.2. An *action* is one of the following:

1. Read from location l with value v , denoted by $\text{Rd}(l, v)$,
2. Write to location l with value v , denoted by $\text{Wr}(l, v)$,
3. Lock of monitor m , $\text{L}(m)$,
4. Unlock of monitor m , $\text{U}(m)$,
5. External action with value v , $\text{Ext}(v)$,
6. Thread start with entry point e , $\text{S}(e)$, where the entry point is a thread identifier.

The thread start action is always the first action of a thread. Its goal is to provide a connection between the identity of a thread and its entry point. To simplify the framework, we create threads statically and we use thread identifiers as entry points. We discuss a possible extension to dynamically created threads in Section 6.3.2.

We will use the following terminology to refer to classes of actions:

- a *memory access* to location l is a read or a write to l ,
- a *volatile memory access*, *volatile read* and *volatile write* are a memory access, a read and a write to a volatile location respectively,
- a *normal memory access*, *normal read* and a *normal write* are an access, a read and a write to a non-volatile location respectively,
- an *acquire* action is either a lock or a volatile read,
- a *release* action is an unlock or a volatile write,
- a *synchronisation* action is an acquire or release action.

Programs are represented as sets of memory traces of its threads. A trace does not have to be executed completely, its execution can finish at any point. In other words, the set of traces of a program is prefix-closed. Each trace contains only actions of a single thread. We will show how to interleave such traces in Section 2.1.1.

Example 2.1. The set of traces of the program (r is a thread-local register)

Thread 0: `r := x; print r`

Thread 1: `x := 1`

is the prefix closure of the set

$$\{[S(0), \text{Rd}(x, v), \text{Ext}(v)] \mid v \in \tau(x)\} \cup \{[S(1), \text{Wr}(x, 1)]\}.$$

For example, the sequences $[S(0), \text{Rd}(x, 2)]$ and $[S(0), \text{Rd}(x, 1), \text{Ext}(1)]$ are traces of the program above, but the sequence $[S(0), \text{Rd}(x, 1), \text{Ext}(2)]$ is not. The traces do not contain any actions on registers, because registers are not part of the shared memory—registers reside in thread-local memory, usually on the thread's stack, and they cannot be used to interact with other threads.

To work with sequences of actions we use the following notation:

- $t \# t'$ is a *concatenation* of lists t and t' .
- t is a prefix of t' , written $t \leq t'$, if there is s such that $t \# s = t'$. Trace t is a *strict prefix* of t' ($t < t'$), if $t \leq t'$ and $t \neq t'$.
- $|t|$ denotes the length of the sequence t .
- t_i denotes i -th element of the list t . Indices are 0-based.
- $[a \leftarrow t. P(a)]$ stands for the list of all actions in t that satisfy condition P . In functional languages, this is often expressed by `filter P t`.
- we generalise the filter notation to a *map-filter* notation. The list $[a \leftarrow t. P(a)]$ with each element transformed by function f will be denoted by the expression $[f(a) \mid a \leftarrow t. P(a)]$. In functional languages, this is written `map f (filter P t)`.
- $\text{dom}(t)$ is the set of all indices to t :

$$\text{dom}(t) = \{0, \dots, |t| - 1\}.$$

- $\text{ldom}(t)$ is the list of all indices to t in the increasing order:

$$\text{ldom}(t) = [0, \dots, |t| - 1].$$

- $t|_S$ is a *sublist* of t that contains all elements with indices from S . For example, $[a, b, c, d]|_{\{1,3\}}$ is $[b, d]$. Formally,

$$t|_S = [t_i \mid i \leftarrow \text{dom}(t). i \in S]$$

Definition 2.3. We say that a set of traces T is a *traceset* if it satisfies these properties:

1. Prefix closedness, i.e. $t \leq t'$ and $t' \in T$ implies $t \in T$.
2. Proper locking, i.e. for each $t \in T$ and monitor m , the number of unlocks of m in t is at most the same as the number of locks of m in t . The monitors in our semantics are Java-style re-entrant.
3. Traces are properly started, meaning that if a trace is not empty, then $t_0 = S(e)$ for some e . Moreover, if $t_i = S(e)$ then $i = 0$.

4. Proper typing. If $t_i \in \{\text{Rd}(l, v), \text{Wr}(l, v)\}$, then $v \in \tau(l)$.

We should note that this notion of traceset is quite weak and there are tracesets that cannot be generated by programs in reasonable programming languages. Even if we ignore the ability of tracesets to represent obviously non-computable programs, such as a program that computes halting problem, and consider only tracesets with traces of bounded length, there are still two cases that might seem counterintuitive:

- *Partial reads.* Consider the traceset

$$\{\ [], [S(0)], [S(0), \text{Rd}(x, 1)] \},$$

where the type of x contains (at least) values 0 and 1. Although this is a valid traceset, no program can possibly read value 1 but avoid reading 0. Even the simplest program with a read, such as

```
Thread 0: r := x
```

has a traceset

$$\{\ [], [S(0)] \} \cup \{ [S(0), \text{Rd}(x, v)] \mid v \in \tau(x) \}.$$

In fact, our proof of legality of transformations in the Java Memory Model excludes such tracesets (see the second condition in Definition 5.4); however, it seems that such a condition is not necessary for our framework for transformation safety and we can leave it out.

- *Nondeterminism.* The traceset definition does not entail that the next memory operation must be uniquely determined. For example, there is a traceset that allows a thread to write to location x or to y , or even write different values to the same location:

$$\{\ [], [S(0)], [S(0), \text{Wr}(x, 1)], [S(0), \text{Wr}(y, 1)], [S(0), \text{Wr}(y, 2)] \}$$

In fact, such loose definition of tracesets might be useful to represent an underspecified evaluation order. Although Java or C# programs cannot have such tracesets, C++ does not specify the order of evaluation of function arguments. For instance, the following C++ program generates a superset of the traceset above.

```
void f(int a, int b, int c) {}
f(x = 1, y = 1, y = 2)
```

We believe that it is useful to define tracesets as loosely as possible so that we can represent a large class of languages. Of course, the definition cannot be too broad, otherwise we would not be able to prove any useful guarantees.

2.1.1 Interleavings and Executions

Interleavings are sequences of thread-identifier–action pairs. For such pair $p = \langle \theta, a \rangle$, we use the notation $\mathcal{A}(p)$ to refer to the action a , and $\mathcal{T}(p)$ to refer to the thread θ . Any sequence of these pairs is an *interleaving*. Given an interleaving I , the *trace of θ in I* is the sequence of actions of thread θ in I , i.e. $[\mathcal{A}(p) \mid p \leftarrow I. \mathcal{T}(p) = \theta]$. In the text, we often omit the projection $\mathcal{A}(-)$ when referring to the action part of a thread-action pair. For example, we might abbreviate “ $\mathcal{A}(I_i)$ is a read” to “ I_i is a read”. Sometimes we use the indices as action identifiers and say that i is a read (resp. write) of location l with value v in I if $\mathcal{A}(I_i) = \text{Rd}(l, v)$ (resp. $\mathcal{A}(I_i) = \text{Wr}(l, v)$).

A reasonable interleaving should respect mutual exclusivity of locks:

Definition 2.4. An interleaving I is *properly locked* if $\mathcal{A}(I_i) = \text{L}(m)$ implies that for each thread $\theta \neq \mathcal{T}(I_i)$ we have an equal number of locks of m and unlocks of m in θ before i , i.e.,

$$|\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = \text{L}(m)\}| = |\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = \text{U}(m)\}|$$

Definition 2.5. An interleaving I is an *interleaving of traceset T* if:

1. for all thread identifiers θ , the trace of θ in I is in T ,
2. thread identifiers correspond to entry-points, i.e., $\mathcal{A}(I_i) = \text{S}(\theta)$ implies $\mathcal{T}(I_i) = \theta$ for all i and θ .

An interleaving is sequentially consistent if all reads have the same value as the most recent write to the same variable, or the default value, if there is no earlier write to the same location. Formally:

Definition 2.6. Given an interleaving I , we say that $r \in \text{dom}(I)$

1. *sees write w* if $\mathcal{A}(I_r) = \text{Rd}(l, v)$, $\mathcal{A}(I_w) = \text{Wr}(l, v)$, $w < r$, and for all i such that $w < i < r$ the action $\mathcal{A}(I_i)$ is not a write to l .
2. *sees default value* if I_r is a read of the default value $D(l)$ from some location l and there is no $i < r$ such that $\mathcal{A}(I_i)$ is a write to l .
3. *sees the most recent write* if r sees the default value or r sees write w for some w or I_r is not a read.

An interleaving I is *sequentially consistent* (SC) if all $j \in \text{dom}(I)$ see the most recent write in I .

We often refer to properly locked and sequentially consistent interleavings of T as *executions* of T .

Example 2.2. The program (already used in Example 2.1)

Thread 0: `r := x; print r`

Thread 1: `x := 1`

has an interleaving

$$[\langle 0, \text{S}(0) \rangle, \langle 0, \text{Rd}(x, 0) \rangle, \langle 1, \text{S}(1) \rangle, \langle 1, \text{Wr}(x, 1) \rangle, \langle 0, \text{Ext}(0) \rangle].$$

This interleaving is in fact an execution of the program, because all its reads see the most recent writes.

Similarly, the sequence

$$[\langle 0, \text{S}(0) \rangle, \langle 1, \text{S}(1) \rangle, \langle 1, \text{Wr}(x, 1) \rangle, \langle 0, \text{Rd}(x, 0) \rangle]$$

is an interleaving of the program above, but it is not sequentially consistent, because the read of x has a different value from the value of the most recent write to location x .

On the other hand, the interleaving

$$[\langle 0, \text{S}(0) \rangle, \langle 1, \text{S}(1) \rangle, \langle 0, \text{Rd}(x, 0) \rangle, \langle 0, \text{Ext}(1) \rangle]$$

is not even an interleaving of the program above, because the trace of the thread 0 is not a trace of the program.

Definition 2.7. The *external behaviour* of execution I is the subsequence of all its external actions, i.e.,

$$[\mathcal{A}(p) \mid p \leftarrow I. \exists v. \mathcal{A}(p) = \text{Ext}(v)].$$

Traceset T has an external behaviour B if there is an execution of T that has the external behaviour B .

2.1.2 Orders on Actions

In this work, we will use the standard terminology for orders: A relation $R \subseteq S \times S$ on set S is *reflexive*, *antisymmetric*, and *transitive* if $x \in S$ implies xRx , xRy and yRx implies $x = y$, and xRy and yRz implies xRz , respectively. A relation R on S is a *partial order* if it is reflexive, antisymmetric and transitive. A partial order R on S is a *total order* if for all x and y in S we have xRy or yRx . The relations R and Q are *consistent* if xRy and yQx implies $x = y$. The *transitive closure* of a relation R , denoted by R^+ , is the smallest transitive relation that contains R .

The sequencing of actions in interleavings imposes a total order on the execution of actions. In reality, actions are often performed concurrently, and we will model this by constructing a partial ‘happens-before’ order (Lamport, 1978), which relates actions only if the ordering is enforced by the program or by synchronisation. For a given interleaving, we will use the indices to actions in the interleaving, i.e., $\text{dom}(I)$, as the carrier of the order.

First, we define an order that relates actions of the same thread. This order is determined by the sequence of actions in the trace of each thread.

Definition 2.8. Given an interleaving I , we define the *program order* of I as follows:

$$\leq_{po}^I = \{(i, j) \mid 0 \leq i \leq j < |I| \wedge \mathcal{T}(I_i) = \mathcal{T}(I_j)\}.$$

Fact 2.1. The relation \leq_{po}^I is a partial order on $\text{dom}(I)$.

Next, we define a total order on all synchronisation actions in I .

Definition 2.9. Given an interleaving I , the *synchronisation order* of I is the relation

$$\leq_{so}^I = \{(i, j) \mid 0 \leq i \leq j < |I| \text{ and } \mathcal{A}(I_i), \mathcal{A}(I_j) \text{ are synchronisation actions}\}.$$

Fact 2.2. The relation \leq_{so}^I is a total order on the set of synchronisation actions, i.e., on the set

$$\{i \mid 0 \leq i < |I| \text{ and } \mathcal{A}(I_i) \text{ is a synchronisation action}\}.$$

However, not all synchronisation enforces ordering, e.g., synchronisation on different monitors should not enforce any ordering. This is captured by the following definitions.

Definition 2.10. Actions a and b are a *release-acquire* pair if

- a is an unlock of monitor m and b is a lock of m , or
- a is a write to a volatile location l and b is a read of l .

Definition 2.11. We say that index i *synchronises-with* j in an interleaving I , written $i <_{sw}^I j$, if $i \leq_{so}^I j$ and I_i, I_j are a release-acquire pair.

Equivalently, i synchronises-with j if $i < j$ and I_i, I_j are a release-acquire pair.

Definition 2.12. For an interleaving I , we define the *happens-before order* as the relation

$$\leq_{hb}^I = (\leq_{po}^I \cup <_{sw}^I)^+.$$

Note that $i \leq_{po}^I j$ implies $i \leq j$, and $i <_{sw}^I j$ implies $i \leq j$. Hence, $i \leq_{hb}^I j$ implies $i \leq j$. As any subset of a total order is antisymmetric and \leq_{hb}^I is transitive and reflexive by construction, the happens-before order is a partial order.

Example 2.3. Figure 2.1 shows an interleaving of the program

```
Thread 1: lock m; x := 1; unlock m;
Thread 2: lock m; r := x; unlock m;
```

The solid edges in Figure 2.1 represent the program order of the interleaving, and the dashed edge represents synchronises-with relation. Action a happens-before b if there is a path from a to b using program order and synchronises-with edges. In our example, the action in position 3, i.e., the write to x , happens-before the action in position 6, i.e., the read of x . Formally, we will write $3 \leq_{hb} 6$.

On the other hand, if we replace the monitor m in thread 1 by monitor $m1$ and the monitor m in thread 2 by $m2$, then the unlock of $m1$ in thread 1 and the lock of $m2$ in thread 2 are not a release-acquire pair, see Figure 2.2. Therefore, there is no path between the write to x and the read of x , i.e., $3 \not\leq_{hb} 6$. Such unordered accesses will be called data races (Definition 2.16).

Example 2.4. Another way to induce happens-before ordering on actions is using volatile locations. For example, Figure 2.3 shows an interleaving of the program

```
Thread 1: x:=3; v:=1;
Thread 2: y:=2; r1:=v; if (r1==1) {r2:=x; print r2;}
```

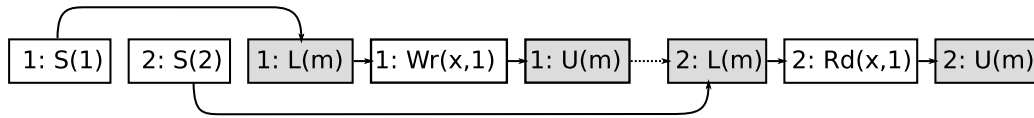


Figure 2.1: Example of ordering through locks.

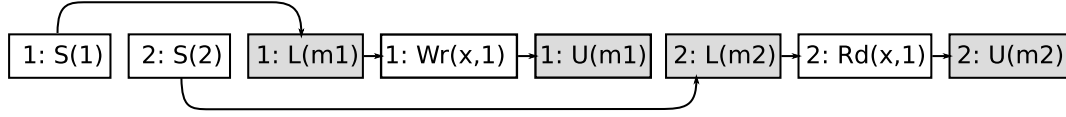


Figure 2.2: Different monitors do not induce ordering.

In this interleaving, the write to x does not happen-before the write to y ($2 \not\leq_{hb} 3$). This will not constitute a data race, because data races can only occur on accesses of the same variable. Also observe that the write to x happens-before the read of x , i.e., $2 \leq_{hb} 6$, because there is an ordering path through the release-acquire pair on the volatile location.

2.1.3 Matchings

In this subsection, we define a special function that will let us relate actions in different interleavings. Given two sequences, a *matching between* them is any injective function that relates the same elements from these sequences. More precisely:

Definition 2.13. We say that function f is a *matching* between I and I' if

- the domain of f , denoted by $\text{dom}(f)$, is a subset of $\text{dom}(I)$,
- the range of f , $\text{rng}(f)$, is a subset of $\text{dom}(I')$,
- f is injective,
- for all $i \in \text{dom}(f)$ we have $I_i = I'_{f(i)}$.

The matching f is *complete* if $\text{dom}(f) = \text{dom}(I)$.

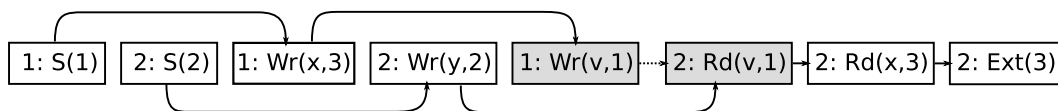


Figure 2.3: Ordering through volatile location accesses.

Observe that if f is a matching between I and I' , then f^{-1} is a matching between I' and I . However, the inverse does not preserve completeness. For example, the function $f = \{(0, 1)\}$ is a complete matching between $[a]$ and $[b, a]$. On the other hand, the matching $f^{-1} = \{(1, 0)\}$ is not a complete matching between $[b, a]$ and $[a]$, because the element b is not matched. Complete matchings are interesting because they determine the source list in the following sense:

Definition 2.14. Suppose we have a list I . Any list l of distinct indices to I is called an *index list* to I . We define the *index function* χ_l as $\chi_l(i) = l_i$ for $i \in \text{dom}(l)$.

Lemma 2.3. Let I be a list and l an index list to I . Then χ_l is a complete matching function between $[I_i \mid i \leftarrow l]$ and I .

Similarly, there is a correspondence between matchings and sublists.

Lemma 2.4. Assume that I and J are lists and $S \subset \text{dom}(I)$. Then $J = I|_S$ if and only if there is an increasing complete matching $f : \text{dom}(J) \rightarrow \text{dom}(I)$ such that

$$\text{rng}(f) = \{i \mid i \in \text{dom}(I) \wedge i \in S\}.$$

Proof. By induction on the size of I . □

In the following, we will prove that a certain class of matchings reflects proper locking of interleavings. First, we observe that locks on the same monitor are always totally ordered by the happens-before order:

Lemma 2.5. Let m be a monitor and t be a trace from a traceset T with the same number of locks and unlocks on a monitor m . Then $t_i = L(m)$ implies that there is $j > i$ such that $t_j = U(m)$.

Proof. This is a direct consequence of the proper locking of T (Definition 2.3). □

Lemma 2.6. Let I be a properly locked interleaving of a traceset T , and let $\mathcal{A}(I_i) = \mathcal{A}(I_j) = L(m)$ for some monitor m and indices $i < j \in \text{dom}(I)$. Then $i \leq_{hb}^I j$.

Proof. If I_i and I_j belong to the same thread, then $i \leq_{po}^I j$ and we are done. On the other hand, if $\mathcal{T}(I_i) \neq \mathcal{T}(I_j)$, let us take the sequence t all the actions $\mathcal{A}(I_k)$ such that $\mathcal{T}(I_i) = \mathcal{T}(I_k)$ and $k < j$. By proper locking of I , there must be the

same number of locks and unlocks of m in t . Moreover, t is a prefix of the trace of $\mathcal{T}(I_i)$ in I , so t is in T . By Lemma 2.5, for each lock of m in t there must be a later unlock of m in t . In particular, there must be k such that $i < k < j$ such that $\mathcal{A}(I_k) = U(m)$ and $\mathcal{T}(I_k) = \mathcal{T}(I_i)$. Thus, $i <_{po}^I k <_{sw}^I j$. \square

Corollary 2.7. *Let I be a properly locked interleaving of some traceset T , $i < j \in \text{dom}(I)$, $\mathcal{A}(I_i) \in \{L(m), U(m)\}$ and $\mathcal{A}(I_j) = L(m)$ for some monitor m . Then $i \leq_{hb}^I j$.*

With this observation in hand, we can prove that some restricted class of matchings preserves proper locking.

Lemma 2.8. *Let I be a properly locked interleaving of a traceset T and I' an interleaving such that there is a complete matching f between I' and I such that*

1. $\text{rng}(f)$ is downward closed under the happens-before order \leq_{hb}^I ,
2. f preserves ordering of actions on each monitor, i.e., if $i < j \in \text{dom}(I')$ and $\mathcal{A}(I'_i), \mathcal{A}(I'_j) \in \{L(m), U(m)\}$ for some monitor m , then $f(i) < f(j)$.

Then I' is properly locked.

Proof. Let $\mathcal{A}(I'_i) = L(m)$ and $\theta \neq \mathcal{T}(I'_i)$. Let

$$\begin{aligned} L_{I'} &= \{j \mid j < i \wedge \mathcal{T}(I'_j) = \theta \wedge \mathcal{A}(I'_j) = L(m)\} && \text{and} \\ L_I &= \{j \mid j < f(i) \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = L(m)\} && \text{and} \\ U_{I'} &= \{j \mid j < i \wedge \mathcal{T}(I'_j) = \theta \wedge \mathcal{A}(I'_j) = U(m)\} && \text{and} \\ U_I &= \{j \mid j < f(i) \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = U(m)\} \end{aligned}$$

We want to show that

$$|L_{I'}| = |L_I| = |U_I| = |U_{I'}|. \quad (2.1)$$

The second equality follows directly from proper locking of I and the fact that $I_{f(i)}$ is a lock of m .

Let us prove the first equality of 2.1. First note that $|L_{I'}| = |f(L_{I'})|$, because f is an injective function. Moreover, we have $f(L_{I'}) = L_I$:

- Let $k \in L_{I'}$. By the assumption (2), we have $f(k) < f(i)$. As a result, $f(k) \in L_I$.
- If $k \in L_I$ then $k \leq_{hb}^I f(i)$ by Corollary 2.7. Since $\text{rng}(f)$ is downward closed on happens-before, there must be j such that $f(j) = k$. As f preserves the order of locks and unlocks on the same monitor, we have $j \in L_{I'}$, thus $k = f(j) \in f(L_{I'})$.

Using the same reasoning, we prove that $|U_I| = |U_{I'}|$. Therefore, I' is properly locked. \square

Lemma 2.9. *Let I be an interleaving of a traceset T and I' an interleaving such that there is a complete, program order preserving matching f between I' and I such that $\text{rng}(f)$ is downward closed under the program order \leq_{po}^I . Then I' is an interleaving of T .*

Proof. For any thread θ , the trace of θ in I' is a prefix of the trace of θ in I ; therefore, it belongs to T since T is prefix closed. \square

2.2 Data Race Freedom

A program is data race free, if in all its executions, all conflicting actions are ordered by happens-before relation. The formal definition follows.

Definition 2.15. Actions a and b are *conflicting* if they are memory accesses to the same non-volatile location and at least one of them is a write.

Definition 2.16. Given an interleaving I , indices $i < j \in \text{dom}(I)$ are a *data race* if $\mathcal{A}(I_i)$ and $\mathcal{A}(I_j)$ are conflicting actions and $i \not\leq_{hb}^I j$. We say that I has a *data race* if there are such i and j .

Definition 2.17. Traceset T is *data race free* if there is no execution of T that has a data race.

Example 2.5. A classical example of a data race free program is a program that protects each shared location with a designated monitor. For example, the program

```

Thread 0: lock m1; x := 1; unlock m1;
          lock m2; r1 := y; unlock m2;
          print r1;
Thread 1: lock m2; y := 1; unlock m2;
          lock m1; r2 := x; unlock m1
          print r2;

```

protects location x with monitor $m1$ and location y with monitor $m2$. Therefore, in any interleaving of the program there must be a unlock-lock pair of actions on the same monitor between two accesses of the same location.

In addition to synchronisation with monitors, we can use volatile locations to get data race freedom. Example 2.4 shows a simple data race free program, where volatile location enforces ordering. The following example shows a slightly more complex usage of volatile variables.

Example 2.6. For example, let u, v be volatile locations and x a normal location with default values 0. Observe that in any interleaving of the following program each pair of memory accesses to location x must be separated either by a release-acquire pair on volatile location u or v , or the memory accesses occur in the same thread.

```

Thread 1: x := 1;
          u := 1;
          r1 := v;
          if (r1 == 1) {
            r2 := x;
            print r2;
          }
Thread 2: r3 := u;
          if (r3 == 1) {
            x := 2;
            v := 1;
          }

```

Note this program is essentially the same as our first motivating example in Chapter 1. Using the analogy to the program in the figure, it is interesting to

note that it suffices to mark the synchronisation locations u and v volatile to get data race freedom. The location x with the data can remain non-volatile.

This example also shows that the `volatile` modifier from C/C++ is not strong enough to ensure sequential consistency for this program. In C/C++, the `volatile` modifier prevents any optimisations on the modified variable. As noted in the introduction, many optimising C/C++ compilers will replace the statement `print r2` with `print 1` and remove `r2:=x`, because x is not volatile. After this optimisation, the transformed program can output 1. This was not possible in any interleaving of the original program. As we will see later in Chapter 3, compilers should be more careful about the volatile locations and prevent optimisations across the volatile locations in addition to the optimisations on the volatile location itself.

Finally, let us note that programs may be data race free even if they do not contain any synchronisation (Manson et al., 2005):

Example 2.7. Consider the program

```
Thread 0: r1 := x;
          if (r1 == 1) y := 1;
          print r1;
Thread 1: r2 := y;
          if (r2 == 1) x := 1;
```

The program is data race free, because there is no interleaving that contains a write to x or y , thus there cannot be any pair of conflicting actions.

2.2.1 Alternative DRF

The Definition 2.17 says that all conflicting accesses to the same variable must be ordered by synchronisation. Perhaps a more intuitive definition should say that a program is data race free if there is no execution such that some two conflicting actions are executed by different threads ‘at the same time’. More precisely:

Definition 2.18. An interleaving I has a *strong data race* if I can be decomposed into

$$I = \alpha \# [a, b] \# \beta$$

so that $\mathcal{A}(a)$ and $\mathcal{A}(b)$ conflict and $\mathcal{T}(a) \neq \mathcal{T}(b)$.

Definition 2.19. A traceset T is *weakly data race free*, if no execution of T has a strong data race.

As our intuition suggests, the standard data race freedom and the weak data race freedom are indeed equivalent, because we can always transform an interleaving with a data race to an interleaving with a strong data race. This result is not completely new—Boehm and Adve (2008) and Boyland (2008) proved essentially the same result independently using a similar observation. However, it nicely demonstrates the techniques that we will use to prove the safety of transformations in Chapter 3.

Lemma 2.10. *Let I be an execution of T with a data race. Then there is an execution $I' = \alpha \# [a, b]$ of T such that $\mathcal{A}(a)$ and $\mathcal{A}(b)$ conflict, $\mathcal{T}(a) \neq \mathcal{T}(b)$, and $\alpha \# [a]$ does not have a data race.*

The main idea of the proof is straightforward. We have $I = \alpha \# [a] \# \beta \# [b] \# \gamma$ such that $\mathcal{A}(a)$ and $\mathcal{A}(b)$ conflict, their indices in I are not ordered by happen-before, and a and b are the earliest race in I in the sense that $\alpha \# [a] \# \beta$ is data race free. If there is more than one data race in $\alpha \# [a] \# \beta \# [b]$, we will choose the decomposition with minimum length of β . In other words, we choose the latest a such that a is in a data race with b .

We construct the interleaving I' in the following way: we take the prefix α , then we append to it all actions from β that strictly happen-before b and finally we append a and b . More precisely, let i and j be the positions of a and b in I , i.e., $i = |\alpha|$ and $j = |\alpha| + |\beta| + 1$. Then

$$I' = I|_{\{0, \dots, i-1\}} \# I|_{\{k \mid i < k < I_{hb}j\}} \# [a, b].$$

The more difficult part of the proof is showing that I' is indeed an execution of T . We will prove this claim in the rest of this section.

We will say that an interleaving I is *sequentially consistent on* a set of indices S , if each $i \in S$ sees the most recent write in I . Our next lemma will establish sequential consistency for all elements in our newly constructed interleaving I' , except the last two elements, which will be treated later.

Lemma 2.11. *Let us assume that we have interleavings I , I' , and a matching f between I' and I such that*

1. I is sequentially consistent on $\text{rng}(f)$,

2. $\text{dom}(f)$ is an initial segment of natural numbers; in other words, $\text{dom}(f) = \{0, \dots, n\}$ for some $n < |I'|$.
3. f is an increasing function,
4. $\text{rng}(f)$ is downward closed on happens-before of I ,
5. if for $i \in \text{rng}(f)$, the action I_i is a read, then either the most recent write to the same location in I happens-before i , or there is no earlier write to the same location and I_i has the default value.

Then I' is sequentially consistent on $\text{dom}(f)$.

Proof. By contradiction. Let $r \in \text{dom}(f)$ be a read in I' such that either (i) the most recent write I'_w to the same location does not have the same value as I'_r , or (ii) there is no previous write and the value of I'_r is not the default value.

In the first case, we have $f(w) < f(r)$ by assumptions (2) and (3). From assumption (1) there must be the most recent write $I_{w'}$ to the same location and with the same value as the read $I_{f(r)}$. Since w' is most recent, we have $f(w) < w' < f(r)$. By assumptions (5) and (4), $f^{-1}(w')$ must exist. Using assumption (3), we have $w < f^{-1}(w') < r$. This is a contradiction with w being the index of the most recent write for the read r in interleaving I' .

Similarly, if I'_r does not have the default value, then there must be $w < f(r)$ such that I_w is a write to the same location. By assumptions (5), (4) and (3), $f^{-1}(w) < r$ is a write to the same location as I'_r . This contradicts case (ii). \square

The following two lemmata show two cases of rearranging actions in an execution with a data race so that the resulting interleaving is an execution with a strong data race. The first lemma handles the case when the execution I contains a data race of a read followed by write. We illustrate the construction on the following program:

```
Thread 1: x := 1; r1 := x; print r1;
Thread 2: r2 := z; print r2; x := 2;
```

Let us consider the interleaving I from Figure 2.4. This interleaving has a data race between the actions in positions 3 and 7 (shaded), i.e., between the read of x in thread 1 and the write to x in thread 2. We will construct a new interleaving J from I in the following way: J is a list of all actions that are strictly

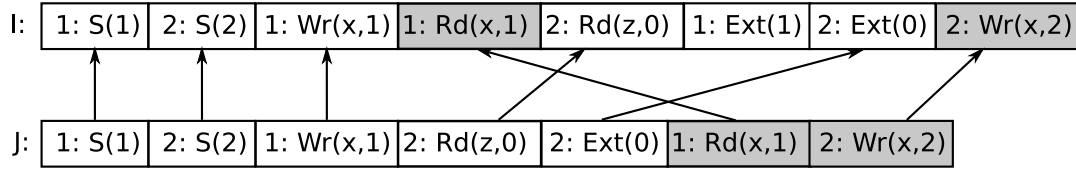


Figure 2.4: Data race to strong data race (read-write case).

before the read of x in I , followed by all actions are strictly after the read, but strictly happen-before the write to x in the second thread. Finally, we append the read and then the write. The arrows in Figure 2.4 show a complete matching between J and I . Note that the resulting interleaving J is an execution of the program. The following lemma identifies a class of executions with a data race such that the procedure above yields an execution with a strong data race.

Lemma 2.12. *Suppose that I is a properly locked sequentially consistent interleaving of traceset T and $r < w$ are indices from $\text{dom}(I)$ such that:*

1. w is a write and r is a read from the same non-volatile location in I ,
2. r does not happen-before w in I ,
3. there is no write w' to the same location as w between r and w in I ,
4. if $i \in \{0, \dots, w\} \setminus \{r\}$ sees write j then $j \leq_{hb}^I i$.

Then the interleaving J defined as

$$l = [i \leftarrow \text{dom}(I). i < w \vee i <_{hb}^I r] \# [r, w]$$

$$J = [I_i \mid i \leftarrow l]$$

is an execution of T .

Proof. Let us take the complete matching χ_l between J and I (Lemma 2.3) and let $S =_{\text{def}} \{0, \dots, |J| - 3\}$. Observe that χ_l is an increasing function on S . By Lemmata 2.8 and 2.9, J is a properly locked interleaving of T . Using Lemma 2.11 with the function $\chi_l|_S$, interleaving J is sequentially consistent on S . To establish sequential consistency of J , we only need to show that the read $|J| - 2 = \chi_l^{-1}(r)$ sees the most recent write in J . We show that if r sees write w' in I , then $\chi_l^{-1}(r)$ sees $\chi_l^{-1}(w')$ in J . Suppose it is not the case. Then there is a write w'' to the same location as w between $\chi_l^{-1}(w')$ and $\chi_l^{-1}(r)$. By monotonicity of χ_l on S

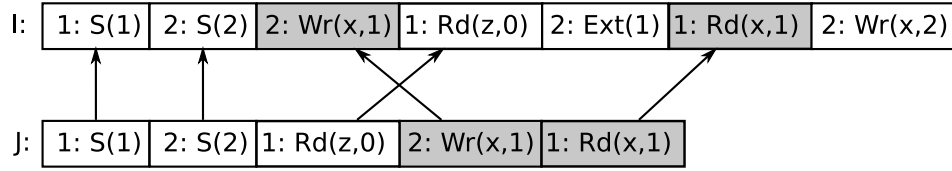


Figure 2.5: Data race to strong data race (write-first case).

we have $w' < \chi_I(w'')$. By assumption (3), $r \not\prec \chi_I(w'')$, so $\chi_I(w'') < r$. This is a contradiction with r seeing w' in I . Similarly, if r sees the default value in I , then $\chi_I^{-1}(r)$ must see the default value in J . \square

The second data race reduction lemma handles the situation when we have two ‘racy’ writes or a data race between a read and a write with the same value. The main difficulty of the first lemma was to prove that the ‘racy’ read sees the most recent write. In this case, it is much easier: If we have a write-write race, sequential consistency follows trivially. On the other hand, if we have a write-read data race with the same value, we move the read to be the last action in the interleaving and the write to be just before that read. Then the read obviously sees the most recent write. For an illustration, see the example in Figure 2.5. The formal details follow:

Lemma 2.13. *Suppose that I is an execution of traceset T and $w < a$ are indices from $\text{dom}(I)$ such that:*

1. w is a write in I and a is a write to the same non-volatile location or a read from the same non-volatile location with the same value as w in I ,
2. w and a are not ordered by happens-before in I ,
3. for any $i, j \in \{0, \dots, a-1\}$, if i sees write j then $j \leq_{hb}^I i$.

Then the interleaving J defined as

$$l = [i \leftarrow \text{dom}(I). i < w \vee i <_{hb}^I a] \# [w, a]$$

$$J = [I_i \mid i \leftarrow l]$$

is an execution of T .

Proof. Let us take the complete matching χ_I between J and I and let $S =_{\text{def}} \{0, \dots, |J| - 3\}$. Using the same argument as in Lemma 2.12, J is properly

locked interleaving of T , and J is sequentially consistent on S . If a is a write in I , then $|J| - 1 = \chi_i^{-1}(a)$ and $|J| - 2 = \chi_i^{-1}(w)$ are writes in J , hence J is sequentially consistent. If a is a read, then the read $|J| - 1$ sees the write $|J| - 2$ in J , and J is sequentially consistent. \square

Proof of Lemma 2.10. Let us have an execution I of T containing a data race. Let us take the shortest prefix I^* that still contains a data race, and let l be the last index in I^* , i.e., $l = |I^*| - 1$. Observe that if $i < j$ are a data race in I^* , then $j = l$, because I^* is the shortest prefix with a data race. Let $i < |I^*|$ be the highest index such that I_i^* conflicts with I_l^* and i does not happen-before l in I^* . The interleaving I^* is an execution of T , using Lemmata 2.8, 2.9, and 2.11 with the identity function on $\text{dom}(I^*)$. Observe that there cannot be any write to the same location as I_l^* between i and l : if there was a write w in I^* such that $i < w < l$, then $i \leq_{hb}^{I^*} w$ because I^* is the shortest prefix with a data race, and $w \leq_{hb}^{I^*} l$ because i is the highest index in I^* that is in data race with l , hence $i \leq_{hb}^{I^*} l$ by transitivity, which contradicts i and l being a data race in I^* .

Depending on the kind of action $\mathcal{A}(I_i^*)$, we will use Lemma 2.12 or 2.13 to construct an execution with a strong data race:

1. If $\mathcal{A}(I_i^*)$ is a read, then $\mathcal{A}(I_l^*)$ must be a write, and we get the execution from Lemma 2.12. The only non-trivial assumption is assumption 4, which follows from data race freedom of all prefixes of I^* .
2. If $\mathcal{A}(I_i^*)$ is a write, then either (i) $\mathcal{A}(I_l^*)$ is a write, or (ii) $\mathcal{A}(I_l^*)$ a read with the same as the value of $\mathcal{A}(I_i^*)$, because there are no writes to the same location between i and l and I^* sequentially consistent. In any case, we apply Lemma 2.13 to get an execution of T with a strong data race.

\square

Corollary 2.14. *A traceset is data race free if and only if it is weakly data race free.*

Chapter 3

Safety of Transformations

The goal of this chapter is to establish the safety of important classes of transformations. We believe that these classes are general enough to express most common compiler transformations, such as common subexpression elimination, dead code elimination, and various types of loop optimisations. Similarly to Manson et al. (2005), we consider a transformation safe if it does not introduce any new behaviours for data race free programs. Note that a safe transformation may reduce the possible behaviours.

We take a trace-semantic approach. We define reordering and elimination transformations as relations on tracesets, and prove that they guarantee the illusion of sequential consistency for data race free programs and provide simple security guarantees for all programs. To ensure compositionality, we show that these semantic transformations preserve data race freedom. We believe that trace-semantics is a good intermediate step for two reasons:

- Language independence. A compiler can reuse the same semantics for all its intermediate languages until the stage where it transforms the high-level synchronisation primitives (locks, unlocks, volatile variables) to low-level instructions (memory fences, atomic compare-and-swaps, etc.).
- Semantic nature of some transformations. For example, the trace-preserving transformations from Section 1.4 are inherently semantic.

The chapter is organised as follows: in Section 3.1, we define the semantic transformations on tracesets. We prove the safety of our semantic transformations in Section 3.2.

3.1 Trace-semantic Transformations

3.1.1 Eliminations

To define semantic read eliminations, we introduce *wildcard traces*. The wildcard traces are generalisations of ordinary traces; each element of a wildcard trace is either an action or a wildcard read $\text{Rd}(x, *)$. We use the wildcards to express that the trace's validity does not depend on the value that the wildcard reads might read.

A trace t is an *instance of* a wildcard trace t^* if $|t| = |t^*|$ and for all $i \in \text{dom}(t)$ we have:

- if t_i^* is an (ordinary, non-wildcard) action, then $t_i^* = t_i$,
- if t_i^* is a wildcard read $\text{Rd}(l, *)$, then $t_i = \text{Rd}(l, v)$ for some $v \in \tau(l)$.

A *wildcard trace belongs to a set of traces* T if all instances of the wildcard trace belong to T .

Similarly to traces, we can generalise interleavings to *wildcard interleavings*: A wildcard interleaving is a sequence of pairs $\langle \theta, a \rangle$, where θ is a thread identifier and a is either an action or a wildcard read. We obtain an *instance of a wildcard interleaving* by replacing each wildcard read by a read from the same location with the value of the most recent write to the same location, or with the default value if there is no earlier write to the same location. A wildcard interleaving I is a *wildcard interleaving of a traceset* T if for each thread θ , the (wildcard) trace of θ belongs to T .

Fact 3.1. *If I is a wildcard interleaving of T then any instance of I is an interleaving of T .*

In general, the converse is not true.

Example 3.1. Let T be the traceset of the program

Thread 1: $y := 1; r1 := x; \text{print } r1$

Thread 2: $r2 := y; x := 1.$

For example, the wildcard traces

$$t_1 = [\text{S}(1), \text{Wr}(y, 1), \text{Rd}(x, *)]$$

$$t_2 = [\text{S}(2), \text{Rd}(y, *), \text{Wr}(x, 1)]$$

belong to T . On the other hand, the wildcard trace

$$t_3 = [S(1), \text{Wr}(y, 1), \text{Rd}(x, *), \text{Ext}(1)]$$

does not belong to T , because some instances of t_3 , such as the trace

$$[S(1), \text{Wr}(y, 1), \text{Rd}(x, 2), \text{Ext}(1)],$$

do not belong to T .

By interleaving wildcard traces of T we get wildcard interleavings of T . For example, by interleaving t_1 and t_2 we get a wildcard interleaving

$$I = [\langle 1, S(1) \rangle, \langle 2, S(2) \rangle, \langle 2, \text{Rd}(y, *) \rangle, \langle 1, \text{Wr}(y, 1) \rangle, \langle 2, \text{Wr}(x, 1) \rangle, \langle 1, \text{Rd}(x, *) \rangle]$$

Assuming that the default value of y is 0, the unique instance of I is the interleaving

$$[\langle 1, S(1) \rangle, \langle 2, S(2) \rangle, \langle 2, \text{Rd}(y, 0) \rangle, \langle 1, \text{Wr}(y, 1) \rangle, \langle 2, \text{Wr}(x, 1) \rangle, \langle 1, \text{Rd}(x, 1) \rangle].$$

Our definition of elimination generalises the examples of eliminations from Section 1.4 by allowing eliminations of actions that are not necessarily adjacent. Moreover, the definition allows eliminations of the last occurrence of some types of actions. The last action eliminations, i.e., points 6, 7 and 8 in the following definition, will become useful for reordering, see Example 3.6 for details.

Definition 3.1. Let t be a wildcard trace and i be an index from $\text{dom}(t)$. We say that the index i is a

1. *redundant read after read* if there is a non-volatile location l , an index $j < i$ and a value v such that t_i and t_j are non-wildcard reads of the value v from the location l , and there is no synchronisation action or other memory access to the location l between j and i ,
2. *redundant read after write* if t_i is a normal read and there is an index $j < i$ such that t_j is a write of the same value to the same location, and there is no synchronisation or other access to the same location between j and i ,
3. *irrelevant read* if t_i is a wildcard read from a non-volatile location,
4. *redundant write after read* if t_i is a normal write and there is an earlier read of the same location with the same value such that there is no synchronisation or other access to the same location in between,

5. *overwritten write* if t_i is a normal write and there is $j > i$ such that t_j is a write to the same location and there is no synchronisation or other access to the same location in between,
6. *redundant last write* if t_i is a normal write and there is no later release action, i.e. an unlock or a volatile write, and no later memory access to the same location,
7. *redundant release* if t_i is a release and there are no later synchronisation or external actions,
8. *redundant external action* if t_i is an external action and there are no later synchronisation or external actions.

An index i is *eliminable* in t if i in t satisfies one of the conditions above.

Once we have a definition of eliminable actions, we define the eliminations on traces and tracesets.

Definition 3.2. Given wildcard trace t and (ordinary) trace t' , the trace t' is an *elimination* of t if there is $S \subseteq \text{dom}(t)$ such that $t' = t|_S$ and all $i \in \text{dom}(t) \setminus S$ are eliminable in t .

Definition 3.3. A set of traces T' is an *elimination of a set of traces* T if each trace $t' \in T'$ is an elimination of some wildcard trace from T .

3.1.1.1 Examples of Eliminations

Our first example illustrates the simplest case of overwritten write elimination.

Example 3.2. Let us consider the program

Thread 0: $x := 1; x := 2$

The traceset of this program is

$$T = \{[], [S(0)], [S(0), \text{Wr}(x, 1)], [S(0), \text{Wr}(x, 1), \text{Wr}(x, 2)]\}$$

Obviously, the first write to x is redundant, so we should be able to eliminate it and obtain the program

Thread 0: $x := 2$

with the traceset

$$T' = \{[], [S(0)], [S(0), \text{Wr}(x, 2)]\}$$

Let us check that T' is an elimination of T : the first two traces from T' are trivial eliminations, because they are in T , and by Definition 3.2, any trace t is an elimination of itself, just take S to be $\text{dom}(t)$. The last trace t' from T' , i.e., $t' = [S(0), \text{Wr}(x, 2)]$, is an elimination of the trace $t = [S(0), \text{Wr}(x, 1), \text{Wr}(x, 2)]$, because we can eliminate the overwritten write $\text{Wr}(x, 1)$. To be precise, let $S = \{0, 2\}$. Then $t|_S = t'$. The definition of elimination requires that all indices in $\text{dom}(t) \setminus S$ are eliminable. Since $\text{dom}(t) \setminus S = \{1\}$ and 1 is eliminable in t (overwritten write), all traces from T' are eliminations of some traces from T .

We can also eliminate a read that does not affect subsequent memory operations:

Example 3.3. Let us take the program

```
Thread 0: r := x; if (r == 1) print(r) else print(1)
```

with the traceset

$$T = \{[], [S(0)]\} \cup \{[S(0), \text{Rd}(x, v)] \mid v \in \tau(x)\} \cup \{[S(0), \text{Rd}(x, v), \text{Ext}(1)] \mid v \in \tau(x)\}$$

The read of x is irrelevant, because we can replace the value of the read with any other value and still get a trace from T . Indeed, the traceset

$$T' = \{[], [S(0)], [S(0), \text{Ext}(1)]\}$$

of the program

```
Thread 0: print 1
```

is an elimination of T , because the first two traces of T' are also in T and the trace $[S(0), \text{Ext}(1)]$ is an elimination of the wildcard trace $[S(0), \text{Rd}(x, *), \text{Ext}(1)]$, which belongs to T .

Finally, we show a more complex example that illustrates elimination of a redundant read after read¹:

Example 3.4. Consider the following program (cf. Figure 5.5):

¹Such a transformation is illegal in the Java Memory Model, see Section 5.5 for details.

```

Thread 0: r1 := x; y := r1
Thread 1: r2 := y;
          if (r2 == 1) {r3 := y; x := r3}
          else x := 1
          print r2

```

Let us call the program P . The traceset T of P is the prefix closure of the set

$$\begin{aligned}
& \{[S(0), \text{Rd}(x, v), \text{Wr}(y, v) \mid v \in \mathbb{N}\} \\
& \cup \{[S(1), \text{Rd}(y, 1), \text{Rd}(y, v), \text{Wr}(x, v), \text{Ext}(1) \mid v \in \mathbb{N}\} \\
& \cup \{[S(1), \text{Rd}(y, v), \text{Wr}(x, 1), \text{Ext}(v)] \mid v \in \mathbb{N} \wedge v \neq 1\}
\end{aligned}$$

Let us remove the redundant read of y in thread 1 and denote the resulting program by P' :

```

Thread 0: r1 := x; y := r1
Thread 1: r2 := y;
          x := 1
          print r2

```

Let T' be the traceset of P' , i.e., T' is the prefix closure of the set

$$\begin{aligned}
& \{[S(0), \text{Rd}(x, v), \text{Wr}(y, v) \mid v \in \mathbb{N}\} \\
& \cup \{[S(1), \text{Rd}(y, v), \text{Wr}(x, 1), \text{Ext}(v) \mid v \in \mathbb{N}\}
\end{aligned}$$

Traceset T' is an elimination of T , because each trace from T' is an elimination of some trace from T : the only traces that are in T' but not in T are the traces $[S(1), \text{Rd}(y, 1), \text{Wr}(x, 1)]$ and $[S(1), \text{Rd}(y, 1), \text{Wr}(x, 1), \text{Ext}(1)]$. Since the latter is an elimination of a redundant read after read from $[S(1), \text{Rd}(y, 1), \text{Rd}(y, 1), \text{Wr}(x, 1), \text{Ext}(1)]$ and the former is an elimination from $[S(1), \text{Rd}(y, 1), \text{Rd}(y, 1), \text{Wr}(x, 1)] \in T$, the traceset of T' is an elimination of the traceset of T .

3.1.2 Reordering

Reordering is an operation that changes the order of execution of some instructions. To preserve meaning, reordering can safely swap only some combinations of statements.

Definition 3.4. We say that a is *reorderable* with b if either:

a reorderable with b	b				
a	$\text{Wr}(x, v)^a$	$\text{Rd}(x, v)^a$	Acquire	Release	External
$\text{Wr}(y, u)^a$	$x \neq y$	$x \neq y$	✓	×	✓
$\text{Rd}(y, u)^a$	$x \neq y$	✓	✓	×	✓
Acquire	×	×	×	×	×
Release	✓	✓	×	×	×
External	✓	✓	×	×	×

^aLocations x and y are not volatile.

Table 3.1: Allowed reordering

1. a is a non-volatile memory access and b is
 - (a) a non-conflicting non-volatile memory access, or
 - (b) an acquire action (a lock or a volatile read) , or
 - (c) an external action; or
2. b is a non-volatile memory access and a is
 - (a) a non-conflicting non-volatile memory access, or
 - (b) a release (an unlock or a volatile write), or
 - (c) an external action.

Table 3.1 summarises the permissible reordering in a more readable form. The table is consistent with the existing guides for writing optimising compilers, for example, see Lea (2008). Note that reorderability is not a symmetric relation. For example, we can reorder $\text{Wr}(x, 1)$ with $L(m)$, but not the opposite.

A traceset T' is a reordering of a traceset T if each trace t' in T' is a permutation of some trace t from T . Moreover, the permutation has to satisfy two conditions:

1. it may only swap reorderable actions,
2. if we apply the inverse permutation to any prefix of t' , i.e., if we leave out from t all the actions that are not in the prefix, then the resulting trace belongs to T .

In the rest of this subsection we will make this definition precise and then we demonstrate the definition on a simple example.

Definition 3.5. Given trace t , a bijection $f : \text{dom}(t) \rightarrow \text{dom}(t)$ is a *reordering function* for t if for all $i < j$ we have that $f(j) < f(i)$ implies that t_j is reorderable with t_i .

It might seem that the definition should require t_i to be reorderable with t_j and not the opposite. This is because the function transforms traces in the opposite way: from traces of the transformed program to traces of the original program. The following two definitions adapt the notion of reordering to tracesets:

Definition 3.6. Given a set of traces T and trace t' , we say that function $f : \text{dom}(t') \rightarrow \text{dom}(t')$ reorders T to t' if f is a reordering function for t' and for any $n \leq |t'|$ we have

$$\left[t'_{f^{-1}(i)} \mid i \leftarrow \text{l dom}(t'). f^{-1}(i) < n \right] \in T. \quad (3.1)$$

To assist with reading of (3.1) note that the list $\left[t'_{f^{-1}(i)} \mid i \leftarrow \text{l dom}(t') \right]$ is the list obtained from t' by permuting its actions by function f , and the condition (3.1) says that f de-permutes all prefixes of t' to some trace in T .

Definition 3.7. A set of traces T' is a reordering of a set of traces T if for each $t' \in T'$ there is a function that reorders T into t' .

The application of the semantic definitions to reordering of independent statements in concrete programs is not straightforward, because we use the semantic elimination to remove the earlier reordered statement and only then we apply the semantic reordering. This captures the observation that if two statements are independent then it should be possible to eliminate the the action of the earlier statement from the prefix that ends with the action of the second statement. This is best demonstrated on an example. Later, in Section 4.3, we will generalise this idea and prove that any syntactic reordering of two consecutive independent statements corresponds to a combination of a semantic elimination and a semantic reordering.

Example 3.5. Let P be the program

```
Thread 0: x := 1; print 1
```

with the traceset

$$T = \{[], [S(0)], [S(0), \text{Wr}(\mathbf{x}, 1)], [S(0), \text{Wr}(\mathbf{x}, 1), \text{Ext}(1)]\}.$$

Since both statements are independent we might wish to transform P to the following program P'

Thread 0: `print 1; x := 1`

with the traceset

$$T' = \{[], [S(0)], [S(0), \text{Ext}(1)], [S(0), \text{Ext}(1), \text{Wr}(\mathbf{x}, 1)]\}.$$

Ideally, we would like to show that T' is a reordering of T , i.e., for any trace $t' \in T'$ there is a permutation function that reorders T to t' . However, this is not the case, because none of the two permutations of the trace $[S(0), \text{Ext}(1)]$ belongs to T . Therefore, traceset T' cannot be a reordering of T .

This is where the elimination comes into play: observe that we can obtain the trace $[S(0), \text{Ext}(1)]$ by eliminating the last write to \mathbf{x} from $[S(0), \text{Wr}(\mathbf{x}, 1), \text{Ext}(1)]$. More precisely, let $\hat{T} = T \cup [S(0), \text{Ext}(1)]$ and observe that \hat{T} is an elimination of T (by Definition 3.3). It remains to check that T' is a reordering of \hat{T} . Let us illustrate this on $t' = [S(0), \text{Wr}(\mathbf{x}, 1), \text{Ext}(1)]$. Let

$$\begin{aligned} f &= \{\langle 0, 0 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle\} \\ t^n &= \left[t'_{f^{-1}(i)} \mid i \leftarrow \text{ldom}(t'). f^{-1}(i) < n \right] \end{aligned}$$

The meaning of t^n is the following: for $n = |t'|$, t^n is obtained from t' by applying permutation f so that $t'_i = t^n_{f(i)}$ or, equivalently, $t'_{f^{-1}(i)} = t^n_i$ for all $i \in \text{dom}(t')$. If $n < |t'|$, then we use the transformation only on the prefix of t' of length n . For an illustration, see Figure 3.1. Since $t^n \in \hat{T}$ for any $n \leq |t'| = 3$, we satisfy Definition 3.6 and f reorders \hat{T} to t' . Similarly, for all other traces t' from T' there is a function that reorders \hat{T} to t' . In fact, Figure 3.1 already shows all reorderings of traces from T' , because all traces from T' are prefixes of $[S(0), \text{Wr}(\mathbf{x}, 1), \text{Ext}(1)]$. The notation $f \downarrow_S$ stands for permutation restriction. We define it formally in Definition 3.13.

We will show a more complete example, which demonstrates a reordering of a read with a later write in one of two threads.

Example 3.6. Let P be the program:

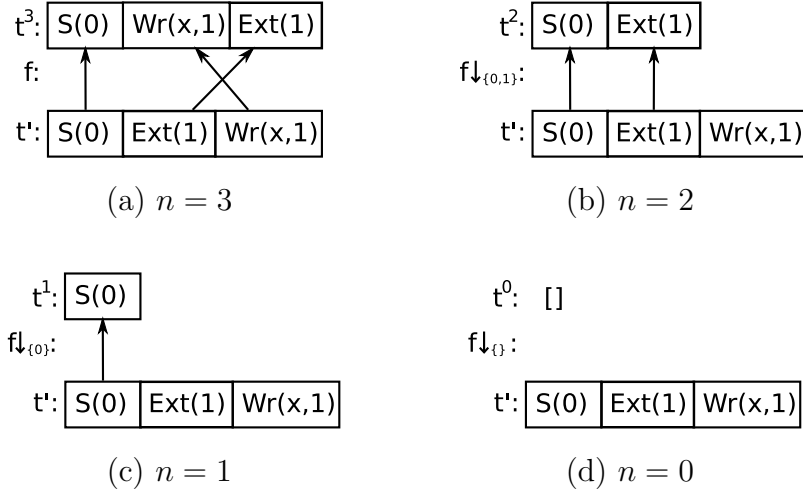


Figure 3.1: Example of reordering traces.

Thread 0: $r1 := y; x := r1$

Thread 1: $r2 := x; y := 1; \text{print } r2;$

Since $r2:=x$ and $y:=1$ are independent, it should be possible to transform P to the following program P' :

Thread 0: $r1 := y; x := r1$

Thread 1: $y := 1; r2 := x; \text{print } r2;$

Programs P and P' have the following tracesets T and T' , respectively:

$$\begin{aligned}
 T &= \{[], [S(0)]\} \cup \{[S(0), \text{Rd}(y, v)] \mid v \in \mathbb{N}\} \cup \{[S(0), \text{Rd}(y, v), \text{Wr}(x, v)] \mid v \in \mathbb{N}\} \\
 &\quad \cup \{[S(1)]\} \cup \{[S(1), \text{Rd}(x, v)] \mid v \in \mathbb{N}\} \cup \{[S(1), \text{Rd}(x, v), \text{Wr}(y, 1)] \mid v \in \mathbb{N}\} \\
 &\quad \cup \{[S(1), \text{Rd}(x, v), \text{Wr}(y, 1), \text{Ext}(v)] \mid v \in \mathbb{N}\} \\
 T' &= \{[], [S(0)]\} \cup \{[S(0), \text{Rd}(y, v)] \mid v \in \mathbb{N}\} \cup \{[S(0), \text{Rd}(y, v), \text{Wr}(x, v)] \mid v \in \mathbb{N}\} \\
 &\quad \cup \{[S(1), [S(1), \text{Wr}(y, 1)]]\} \cup \{[S(1), \text{Wr}(y, 1), \text{Rd}(x, v)] \mid v \in \mathbb{N}\} \\
 &\quad \cup \{[S(1), \text{Wr}(y, 1), \text{Rd}(x, v), \text{Ext}(v)] \mid v \in \mathbb{N}\}
 \end{aligned}$$

We will use the same trick as in the previous example. Since there is no permutation of the trace $[S(1), \text{Wr}(y, 1)]$ in T , we need to obtain the trace by elimination. More specifically, we take the wildcard trace $[S(1), \text{Rd}(x, *), \text{Wr}(y, 1)]$ of T , eliminate the wildcard read of x , and add it to T . The resulting traceset

$$\hat{T} = T \cup \{[S(1), \text{Wr}(y, 1)]\}$$

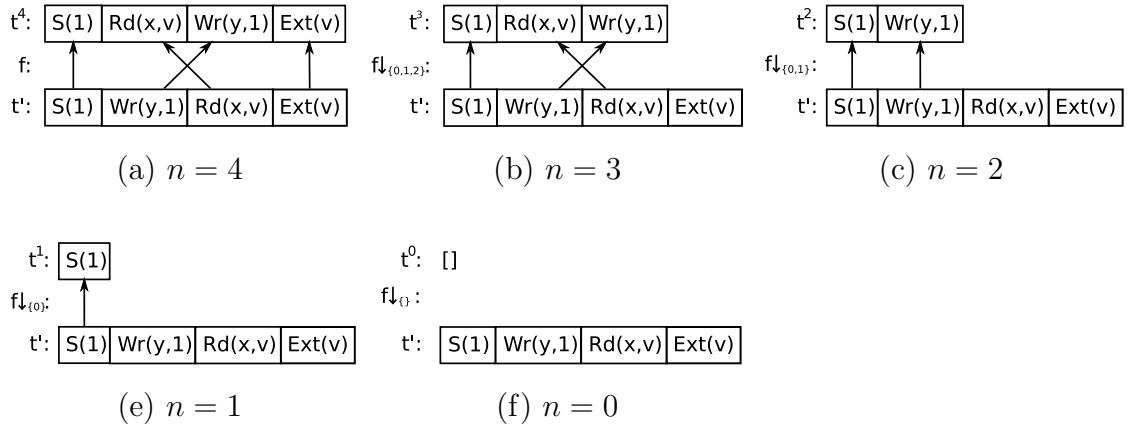


Figure 3.2: Another reordering.

is an elimination of T and any trace in T' is a reordering from \hat{T} . For example, Figure 3.2 shows reordering from \hat{T} to $[S(1), Wr(y, 1), Rd(x, v), Ext(v)]$ using the reordering function

$$f = \{\langle 0, 0 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 3, 3 \rangle\}$$

where $v \in \mathbb{N}$.

3.2 Safety of Transformations

In this section, we show that both the elimination and reordering transformations have these properties:

- Any execution of the transformed traceset has the same behaviour as some execution of the original traceset, provided that the original program was data race free.
- The transformations preserve data race freedom.
- The transformations cannot introduce values out-of-thin-air.

To prove the first two properties, we take an arbitrary execution of the transformed program and construct an execution of the original program that has the same behaviour. The construction is similar for both the elimination and the reordering transformations: we decompose the execution of the transformed program into traces for each thread, use the definitions of transformed traceset

(Definitions 3.3 and 3.7) to obtain ‘untransformed’ traces of the original traceset and then we compose the untransformed traces back into an ‘untransformed’ interleaving so that the order of the external and synchronisation actions is preserved. Then we prove that the constructed interleaving is an execution of the original traceset, if the original traceset was data race free. Moreover, if the happens-before order of the constructed execution relates two memory accesses to the same location, then the corresponding actions in the execution of the transformed program are also related by the happens-before order. This guarantees that the transformed program is data race free if the original program was data race free.

While the construction of the ‘untransformed’ interleaving is similar for both the reordering and the elimination, the transformations require different approaches to prove that the interleavings are executions. In the case of elimination (Subsection 3.2.1), we can prove sequential consistency directly, because the untransformation of elimination embeds the happens-before order on memory accesses to the same location. The main technical difficulty in the proof lies in proving that the extra actions introduced by the untransformation of elimination do not break sequential consistency. On the other hand, the reordering untransformation (Subsection 3.2.2) does not preserve happens-before order on pairs of actions on the same variable in general, because the roach-motel reordering can add happens-before edges. Therefore, we cannot use the direct proof and instead we prove the safety by induction on the size of the interleaving of the transformed program.

Finally, we prove that the transformations cannot introduce origins of values. We say that a traceset is an origin for v if there is a trace that contains a write of v or an output of v without any preceding read of v . In Subsection 3.2.3, we will show that if a program is not an origin of v then its transformation cannot be an origin for v . This gives a simple out-of-thin-air-guarantee: if a program does not contain v explicitly, then it is not an origin for v . Since transformations cannot introduce origins, no transformation of the program can output v . The rest of this section contains detailed descriptions and proofs based on these ideas.

3.2.1 Elimination

Here we prove that given a data race free traceset T , its elimination T' , and an execution of T' , we can ‘untransform’ the execution so that the ‘untransformation’ is an execution of T with the same behaviour as the execution of T' . Formally, we will prove the following two theorems:

Theorem 3.1. *Let traceset T' be an elimination of a data race free traceset T . Then T' is data race free.*

Theorem 3.2. *Suppose that traceset T' is an elimination of a data race free traceset T . Then any execution of T' has the same behaviour as some execution of T .*

The first step is the definition of the untransformation: our definition of elimination on tracesets (Definition 3.3) only describes eliminations for traces. Since we perform untransformations on interleavings rather than on traces, we need to lift the definition of eliminations to interleavings. A naïve definition would require that the eliminated interleaving is just a sublist with some eliminable actions left out. However, this does not guarantee sequential consistency for volatile locations in the untransformed interleaving, because the untransformation might ‘introduce’ a volatile write action, as shown in Example 3.7. Instead we will allow the untransformation to reorder actions while preserving the program order and the order of synchronisation and external actions. Moreover, all the release and external actions introduced by the untransformation must be ordered after the synchronisation and external actions from the interleaving of the transformed program.

The precise definitions follow:

Definition 3.8. An *index i is eliminable in an interleaving I* if the corresponding index in the trace of $\mathcal{T}(I_i)$, i.e., the index $|\{j \mid j < i \wedge \mathcal{T}(I_i) = \mathcal{T}(I_j)\}|$, is eliminable in the trace of $\mathcal{T}(I_i)$ in I .

Definition 3.9. Function f is an *unelimination* function from interleaving I' to wildcard interleaving I if f is a complete matching f between I' and I such that

1. if $i < j \in \text{dom}(I')$ and $\mathcal{T}(I'_i) = \mathcal{T}(I'_j)$ then $f(i) < f(j)$,
2. if $i < j \in \text{dom}(I')$ and $\mathcal{A}(I'_i), \mathcal{A}(I'_j)$ are synchronisation or external actions then $f(i) < f(j)$,

3. if $i \notin \text{rng}(f)$, $j \in \text{dom}(I) \setminus \text{rng}(f)$ and $\mathcal{A}(I_i)$, $\mathcal{A}(I_j)$ are synchronisation or external actions, then $i < j$,
4. if $i \notin \text{rng}(f)$, then i is eliminable in I .

To use the unelimination, we have to show that for each interleaving I' of a transformed traceset there is an unelimination to some interleaving of the original traceset. We construct the function f and the uneliminated interleaving in three steps: we decompose the interleaving I' into individual threads, then we obtain ‘uneliminated’ traces for each thread, and finally we interleave the ‘uneliminated’ traces of the threads so that we preserve the order of synchronisation and external actions from I' while ordering all introduced synchronisation and external actions after the synchronisation and external actions from I' .

Example 3.7. Consider the program

```
Thread 1: v := 1; y := 1;
Thread 2: r1 := x; r2 := v; print r2;
```

By our definition of elimination on tracesets, we can eliminate the last release $v:=1$ in thread 1 and the irrelevant read $r1:=x$ in thread 2:

```
Thread 1: y := 1;
Thread 2: r2 := v; print r2;
```

Let

$$I' = [\langle 1, S(1) \rangle, \langle 2, S(2) \rangle, \langle 1, \text{Wr}(y, 1) \rangle, \langle 2, \text{Rd}(v, 0) \rangle, \langle 2, \text{Ext}(0) \rangle]$$

Figure 3.3 shows one possible construction of unelimination I of I' , where the unelimination function is a composition of the functions $f_{I'}$, f_e and f_I . For example, the unelimination function maps 2 to 6, i.e., it moves the action in position 2 in I' , i.e., $I'_2 = \langle 1, \text{Wr}(y, 1) \rangle$, to the last position in I . Note that we cannot just insert the eliminated actions into I' to get the unelimination, because we would have to insert the write $\text{Wr}(v, 1)$ between the start of thread 1 and the write to y and this would break sequential consistency for the read of v .

A more precise description of the construction follows:

Lemma 3.2. *Let traceset T' be an elimination of traceset T and I' an interleaving of T' . Then there is a wildcard interleaving I of T and a function f such that f is an unelimination function from I' to I .*

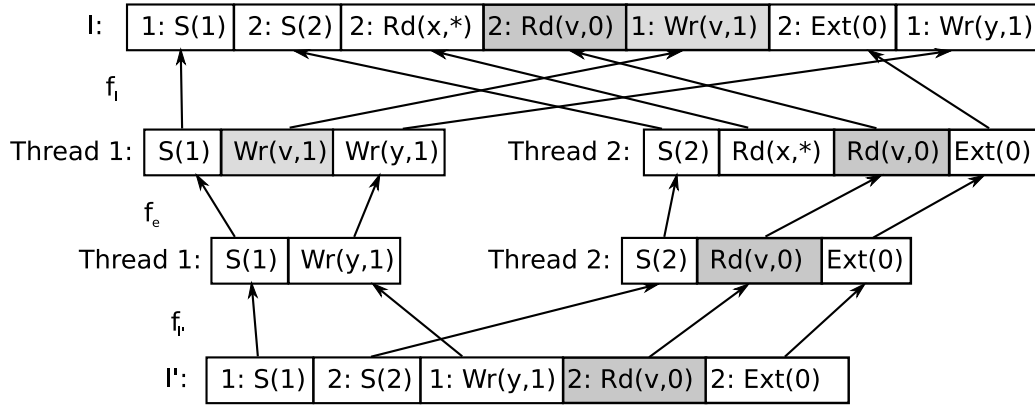


Figure 3.3: Unelimination construction.

Proof. Let $t^{I',\theta}$ be the trace of θ in I' . First, we define a set

$$N' = \{\langle \theta, i \rangle \mid 0 \leq i < |t^{I',\theta}|\}$$

For a pair $n' = \langle \theta, i \rangle$ we will use the notation $\mathcal{T}(n')$ to refer to the thread identifier θ and $\iota(n')$ to the index i . We construct function $f_{I'} : \text{dom}(I') \rightarrow N'$ so that $f(i)$ is the pair of the thread of I'_i and the position of the action I'_i in the trace of $\mathcal{T}(I'_i)$. In other words, by Lemma 2.4 there is a bijection $f_{I'}$ from $\text{dom}(I')$ to N' such that

$$I'_i = \langle \mathcal{T}(f_{I'}(i)), t_{\iota(f_{I'}(i))}^{I', \mathcal{T}(f_{I'}(i))} \rangle \quad (3.2)$$

$$0 \leq i < j < |I'| \text{ and } \mathcal{T}(I'_i) = \mathcal{T}(I'_j) \text{ implies } \iota(f_{I'}(i)) < \iota(f_{I'}(j)) \quad (3.3)$$

Using Lemma 2.4 and the assumption that T' is an elimination of T , for each thread θ there is a wildcard trace t^θ of T and an increasing matching f^θ between $t^{I',\theta}$ and t^θ such that $i \in \text{dom}(t^\theta) \setminus \text{rng}(f)$ are eliminable in t^θ .

We take the set

$$N = \{\langle \theta, i \rangle \mid 0 \leq i < |t^\theta|\}$$

as the set of unique identifiers of the actions in the wildcard traces and we define $f_e : N' \rightarrow N$ to be

$$f_e(\langle \theta, i \rangle) = \langle \theta, f^\theta(i) \rangle. \quad (3.4)$$

Since f^θ are complete matchings, we have

$$t_{\iota(n')}^{I', \mathcal{T}(n')} = t_{\iota(f_e(n'))}^{\mathcal{T}(n')} = t_{\iota(f_e(n'))}^{\mathcal{T}(f_e(n'))} \text{ for } n' \in N' \quad (3.5)$$

In the final step, we construct an order on the actions of the uneliminated threads so that we satisfy the requirements of Definition 3.9. We start with the

total order on N induced by the total order on synchronisation and external actions from I' :

$$\leq_{I'}^N = \{ \langle i, j \rangle \mid \exists u \leq v \in \text{dom}(I'). i = f_e(f_{I'}(u)) \text{ and } j = f_e(f_{I'}(v)) \text{ and } \mathcal{A}(I'_u) \text{ and } \mathcal{A}(I'_v) \text{ are external or synchronisation actions} \}$$

Note that this order does not relate the synchronisation and external actions, which were introduced by the unelimination on the traces of the individual threads. To satisfy the requirements for these actions (point 3 of Definition 3.9) we extend the total order $\leq_{I'}^N$ to place all introduced external and synchronisation actions after the ones from I' , where the introduced actions are arbitrarily totally ordered among themselves. Let us denote this total order on the set $\{ \langle \theta, i \rangle \mid \langle \theta, i \rangle \in N \text{ and } \mathcal{A}(t_i^\theta) \text{ is a synchronisation or external action} \}$ by \leq_{se}^N . Let \leq_{po}^N be the program order on N induced by ordering of actions in each thread:

$$\leq_{po}^N = \{ \langle \langle \theta, i \rangle, \langle \theta, j \rangle \rangle \mid \langle \theta, i \rangle, \langle \theta, j \rangle \in N \wedge i \leq j \}$$

Note that $\langle \theta, i \rangle \leq_{se}^N \langle \theta, j \rangle$ implies $i \leq j$, so the orders \leq_{se}^N and \leq_{po}^N are consistent. Therefore, the order

$$\leq^N = (\leq_{se}^N \cup \leq_{po}^N)^*$$

is a partial order (see Fact 3.3 below). We construct a topological sort of \leq^N and order the set N by the topological sort into a list l . For $n \in N$, let $f_I(n)$ be the unique index i of n such that $l_i = n$. Let us define

$$I = \left[\left\langle \mathcal{T}(f_I^{-1}(i)), t_{\iota(f_I^{-1}(i))}^{\mathcal{T}(f_I^{-1}(i))} \right\rangle \mid i \leftarrow [0, \dots, |N| - 1] \right] \quad (3.6)$$

$$f = f_I \circ f_e \circ f_{I'} \quad (3.7)$$

By 3.2, 3.4, 3.5, 3.6, we have that f is a matching. Since $f_{I'}$, and f_e are bijections, and f_I is a complete matching, the matching f must be complete.

It is straightforward to check that I and f have the other properties required by Definition 3.9. \square

Fact 3.3. *Let R be a partial order on S and Q a total order on $T \subseteq S$ consistent with R . Then $(R \cup Q)^+$ is a partial order on $S \cup T$.*

The harder part is establishing that the unelimination is an execution. We start with locking:

Lemma 3.4. *Let I' be a properly locked interleaving, I be an interleaving of traceset T and f an unelimination function from I' to I . Then I is properly locked.*

Proof. Let $\mathcal{A}(I_i) = L(m)$. Since locks are never eliminable, there must be i' such that $f(i') = i$. We have:

$$\begin{aligned} |\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = L(m)\}| &= && \text{by Definition 3.9} \\ |\{j \mid j < i' \wedge \mathcal{T}(I'_j) = \theta \wedge \mathcal{A}(I'_j) = L(m)\}| &= && \text{by proper locking of } I' \\ |\{j \mid j < i' \wedge \mathcal{T}(I'_j) = \theta \wedge \mathcal{A}(I'_j) = U(m)\}| &= && \text{by Definition 3.9} \\ |\{j \mid j < i \wedge \mathcal{T}(I_j) = \theta \wedge \mathcal{A}(I_j) = U(m)\}| & & & \end{aligned}$$

□

To prove sequential consistency, we use well-behaved interleavings, which were employed by Manson et al. (2005); Boehm and Adve (2008). An interleaving is well-behaved if each read from the interleaving has the same value as some most recent write in the happens-before order. We generalise this definition to wildcard interleavings and eliminations.

Definition 3.10. Suppose that I is a wildcard interleaving. Let S be a subset of $\text{dom}(I)$ and r an index from S . We say that r is *well-behaved* in the wildcard interleaving I on S if $\mathcal{A}(I_r) = \text{Rd}(l, v)$ implies that either (i) I_r is a wildcard read, i.e., $v = *$, or (ii) there is $w \in S$ such that $w \leq_{hb}^I r$, $\mathcal{A}(I_w) = \text{Wr}(l, v)$, and there is no write that ‘happens-in-between’, i.e., there is no $w' \in S$ such that $I_{w'}$ is a write to l , $w \neq w'$ and $w \leq_{hb}^I w' \leq_{hb}^I r$, or (iii) I_r is a read of the default value and there is no write from S to the same location that happens-before r in I .

A wildcard interleaving I is *well-behaved on S* if each $j \in S$ is well-behaved in I on S . Moreover, I is *well-behaved* if it is well-behaved on $\text{dom}(I)$.

The well-behaved interleavings are useful, because they are sequentially consistent if they are interleavings of a data race free program. Before we prove this result, we state two useful observations:

Lemma 3.5. *Suppose that I is an interleaving, $w \in \text{dom}(I)$ is an overwritten write, and there is $r \in \text{dom}(I)$ such that $w \leq_{hb}^I r$ and $\mathcal{A}(I_r)$ is a read from the same location as $\mathcal{A}(I_w)$. Then there is w' such that $\mathcal{A}(I_{w'})$ is a write to the same location and $w \leq_{po}^I w' \leq_{hb}^I r$. Moreover, w' is not a redundant write after read.*

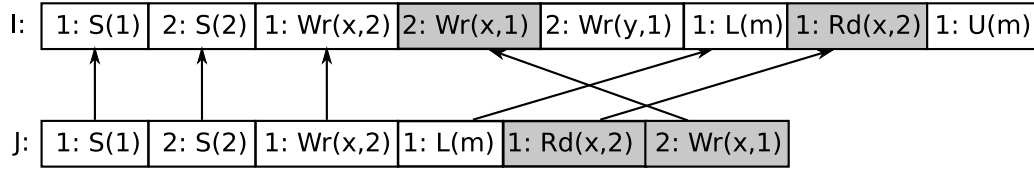


Figure 3.4: Fixing sequential consistency in a well-behaved interleaving.

Proof. This follows directly from the definition of redundant and overwritten writes and happens-before. \square

The second observation describes a construction of an execution with a data race from an almost sequentially consistent interleaving with a data race. The construction is very similar to the one used in the proof of equivalence of strong and weak data race freedom (Lemmata 2.12 and 2.13). The idea is that we can move the first write that breaks sequential consistency of a well-behaved interleaving right behind the first read that does not see the most recent write while removing all actions between the read and the write that do not happen-before the read. For an example, see Figure 3.4.

Lemma 3.6. *Suppose that I is a properly locked interleaving of traceset T and $w < r$ are indices from $\text{dom}(I)$ such that:*

1. w is a write and r is a read from the same non-volatile location in I ,
2. w does not happen-before r in I ,
3. either (i) I_r has the default value and there is no w' such that $w' < w$ or $w' \leq_{hb}^I r$, and $I_{w'}$ is a write to the same location as I_r , or (ii) there is $w' \leq_{hb}^I r$ such that $I_{w'}$ is a write to the same location and with the same value as I_r , and there is no write w'' to the same location as I_r such that $w' < w'' \leq_{hb}^I r$ or $w' < w'' < w$.
4. each $i < r$ sees the most recent write in I , and if i sees write j then $j \leq_{hb}^I i$.

Then the interleaving J defined as

$$l = [i \leftarrow \text{dom}(I). i < w \vee i <_{hb}^I r] \# [r, w]$$

$$J = [I_i \mid i \leftarrow l]$$

is an execution of T .

Proof. Let us take the complete matching χ_l between J and I (Lemma 2.3) and let $S =_{\text{def}} \{0, \dots, |J| - 3\}$. Using the same argument as in Lemma 2.12, J is properly locked interleaving of T , and J sequentially consistent on S (Lemmata 2.8, 2.9 and 2.11). It remains to show that the index $|J| - 2 = \chi_l^{-1}(r)$ sees the most recent write in J . This follows from the assumption (3). If r satisfies part (i) of the assumption, then $|J| - 2$ must see the default value in J : if not, then there would have to be a write $w'' < |J| - 2$ to the same location as the read $|J| - 2$ in J . By the definition of l , we have $\chi_l(w'') < w$ or $\chi_l(w'') <_{hb}^I r$. This contradicts the assumption (i). On the other hand, if r satisfies (ii) of the assumption (3), then $|J| - 2$ must see $\chi_l^{-1}(w')$ in J . Otherwise, there would be write w'' between $\chi_l^{-1}(w')$ and $|J| - 2$ in J . By monotonicity of χ_l on S , $w' < \chi_l(w'')$. Moreover, by the definition of l , $\chi_l(w'') < w$ or $\chi_l(w'') <_{hb}^I r$, which contradicts our assumption. \square

Lemma 3.7. *Suppose I is a wildcard interleaving of a data race free traceset T and $S \subseteq \text{dom}(I)$ is a set such that $i \in \text{dom}(I) \setminus S$ implies that i is eliminable in I . If I is well-behaved on S then I is sequentially consistent.*

Proof. We prove by induction on i that for any i :

1. i sees the most recent write,
2. if i is a redundant write after read in I , then the most recent write from S to the same location as I_i has the same value, or there is no previous write to the same location in S and I_i has the default value.

Let us assume that all $j < i$ satisfy both (1) and (2).

We prove (1) by contradiction. Suppose that $i \in \text{dom}(I)$ does not see the most recent write in I . Note that i cannot be a redundant read after read nor a redundant read after write: Suppose that i is a redundant read after read. Then there must be a previous read r' of the same value in the same thread as i such that there is no synchronisation or other access of the same location between r' and i in the same thread. Since i does not see the most recent write, then there must be a write w to the same location as i such that $r' < w < i$. From the induction hypothesis and data race freedom of T we have $r' \leq_{hb}^I w$. This is a contradiction with the definition of redundant read after read: if w is in the same thread as i then there is a memory access to the same location between r' and i ; if $\mathcal{T}(I_w) \neq \mathcal{T}(I_i)$ then there must be a release action between r' and i . Using a similar argument, i cannot be a redundant read after write.

The index i must fall into one of the cases from Definition 3.10. It cannot be the case (i), because a wildcard read always sees the most recent value.

Suppose that i satisfies (ii), i.e., there is a most recent write $w \in S$ in happens-before order to the same location with the same value. Because i does not see the most recent write, there must be an index w' between w and i such that $I_{w'}$ is a write with a different value, but to the same location as the read I_i .

First, we prove by contradiction that for any w' , if $\mathcal{A}(I_{w'})$ is a write to the same location as $\mathcal{A}(I_i)$, $w < w' \leq_{hb}^I i$, then the value of $\mathcal{A}(I_i)$ and $\mathcal{A}(I_{w'})$ are the same and w' is a redundant write after read. Suppose that this is not the case and $w' \leq_{hb}^I i$ is the highest index that does not satisfy the implication. We have these cases:

1. If $w' \in S$, then we have a contradiction with the assumption (ii) of well-behavedness of i on S , because $w \leq_{hb}^I w'$ by data race freedom and $w' \leq_{hb}^I i$ by assumption.
2. w' cannot be a redundant write before write by Lemma 3.5.
3. w' cannot be a last write, because there cannot be any release or memory access to the same location after w' in the same thread as $\mathcal{T}(w')$,
4. If w' is a redundant write after read, then there is $w'' \in S$ with the same value as $\mathcal{A}(I_{w'})$ such that $w \leq w'' < w' < i$ (by the second part of the induction hypothesis). From the second part of the induction hypothesis and data race freedom we have $w \leq_{hb}^I w'' \leq_{hb}^I w' \leq_{hb}^I i$. Since w is most recent write from S in happens-before relation, we have $w = w''$. As a result, the value of w' must be the same as the value of w , which is the same as the value of i .

By the observation above, there must be w' such that $w < w' < i$, $\mathcal{A}(I_{w'})$ is a write action to the same location as $\mathcal{A}(I_i)$, and $w' \not\leq_{hb}^I i$. Let us take the lowest w' satisfying these conditions. Applying Lemma 3.6 (taking w' as w and i as r) gives an execution of T with a data race. This contradicts data race freedom of T .

Suppose that i satisfies (iii), i.e., there is no write that happens-before i , and $\mathcal{A}(I_i)$ has the default value. Then we use the same construction as in the previous case, except that we take w' to be the lowest index such that $\mathcal{A}(I_{w'})$ writes the

same location as $\mathcal{A}(I_i)$ and $w' < i$, but $w' \not\leq_{hb}^I i$. Using Lemma 3.6, we obtain an execution of T with a data race.

It remains to prove the part (2) of the induction hypothesis for i : Let i be a redundant write after read in I . Then there must be a previous read r in the same thread with the same value with no synchronisation or memory access to the same location in between. By the data race freedom of T , $I|_{\{0, \dots, i\}}$ does not have data races, so there cannot be any write to the same location between r and i . Note that r cannot see a last write or an overwritten write. As a result, it must see a default write or a write from S or a redundant write after read. \square

Lemma 3.8. *Suppose that f is an unelimination function from an interleaving I' to an interleaving I . Let us have indices $i, j \in \text{dom}(I')$. Then $i \leq_{hb}^I j$ if and only if $f(i) \leq_{hb}^I f(j)$.*

Proof. We get the direction “ \Rightarrow ” by straightforward induction on the definition of happens-before.

“ \Leftarrow ”: Let $f(i) \leq_{hb}^I f(j)$. We prove by induction on the definition on happens-before that $f(i) \leq_{hb}^I k$ implies that either $f(i) \leq_{po}^I k$ or there is $a \in \text{dom}(I')$ such that $\mathcal{A}(I'_a)$ is an acquire action, $i \leq_{hb}^{I'} a$ and $f(a) \leq_{po}^I k$. Letting $k = f(j)$ yields the desired result. \square

Given an interleaving I , we say that the most recent write happens-before r in I if $r \in \text{dom}(I)$, and if $\mathcal{A}(I_r)$ is a read then either r sees a default value, or r sees some w in I and $w \leq_{hb}^I r$.

Lemma 3.9. *Let traceset T' be an elimination of a data free traceset T and f an unelimination function from an execution I' of T' to a wildcard interleaving I of T . Suppose that for each index $r \in \text{dom}(I')$, the most recent write happens-before r in I' . Then I is sequentially consistent.*

Proof. Note that I is well-behaved on $\text{rng}(f)$ in I :

- if $\mathcal{A}(I_i)$ is a wildcard read, then i is well-behaved,
- if $f^{-1}(i)$ sees the most recent write w in I' , then $f(w)$ satisfies the condition (ii) of Definition 3.10 (taking $f(w)$ as w and using Lemma 3.8).
- if $f^{-1}(i)$ sees the default value in I' , then there is no write $w' \in S$ before i in I .

Using Lemma 3.7 gives sequential consistency. \square

Theorem 3.1. *Let traceset T' be an elimination of a data race free traceset T . Then T' is data race free.*

Proof. Suppose that I' is a shortest execution of T' with a data race. Let f be an unelimination function from I' to some I (existence is guaranteed by Lemma 3.2).

If for all $r \in \text{dom}(I')$, the most recent write happens-before r in I' , then I is sequentially consistent (Lemma 3.9). In fact, I is an execution (Lemma 3.4). If a and b are a data race in I' , then $f(a)$ and $f(b)$ are a data race in I (Lemma 3.8). This is a contradiction with data race freedom of T .

On the other hand, assume that the most recent write does not happen before some r in I' . Note that $r = |I'| - 1$ and all data races in I' must involve r . Otherwise, I' would not be the shortest interleaving with a data race. Let w be the most recent write to the same location as I'_r . We construct a new interleaving from I so that it contains all actions that happen-before the image of r or the image of w followed by I'_w and I'_r :

$$l = [i \leftarrow \text{l dom}(I). i <_{hb}^I f(r) \vee i <_{hb}^I f(w)] \# [f(w), f(r)]$$

$$\hat{I} = [I_i \mid i \leftarrow l]$$

Notice that the prefix $\hat{I}|_{\{0, \dots, |\hat{I}|-2\}}$ of \hat{I} is well-behaved on $\chi_l^{-1}(\text{rng}(f))$ in \hat{I} . By Lemma 3.7 and 3.8, \hat{I} is sequentially consistent on $\{0, \dots, |\hat{I}|-2\}$. Since $\mathcal{A}(I'_r)$ and $\mathcal{A}(I'_w)$ have the same value, the last element of \hat{I} sees the most recent write, too. Using Lemmata 3.4 and 2.8 to establish proper locking, \hat{I} is an execution of T with a data race. This is a contradiction. \square

Lemma 3.10. *Let traceset T' be an elimination of a data free traceset T and f an unelimination function from an execution I' of T' to a wildcard interleaving I of T . Then I is sequentially consistent.*

Proof. The result follows directly from Theorem 3.1 and Lemma 3.9. \square

Theorem 3.2. *Suppose that traceset T' is an elimination of a data race free traceset T . Then any execution of T' has the same behaviour as some execution of T .*

Proof. Let I' be an execution of T' . By Lemma 3.2, there is interleaving I of T and a function f such that f is an unelimination function from I' to I . Using Lemmata 3.4 and 3.10, I is an execution. Let i be the index of the last external action in I' . Then the prefix $I|_{\{0, \dots, f(i)\}}$ of I is an execution with the same behaviour as I' (see parts 2 and 3 of Definition 3.9). \square

3.2.2 Reordering

In this section, we will give detailed proofs of the following two theorems:

Theorem 3.3. *Suppose that traceset T' is a reordering of a data race free traceset T . Then any execution of T' has the same behaviour as some execution of T .*

Theorem 3.4. *Let traceset T' be a reordering of data race free traceset T . Then T' is data race free.*

When reasoning about reordering of interleavings we often need to relate indices of actions in traces of individual threads to indices of actions in interleavings. To describe the relationship, we will use index lists and index sets for individual threads. The index list (resp. index set) of a given thread in a given interleaving is a list (resp. set) of positions in the interleaving such that the action at the position belongs to the thread. More precisely:

Definition 3.11. The *index list of thread θ in interleaving I* is the set

$$\text{ldom}_\theta(I) =_{\text{def}} [i \leftarrow \text{ldom}(I). \mathcal{T}(I_i) = \theta].$$

The *index set of θ in I* is the set

$$\text{dom}_\theta(I) =_{\text{def}} \{i \mid i \in \text{dom}(I). \mathcal{T}(I_i) = \theta\}.$$

Observe that index lists of threads are always monotone. That is, each member of the list is greater than the previous one, or in other words, the index function (see Definition 2.14) of the index list is a strictly increasing function. Moreover, if I is an interleaving, θ a thread and t^θ the trace of θ in I then the function $f = \chi_{\text{ldom}_\theta(I)}$ is a strictly increasing bijection from $\text{dom}(t^\theta)$ to $\{i \mid i \in \text{dom}(I) \wedge \mathcal{T}(I_i) = \theta\}$, i.e., it assigns to each index i to t^θ an index $f(i)$ to I such that the thread of $\mathcal{T}(I_{f(i)}) = \theta$ and $\mathcal{A}(I_{f(i)}) = t_i^\theta$.

Similarly to the eliminations, we will generalise the notion of reordering to interleavings. Our approach is straightforward: similarly to reordering traces,

we will define a function that will describe how to ‘shuffle’ the actions in the transformed interleaving to get an interleaving of the original program. We will call this function ‘unordering’, because it permutes actions in an interleaving of the reordered programs to obtain an interleaving of the original program. We require that whenever we restrict an unordering on an interleaving to actions of one thread, we get a reordering function on traces, as defined by Definition 3.6. To express this property we will define what it means to restrict a permutation. First, observe that there is only one way to construct a bijection that preserves ordering:

Fact 3.11. *Let S, T be finite sets of the same cardinality and \leq_S (resp. \leq_T) a total order on S (resp. T). Then there is a unique monotone bijection from S to T .*

Proof. By induction on the cardinality of S . □

Definition 3.12. Let f and g be two bijections with the same domain. Let $<$ be an irreflexive total order on some superset of $\text{rng}(f)$ and $\text{rng}(g)$. We say that f and g agree on the order $<$ if for all $i, j \in \text{dom}(f)$ we have $f(i) < f(j)$ if and only if $g(i) < g(j)$.

Definition 3.13. Suppose we have a bijection $f : \{0, \dots, n\} \rightarrow \{0, \dots, n\}$ and a set $S \subseteq \text{dom}(f)$. A bijection g from S to $\{0, \dots, |S| - 1\}$ is a *permutation restriction* of f to S if $f|_S$ and g agree on $<$. We denote the restriction of f to S by $f \downarrow_S$.

Using our new vocabulary, Fact 3.11 has an interesting implication:

Corollary 3.12. *For every bijection $f : \{0, \dots, n\} \rightarrow \{0, \dots, n\}$ and a set $S \subseteq \text{dom}(f)$ there is a unique permutation restriction of f to S .*

Note that here is a direct way to compute the restriction of a permutation:

Lemma 3.13. *For every bijection $f : \{0, \dots, n\} \rightarrow \{0, \dots, n\}$ and a set $S \subseteq \text{dom}(f)$ the equality*

$$f \downarrow_S = \chi_{[i \leftarrow [0, \dots, n], i \in S]}^{-1} \circ f$$

holds.

Proof. Observe that $\chi_{[i \leftarrow [0, \dots, n]. i \in S]}^{-1}$ is a monotone bijection from $f(S)$ to $\{0, \dots, |S| - 1\}$, and $f|_S$ is a bijection from S to $f(S)$. As a result, $\chi_{[i \leftarrow [0, \dots, n]. i \in S]}^{-1} \circ f$ is a bijection from S to $\{0, \dots, |S| - 1\}$. Moreover, $\chi_{[i \leftarrow [0, \dots, n]. i \in S]}^{-1} \circ f$ agrees with $f|_S$ on $<$. Using Corollary 3.12, it must equal to $f \downarrow_S$. \square

Note that the permutation restriction is not a permutation unless S is an initial segment of the set of natural numbers. To get a function that is a permutation, we need to compose the permutation restriction with the function that maps i to the i -th lowest element of S , i.e., the index function of the list $[i \leftarrow [0, \dots, n]. i \in S]$. The following definition uses this observation to define consistency of an unordering on an interleaving with reordering functions on the traces of individual threads.

Definition 3.14. Let T be a traceset and I' an interleaving. We say that function $f : \text{dom}(I') \rightarrow \text{dom}(I')$ is an *unordering* from I' to T if it has the following properties:

1. f is a complete matching,
2. if $i < j \in \text{dom}(I')$, $\mathcal{T}(I'_i) = \mathcal{T}(I'_j)$ and $\mathcal{A}(I'_i), \mathcal{A}(I'_j)$ are not reorderable, then $f(i) < f(j)$,
3. if $i < j \in \text{dom}(I')$ and $\mathcal{A}(I'_i), \mathcal{A}(I'_j)$ are synchronisation or external actions, then $f(i) < f(j)$.
4. for each thread θ , the function f restricted to actions of θ , i.e.,

$$f \downarrow_{\text{dom}_\theta(I')} \circ \chi_{\text{id}_{\text{dom}_\theta(I')}},$$

reorders T into the trace of θ in I' (see Definition 3.6).

Naturally, we have defined the unordering so that it exists for all reordered tracesets. The construction is similar to the construction of an unelimination: we decompose an interleaving of a reordered program to the traces of individual threads, then ‘unorder’ each thread separately and finally interleave the ‘un-ordered’ traces so that the order of synchronisation and external actions is preserved.

Example 3.8. Consider the traceset \hat{T} of the following program

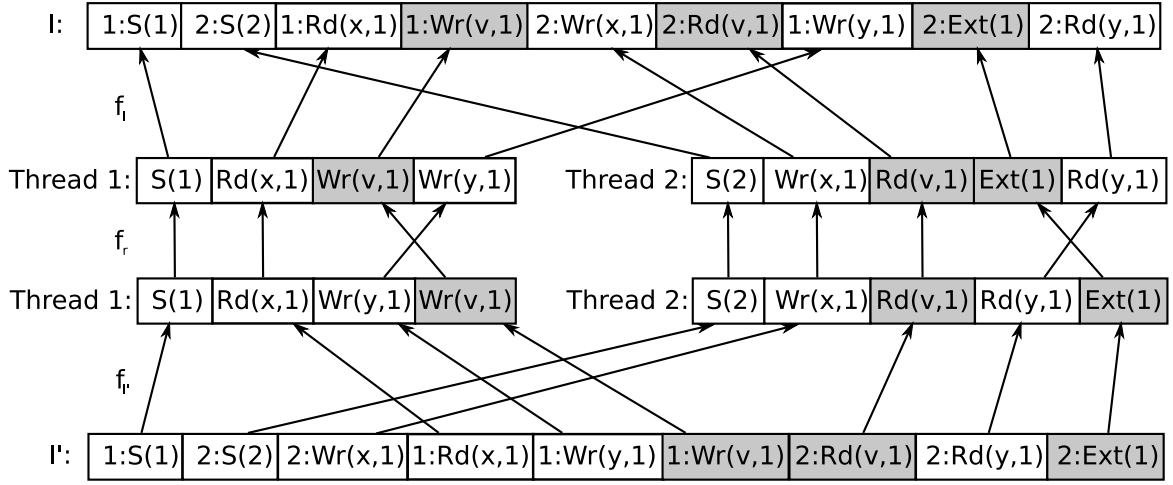


Figure 3.5: Unordering construction.

Thread 1: `r1 := x; v := 1; y := 1;`

Thread 2: `x := 1; r2 := v; print r2; r3 := y;`

and let T be the traceset \hat{T} with the statements `v:=1` and `r3:=y` eliminated, i.e.,

$$T = \hat{T} \cup \{[S(1), \text{Rd}(\mathbf{x}, v)] \mid v \in \tau(\mathbf{x})\} \\ \cup \{[S(2), \text{Wr}(\mathbf{x}, 1), \text{Rd}(v, v_v), \text{Rd}(y, v_y)] \mid v_v \in \tau(v) \wedge v_y \in \tau(y)\}$$

Then the traceset T' of the program

Thread 1: `r1 := x; y := 1; v := 1;`

Thread 2: `x := 1; r2 := v; r3 := y; print r2;`

is a reordering of the traceset T . Figure 3.5 shows the construction of unordering f for an interleaving I' of T' , where

$$I' = [\langle 1, S(1) \rangle, \langle 2, S(2) \rangle, \langle 2, \text{Wr}(x, 1) \rangle, \langle 1, \text{Rd}(x, 1) \rangle, \langle 1, \text{Wr}(y, 1) \rangle, \\ \langle 1, \text{Wr}(v, 1) \rangle, \langle 2, \text{Rd}(v, 1) \rangle, \langle 2, \text{Rd}(y, 1) \rangle, \langle 2, \text{Ext}(1) \rangle].$$

Note that the unordered interleaving I of T is not sequentially consistent. This is because the original program has a data race on \mathbf{x} . Our goal is to show that unordering is always sequentially consistent if the original program is data race free.

Lemma 3.14. *Suppose traceset T' is a reordering of traceset T , and I' is an interleaving of T' . Then there is an unordering from I' to T .*

Proof. Since the construction is essentially the same as in the proof of existence of unelimination function (Lemma 3.2), we will not go into technical details.

We construct the function in three steps. The first step is identical to the first step of the construction of unelimination—we construct a bijection $f_{I'}$ between indices to I' , i.e., $\text{dom}(I')$, and pairs of a thread identifier θ and an index to the trace of θ in I' :

$$f_{I'}(i) = \left\langle \mathcal{T}(I'), \chi_{\text{ldom}_{\mathcal{T}(I'_i)}(I')}^{-1}(i) \right\rangle$$

For each thread we are guaranteed to have a reordering function f_θ from T to the trace of θ in I' . We define $f_r(\langle \theta, i \rangle) = \langle \theta, f_\theta(i) \rangle$ for any θ and $i \in \text{dom}_{I'}(\theta)$.

The last step is again the same as in Lemma 3.2: we take f_I to be any function such that for each θ and $i < j \in \text{dom}_{I'}(\theta)$ we have that $f_I(\langle \theta, i \rangle) < f_I(\langle \theta, j \rangle)$ and if $i < j \in \text{dom}(I')$ and $\mathcal{A}(I_i), \mathcal{A}(I_j)$ are synchronisation or external actions, then $f_I(f_r(f_{I'}(i))) < f_I(f_r(f_{I'}(j)))$. The existence of the function is guaranteed by Fact 3.3, topological sort and Lemma 3.11.

It is straightforward to show that the function $f = f_I \circ f_r \circ f_{I'}$ is an unordering function. The conditions 1, 2 and 3 of Definition 3.14 are satisfied by construction. Using Corollary 3.12, we have

$$f \downarrow_{\text{dom}_\theta(i')} \circ \chi_{\text{ldom}_\theta(I')} = f_\theta.$$

This establishes the condition 4 of the definition of unordering. \square

Definition 3.15. Let t be a list, $n \leq |t|$ and f be a permutation on $\text{dom}(t)$. The *prefix image* of t of length n , denoted by $f_{<n}^\rightarrow(t)$, is the list

$$f_{<n}^\rightarrow(t) = [t_{f^{-1}(i)} \mid i \leftarrow \text{ldom}(t). f^{-1}(i) < n].$$

The *image* of t , written $f^\rightarrow(t)$, is the prefix image of t of length $|t|$. In other words,

$$f^\rightarrow(t) =_{\text{def}} [t_{f^{-1}(i)} \mid i \leftarrow \text{ldom}(t)].$$

Our main goal is to prove that an unordering of an execution of a reordered program is an execution of the original program assuming that the original program was data race free. Since we will prove this by induction on the length of interleavings, we need to capture useful properties of an unordering restricted to a prefix of an interleaving. The following lemmata show that an unordering restricted to an interleaving without the last element is still an unordering.

First, we establish that permutation restriction of a reordering function is a reordering function:

Lemma 3.15. *Suppose that f is a reordering function for t and $n \leq |t|$. Then $f \downarrow_{\{0, \dots, n-1\}}$ is a reordering function for $t \downarrow_{\{0, \dots, n-1\}}$.*

Proof. Let $t' = t \downarrow_{\{0, \dots, n-1\}}$ and $f' = f \downarrow_{\{0, \dots, n-1\}}$. Assume that $i < j < n$ and $f'(j) < f'(i)$. By definition, f and f' agree on order $<$, hence $f(j) < f(i)$, so t_j is reorderable with t_i . As a result, t'_j is reorderable with t'_i , because $t_i = t'_i$ and $t_j = t'_j$. \square

To prove the condition for prefixes in Definition 3.6, we establish that $f \downarrow_{\{0, \dots, n-1\}}$ is a matching between t and $f \overrightarrow{<}_n(t)$:

Lemma 3.16. *Assume that t is a trace, $f : \text{dom}(t) \rightarrow \text{dom}(t)$ is a permutation and $n \leq |t|$. Then*

$$f \overrightarrow{<}_n(t) = \left[t \downarrow_{\{0, \dots, n-1\}}^{-1(i)} \mid i \leftarrow [0, \dots, n-1] \right]. \quad (3.8)$$

Proof. Let $s = [i \leftarrow \text{dom}(t). f^{-1}(i) < n]$. Then we have

$$f \overrightarrow{<}_n(t) = [t_{f^{-1}(i)} \mid i \leftarrow s]. \quad (3.9)$$

On the other hand, from Lemma 3.13 we obtain

$$\left[t \downarrow_{\{0, \dots, n-1\}}^{-1(i)} \mid i \leftarrow [0, \dots, n-1] \right] = \left[t_{(\chi_s^{-1} \circ f)^{-1}(i)} \mid i \leftarrow [0, \dots, n-1] \right] \quad (3.10)$$

By the definition of index function χ , we have $\chi_s(i) = s_i$. Thus,

$$[t_{f^{-1}(s_i)} \mid i \leftarrow [0, \dots, n-1]] = [t_{f^{-1}(i)} \mid i \leftarrow s] \quad (3.11)$$

Combining 3.9 with 3.10 and 3.11 yields 3.8. \square

Lemma 3.17. *Suppose that t is a trace, $|t| > 0$ and f is a function that reorders T to t . Then $f \downarrow_{\{0, \dots, |t|-2\}}$ reorders T to $[t_i \mid i \leftarrow [0, \dots, |t|-2]]$.*

Proof. Let $f' = f \downarrow_{\{0, \dots, |t|-2\}}$ and $t' = [t_i \mid i \leftarrow [0, \dots, |t|-2]]$. By Lemma 3.15, f' is a reordering function.

It remains to show that for all $n \leq |t'| - 1$ we have $f' \overrightarrow{<}_n(t') \in T$. From Lemma 3.16, we have

$$f' \overrightarrow{<}_n(t') = \left[t' \downarrow_{\{0, \dots, n-1\}}^{-1(i)} \mid i \leftarrow [0, \dots, n-1] \right]$$

Since $t'_i = t_i$ for all $i \in \text{dom}(t')$, we obtain

$$f' \overrightarrow{<}_n(t') = \left[t \downarrow_{\{0, \dots, n-1\}}^{-1(i)} \mid i \leftarrow [0, \dots, n-1] \right] \quad (3.12)$$

Moreover, notice that

$$f \downarrow_{\{0, \dots, n-1\}} = (f \downarrow_{\{0, \dots, |t|-2\}}) \downarrow_{\{0, \dots, n-1\}} = f' \downarrow_{\{0, \dots, n-1\}} \quad (3.13)$$

directly by the definition of permutation restriction. Since f reorders T into t , we also know that $f_{<n}^{\rightarrow}(t) \in T$. Together with 3.12, 3.13 and Lemma 3.16, we conclude that

$$f'^{\rightarrow}_{<n}(t') = f_{<n}^{\rightarrow}(t) \in T$$

□

Lemma 3.18. *Let f be an unordering function from a traceset T to an execution $I' \# [a]$. Then the function $f \downarrow_{\{0, \dots, |I'|-1\}}$ is an unordering from T to I' .*

Proof. Since the permutation restriction agrees on order with the original permutation, the properties 1, 2 and 3 of Definition 3.14 are trivially satisfied.

It is not hard to establish the last property using Lemma 3.17: Let θ be a thread and let $f' = f \downarrow_{\{0, \dots, |I'|-1\}}$. If $\theta \neq \mathcal{T}(a)$, then

$$\begin{aligned} f' \downarrow_{\text{dom}_{\theta}(I')} &= f \downarrow_{\text{dom}_{\theta}(I' \# [a])} \\ \chi_{\text{dom}_{\theta}(I')} &= \chi_{\text{dom}_{\theta}(I' \# [a])}, \end{aligned}$$

so $f' \downarrow_{\text{dom}_{\theta}(I' \# [a])} \circ \chi_{\text{dom}_{\theta}(I' \# [a])}$ reorders T into the trace of θ in I' . On the other hand, assume that $\theta = \mathcal{T}(a)$. Let t^{θ} be the trace of θ in $I' \# [a]$, and f^{θ} be the unordering function for the trace of θ in $I' \# [a]$, i.e.,

$$f^{\theta} = f \downarrow_{\text{dom}_{\theta}(I' \# [a])} \circ \chi_{\text{dom}_{\theta}(I' \# [a])}.$$

Note that the trace of θ in I' is t^{θ} with the last element removed, and the unordering function for the trace of θ in I' is $f^{\theta} \downarrow_{\{0, \dots, |t^{\theta}|-2\}}$ (using Corollary 3.12). By Lemma 3.17, $f' \downarrow_{\text{dom}_{\theta}(I')} \circ \chi_{\text{dom}_{\theta}(I')}$ reorders T into the trace of θ in I' . □

In the next lemma, we observe that proper synchronisation only depends on the sequence of synchronisation actions. This will be useful for establishing proper locking of the unordered interleaving.

Lemma 3.19. *Let I and I' be interleavings with the same synchronisation sequence, i.e., for*

$$\text{Sync}(I) =_{\text{def}} [I_i \mid i \leftarrow I. \mathcal{A}(I_i) \text{ is synchronisation}]$$

we have that $\text{Sync}(I) = \text{Sync}(I')$. Then I' is properly locked if and only if I is properly locked.

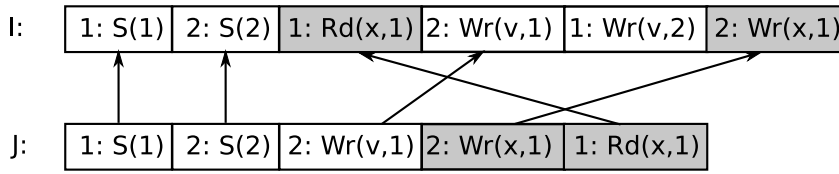


Figure 3.6: Reducing an almost sequentially consistent interleaving to an execution.

Proof. By induction on the size of I , for all I' with the same synchronisation sequence, the number of locks and unlocks in each thread in I is the same as in I' and I is properly locked if only if I' is properly locked. \square

The following lemma relates the happens-before order of an interleaving and its unordered interleaving.

Lemma 3.20. *Suppose that f is an unordering function from an execution I' to a traceset T , and I is $f^{-1}(I')$. Let us have indices $i \leq_{hb}^I j$ such that $\mathcal{A}(I_i)$ and $\mathcal{A}(I_j)$ are memory accesses to the same location. Then $f^{-1}(i) \leq_{hb}^{I'} f^{-1}(j)$.*

Proof. The property (2) of Definition 3.14 guarantees that $k \leq_{po}^I l$ implies $f^{-1}(k) \leq_{po}^{I'} f^{-1}(l)$ if I_k and I_l are conflicting memory accesses or I_k is an acquire or I_l is a release.

Using this observation we prove by induction on the transitive definition of happens-before that $k \leq_{hb}^I j$ implies that either $k \leq_{po}^I j$ or there is $r \in \text{dom}(I)$ such that $\mathcal{A}(I_r)$ is a release action, $k \leq_{po}^I r$ and $f^{-1}(r) \leq_{hb}^{I'} f^{-1}(j)$. Letting $k = i$ and using the observation from the previous paragraph yields the thesis. \square

Lemma 3.21. *Let f be an increasing complete matching between interleavings I' and I . Then for all $i, j \in \text{dom}(I')$, $i \leq_{hb}^{I'} j$ implies $f(i) \leq_{hb}^I f(j)$.*

Proof. By induction on the definition of happens-before. \square

To show sequential consistency of an unordered interleaving by contradiction, we need another procedure that turns an almost sequentially consistent interleaving with a data race into an execution with a data race. The following variant of Lemmata 3.6 and 2.13 constructs such an execution. For an example of the construction, see Figure 3.6.

Lemma 3.22. *Suppose that I is a properly locked interleaving of traceset T and $r < w$ are indices from $\text{dom}(I)$ such that:*

1. r is a read in I and w is a write to the same non-volatile location with the same value in I ,
2. r and w are not ordered by happens-before in I ,
3. each $i \in \{0, \dots, w\} \setminus \{r\}$ sees the most recent write in I and if i sees write j then $j \leq_{hb}^I i$.

Then the interleaving J defined as

$$l = [i \leftarrow \text{dom}(I). i < r \vee i <_{hb}^I w] \# [w, r]$$

$$J = [I_i \mid i \leftarrow l]$$

is an execution of T .

Proof. Same as the proof of Lemma 2.13. □

Finally, we prove the fundamental lemma:

Lemma 3.23. *Let f be an unordering function from an execution I' to a data race free traceset T . Then the interleaving $I = f^{-1}(I')$ is an execution of T .*

Proof. To show that I is an execution we must establish that I is an interleaving of T , I is properly locked, and I is sequentially consistent. The first two properties follow from the definition of unordering: the first one trivially, the second one follows from f preserving synchronisation order and Lemma 3.19.

We prove sequential consistency by induction on the length of I' . If I' is empty, then I is also empty and thus sequentially consistent.

Assume that $I' = J' \# [\langle \theta, a \rangle]$. By Lemma 3.18 and the induction hypothesis, the image J of J' over the function $f \downarrow_{\text{dom}(J')}$ must be an execution. Note that I is J with the last element of I' inserted at position $f(|I'| - 1)$, because by Lemma 3.16,

$$J = f \downarrow_{\text{dom}(J')}^{-1}(J') = f_{<|I'|-1}^{-1}(I') = [I_i \mid i \leftarrow \text{l dom}(I). i \neq f(|I'| - 1)].$$

Hence, if a is not a read or a write, I must be sequentially consistent, because all reads in I see the same (matching) write as the matching read in J . More precisely, there is an increasing complete matching h between J and I , such that $\text{rng}(h) = I \setminus \{f(|I'| - 1)\}$ (Lemma 2.4) and for each $r \in \text{dom}(I)$ such that $\mathcal{A}(I_r)$

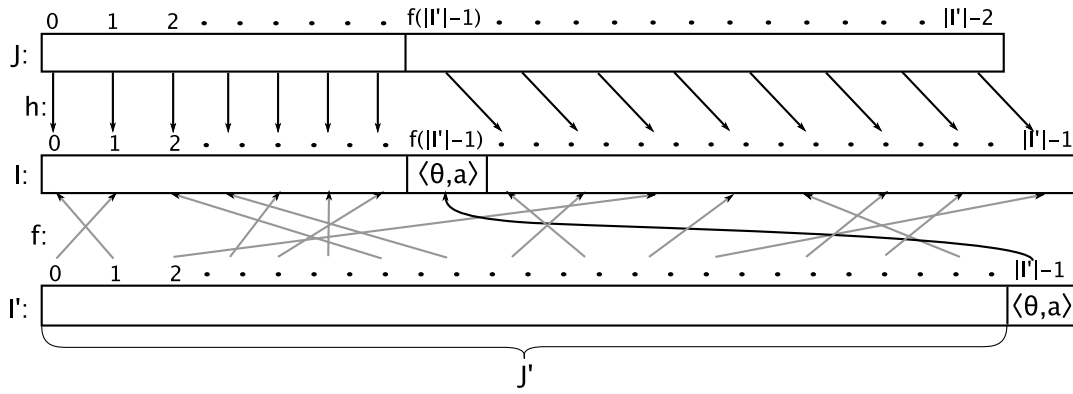


Figure 3.7: Unordering executions by induction.

is a read, if $h^{-1}(r)$ sees the default value in J then r sees the default value in I , and if $h^{-1}(r)$ sees write w in J , then r sees write $h(w)$ in I . See Figure 3.7 for an illustration.

We analyse the remaining three cases for a :

1. If a is a volatile read or write, then the corresponding inserted $f(|I'| - 1)$ -th element of I must be the last synchronisation action of I , because f preserves order of synchronisation actions. If a is a write, then there is no later read from the same location in I and all reads see the same write as the matching read in J . Hence, I is sequentially consistent. If a is a read, then $f(|I'| - 1)$ sees the matching of the most recent write for $|I'| - 1$ in I' , because all accesses to the same variable are synchronisation actions and f preserves the sequence of synchronisation actions.
2. Action a is a normal read. We only need to show that the inserted read at index $f(|I'| - 1)$ of I sees the most recent write, because all other reads in I see the most recent write. We analyse three possible cases:
 - (a) $|I'| - 1$ sees the default value in I' . Then $f(|I'| - 1)$ sees the default value in I : suppose it does not, i.e., suppose that there is a write w' to the same location such that $w' < f(|I'| - 1)$. Observe that $f^{-1}(w') < |I'| - 1$, because $|I'| - 1$ is the highest element of $\text{dom}(f)$. This is a contradiction with $|I'| - 1$ seeing the default value in I' .
 - (b) $|I'| - 1$ sees some index w in I' and $f(w) < f(|I'| - 1)$. Similarly to the previous case, we show that $f(|I'| - 1)$ sees $f(w)$ in I by contradiction: if there is a write w' to the same location such that $f(w) < w' <$

$f(|I'|-1)$, then $f(w) <_{hb}^I w'$ by data race freedom of T , so $w < f^{-1}(w')$ by Lemma 3.20. Since $|I'|-1$ is the highest element of $\text{dom}(f)$, we have $w < f^{-1}(w') < |I'|-1$. This contradicts $|I'|-1$ seeing w .

- (c) $|I'|-1$ sees some index w in I' and $f(w) > f(|I'|-1)$. Observe that $f(|I'|-1)$ cannot happen-before $f(w)$, because that would imply $w > |I'|-1$ (Lemma 3.20). Using Lemma 3.22, we get an execution of T with a data race between its last two elements. This contradicts data race freedom of T .

3. Action a is a normal write. We prove sequential consistency by contradiction. Let r be the index of the first read in I that does not see the most recent write. The most recent write to the same variable in I must be the inserted write at index $f(|I'|-1)$. Otherwise, r would see the same write as $h^{-1}(r)$ in J , which has the same value as I_r by the induction hypothesis. There are two cases:

- (a) There is a write to the same location as a before $f(|I'|-1)$. Let $w < f(|I'|-1)$ be the highest index of such write. From sequential consistency of J , the action $\mathcal{A}(I_w)$ must have the same value as a . By Lemma 3.21 and data race freedom of J , $w \leq_{hb}^I r$. Observe that $f(|I'|-1)$ cannot happen-before r , because that would imply that $f^{-1}(r)$ is greater than $|I'|-1$ (Lemma 3.20), which is the highest index in $\text{dom}(f)$. Applying Lemma 3.6 yields an execution of T with a data race between its last two elements. This is a contradiction with data race freedom of T .
- (b) There is no write to the same location as a before $f(|I'|-1)$. Using the same reasoning as in the previous point, we obtain an execution of T with a data race from Lemma 3.6.

□

The DRF guarantee is the direct consequence of the Lemmata 3.14 and 3.23:

Theorem 3.3. *Suppose that traceset T' is a reordering of a data race free traceset T . Then any execution of T' has the same behaviour as some execution of T .*

Lemma 3.23 also implies preservation of data race freedom by reordering.

Theorem 3.4. *Let traceset T' be a reordering of data race free traceset T . Then T' is also data race free.*

Proof. Suppose that I' is an execution of T' . By Lemmata 3.14 and 3.23, there is an unordering f from I' to T , and $I = f^\rightarrow(I')$ is an execution of T . Since T is data race free, all conflicting accesses in I are ordered by happens-before. Since the inverse of unordering preserves the happens-before order on actions of each location (Lemma 3.20), there cannot be any data races in I' . \square

3.2.3 Out-of-thin-air

In this subsection we prove that our transformations provide a simple out of thin air guarantee. We will establish that for each output action of some value from an execution of the transformed program there is a statement in the original program that must have created that value. In a language without arithmetic, this might mean that if a program outputs value v , then v must be a default value, or the program must contain the value v explicitly. In a language with dynamic object allocation, we might use this technique to show that if a program cannot allocate objects of a certain class, then in no transformed program does a reference to such an object appear out-of-thin-air.

The guarantee is based on a simple observation: Let v be a value that is different from the default values of all locations. If a program without arithmetic does not contain v , then in each trace, each write of the value v and each external action with the value v must be preceded by a read of the value v . In the following, we will prove that our semantic transformations preserve this property. Later, in Section 4.4, we show an application of this semantic property to a simple language.

Definition 3.16. Let t be a trace. We say that t is an *origin* for value v if there is $i \in \text{dom}(t)$ such that t_i is a write of the value v or an external action with the value v , and there is no $j < i$ such that t_j is a read of the value v .

Lemma 3.24. *Let traceset T' be a reordering of traceset T and suppose that no trace in T is an origin for v . Let us assume that no location has a singleton type with value v . Then no trace in T' is an origin for v .*

Proof. By contradiction. Suppose there is $t' \in T'$ and o such that $t'_o = \text{Ext}(v)$ or $t'_o = \text{Wr}(l, v)$ for some l , and there is no preceding read of the value v . Let \hat{t} be

the prefix of t of length $o + 1$, i.e., $\hat{t} = [t_i \mid i \leftarrow [0, \dots, o]]$. Observe that \hat{t} does not contain any read with the value v . By Definitions 3.6 and 3.7, there must be a reordering function f for \hat{t} such that $f^\rightarrow(\hat{t}) \in T$. The trace $f^\rightarrow(\hat{t})$ contains a write or an external action with value v at position $f(o)$, but it does not contain any read of value v , so it is an origin for v . This is a contradiction. \square

Lemma 3.25. *Let traceset T' be an elimination of traceset T and suppose that no trace in T is an origin for v . Let us assume that no location has a singleton type with value v . Then no trace in T' is an origin for v .*

Proof. By contradiction. Assume that we have $t' \in T'$ and o such that $t'_o = \text{Ext}(v)$ or $t'_o = \text{Wr}(l, v)$ for some l , and there is no preceding read of the value v . Let o be the lowest such o ; hence, there is no $i < o$ such that i is a read, write, or external action with the value v . Let \hat{t} be the prefix of t of length $o + 1$. There must be a wildcard trace t^* from T such that \hat{t} is an elimination of t^* . Let t be an instance of t^* such that each wildcard read in t^* is replaced by a read of some value that is different from v . Since t^* is in T , t must also be in T and t cannot be an origin for v . As t contains a write or an external action with value v , there must be an earlier read of v in t . Let us take first such read. The read cannot be eliminable in t , because there is no earlier read or write with the same value. Therefore, the read must be also in \hat{t} . This contradicts our assumption. \square

Finally, we show that if T does not contain an origin for a value, then the value cannot appear in any execution of T :

Lemma 3.26. *Suppose that v is a value, that is not a default value for any type, T is a traceset, and no t in T is an origin for v . Then there is no execution of T that contains a read, write or external action with the value v .*

Proof. By contradiction. Let I be an execution of T that contains a read, a write of an external action with value v . Let i be the lowest index of such action. Observe that i cannot be a read in I , because by sequential consistency of I , it would have to see the earlier write of value v . As a result, the trace of thread $\mathcal{T}(I_i)$ is an origin for v . \square

3.3 High-level Remarks about Roach Motel Reordering

The reader might find the form of some of the definitions and the proofs of safety overly technical. It might be useful to note that much of this complexity comes from the roach motel reordering transformation.

Perhaps surprisingly, the roach motel reordering complicates the proof of safety of elimination, because we must be able to eliminate a release that is reordered with a later memory access. If it was not for the elimination of last release we could construct the unelimination simply by inserting back the eliminated actions instead of the construction in Lemma 3.2. However, the elimination of last release breaks sequential consistency in this construction by insertion, because we might need to insert a volatile write before a volatile read from the same location with a different value. This is illustrated in Example 3.7.

The roach motel reordering also complicates the definition of reordering (Definition 3.6). In its absence, we could omit the condition (3.1) for reordering prefixes of traces from the definition and still get the safety result. The proof of safety could be also simplified—we could use the same argument as in the proof of safety of elimination, because without the roach motel reordering the unordering function preserves happens-before order on each variable in both directions (cf. Lemma 3.20 and Lemma 3.8).

Chapter 4

Syntax to Semantics

In our examples from the previous chapters we have been using a C-like language that contains assignments, loop and conditional control flow statements, and a statement for output, but we have not specified its precise semantics. In this chapter, we fill this gap. We give a formal syntax of the language in Section 4.1, together with a precise trace semantic meaning for the syntactic elements. In Section 4.2, we formalise several peephole transformations that eliminate redundant reads and writes. We establish that these elimination transformations guarantee safety for data race free programs and preserve data race freedom. In Section 4.3, we prove the same guarantees for reordering. Finally, in Section 4.4, we show a simple out-of-thin-air guarantee for our language—if a program does not contain an assignment of a constant c to a register, then none of its transformations can output c .

4.1 Language

Our language operates on memory $\langle L, M, T_\theta, L_v, V, \tau, D \rangle$, where

- the set of locations L contains at least location names \mathbf{x} , \mathbf{y} , \mathbf{z} , \mathbf{u} and \mathbf{v} , which must be distinct,
- the set of monitors M contains at least two distinct monitor names $\mathbf{m1}$ and $\mathbf{m2}$,
- the set of thread identifiers T_θ is the set of natural numbers,
- the set of volatile locations $L_v \subset L$ contains at least \mathbf{u} and \mathbf{v} , but does not contain \mathbf{x} , \mathbf{y} and \mathbf{z} .

- the set of values V is the set of natural numbers,
- the type-of function τ assigns the set of natural numbers to any location from L ,
- the default value function D assigns 0 to any location from L .

Moreover, we use a set of register names R , ranged over by r , that contains at least three register names **r1**, **r2** and **r3**. We assume that L , R and V are disjoint. In the syntax of our toy language, l ranges over the locations from L , m over the monitors from M . The monitors are Java-style re-entrant. The values are ranged over by i . We define the syntax of the language in Figure 4.1.

$$\begin{aligned}
 ri &::= r \mid i \\
 T &::= ri == ri \mid ri != ri \\
 C &::= l := r \mid r := l \mid r := ri \mid \text{lock } m \mid \text{unlock } m \mid \text{print } r \\
 L &::= \emptyset \mid S L \\
 S &::= C; \mid \{L\} \mid \text{if } (T) S \text{ else } S \mid \text{while } (T) S \\
 P &::= S \parallel S \parallel \dots \parallel S
 \end{aligned}$$

Figure 4.1: Syntax of our toy language.

Our trace semantics uses judgements of the form

$$\begin{aligned}
 \Lambda, \sigma, K \Downarrow \Lambda', \sigma', t \\
 \Lambda, \sigma, K \Downarrow_{\perp} t
 \end{aligned}$$

where σ and σ' stand for thread local states, i.e., mappings from register names R to values V , Λ and Λ' are states of monitors, which are functions from monitor names M to natural numbers representing the nesting level of each monitor, K is a code fragment, i.e., either C , L , S or P , and t is a trace. A local state σ is a function from register names to natural numbers. State updates will be denoted by $\sigma[r \mapsto i]$: the value of $\sigma[r \mapsto i](r')$ is i if $r = r'$, or $\sigma(r')$ otherwise. The judgement $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$ means that the code fragment K evaluated in state Λ, σ can generate trace t while finishing in state Λ', σ' . The judgement $\Lambda, \sigma, K \Downarrow_{\perp} t$ says that K can produce trace t if started in state Λ, σ . We

define the semantics for these judgements in Figures 4.2, 4.3, 4.4 and 4.5. Note that we use the latter judgement to approximate non-terminating executions of a program. For example, for any Λ and σ there is no Λ' , σ' and t such that $\Lambda, \sigma, \text{while } (0==0) \ x:=1; \Downarrow \Lambda', \sigma', t$. However, for any n , Λ and σ we have $\Lambda, \sigma, \text{while } (0==0) \ x:=1; \Downarrow_{\perp} [\text{Wr}(x, 1) \mid i \leftarrow [0, \dots, n]]$. In fact, there is a simple relationship between the judgements:

Lemma 4.1. *If $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$ and t' is a strict prefix of t then $\Lambda, \sigma, K \Downarrow_{\perp} t'$. Similarly, $t' \leq t$ and $\Lambda, \sigma, K \Downarrow_{\perp} t$ implies $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$.*

Proof. By induction on the length of derivation. □

Definition 4.1. Given a program P , the *meaning of the program P* , denoted by $\llbracket P \rrbracket$, is the set of all traces t such that either

- $\Lambda_{\text{init}}, \sigma_{\text{init}}, P \Downarrow_{\perp} t$, or
- there are Λ' and σ' such that $\Lambda_{\text{init}}, \sigma_{\text{init}}, P \Downarrow \Lambda', \sigma', t$,

where σ_{init} is an environment that maps every register name to 0 and Λ_{init} maps every monitor name to 0.

Lemma 4.2. *For any program P , the set $\llbracket P \rrbracket$ is a traceset.*

Proof. Lemma 4.1 establishes prefix closedness. To see that $\llbracket P \rrbracket$ satisfies proper locking, observe that $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$ implies that for each monitor m the difference between the number of $L(m)$ in t and the number of $U(m)$ actions in t equals $\Lambda'(m) - \Lambda(m)$ (by induction on the derivation of $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$). Using this, we prove by induction that $\Lambda, \sigma, K \Downarrow_{\perp} t$ implies that

$$\forall m. |\{i \mid i \in \text{dom}(t) \wedge t_i = U(m)\}| - |\{i \mid i \in \text{dom}(t) \wedge t_i = L(m)\}| \leq \Lambda(m)$$

Since $\Lambda_{\text{init}}(m)$ is defined to be 0, the set $\llbracket P \rrbracket$ is properly locked.

Proper starting of $\llbracket P \rrbracket$ follows from the semantics of the parallel composition (Figure 4.5). Proper typing is guaranteed, because the range of the local state functions is natural numbers. □

$$\begin{array}{c}
\overline{\text{Val}(\sigma, i) = i} \\
\frac{\text{Val}(\sigma, ri_1) = \text{Val}(\sigma, ri_2)}{\text{Val}(\sigma, ri_1 == ri_2) = \mathbf{tt}} \\
\frac{\text{Val}(\sigma, ri_1) \neq \text{Val}(\sigma, ri_2)}{\text{Val}(\sigma, ri_1 != ri_2) = \mathbf{tt}}
\end{array}
\qquad
\begin{array}{c}
\overline{\text{Val}(\sigma, r) = \sigma(r)} \\
\frac{\text{Val}(\sigma, ri_1) \neq \text{Val}(\sigma, ri_2)}{\text{Val}(\sigma, ri_1 == ri_2) = \mathbf{ff}} \\
\frac{\text{Val}(\sigma, ri_1) = \text{Val}(\sigma, ri_2)}{\text{Val}(\sigma, ri_1 != ri_2) = \mathbf{ff}}
\end{array}$$

Figure 4.2: Values of variables, constants and tests.

$$\begin{array}{c}
\frac{v \in \mathbb{N}}{\Lambda, \sigma, r := l \Downarrow \Lambda, \sigma[r \mapsto v], [\text{Rd}(l, v)]} \\
\frac{}{\Lambda, \sigma, r := i \Downarrow \Lambda, \sigma[r \mapsto i], []} \\
\frac{}{\Lambda, \sigma, \text{lock } m \Downarrow \Lambda[m \mapsto \Lambda(m) + 1], \sigma, [\text{L}(m)]} \\
\frac{\Lambda(m) > 0}{\Lambda, \sigma, \text{unlock } m \Downarrow \Lambda[m \mapsto \Lambda(m) - 1], \sigma, [\text{U}(m)]} \\
\frac{}{\Lambda, \sigma, \text{print } r \Downarrow \Lambda, \sigma, [\text{Ext}(\sigma(r))]}
\end{array}
\qquad
\begin{array}{c}
\frac{}{\Lambda, \sigma, l := r \Downarrow \Lambda, \sigma, [\text{Wr}(l, \sigma(r))]} \\
\frac{}{\Lambda, \sigma, r_1 := r_2 \Downarrow \Lambda, \sigma[r_1 \mapsto \sigma(r_2)], []} \\
\frac{\Lambda(m) = 0}{\Lambda, \sigma, \text{unlock } m \Downarrow \Lambda, \sigma, []}
\end{array}$$

Figure 4.3: Semantics of simple statements.

$$\begin{array}{c}
\frac{}{\Lambda, \sigma, \emptyset \Downarrow \Lambda, \sigma, \square} \text{(SEQ-EMPTY)} \qquad \frac{\Lambda, \sigma, S \Downarrow_{\perp} t}{\Lambda, \sigma, S L \Downarrow_{\perp} t} \text{(SEQ-1)} \\
\\
\frac{\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \quad \Lambda', \sigma', L \Downarrow_{\perp} t'}{\Lambda, \sigma, S L \Downarrow_{\perp} t \# t'} \text{(SEQ-2)} \\
\\
\frac{\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \quad \Lambda', \sigma', L \Downarrow \Lambda'', \sigma'', t'}{\Lambda, \sigma, S L \Downarrow \Lambda', \sigma'', t \# t'} \text{(SEQ)} \\
\\
\frac{\Lambda, \sigma, C \Downarrow \Lambda', \sigma', t}{\Lambda, \sigma, C; \Downarrow \Lambda', \sigma', t} \qquad \frac{}{\Lambda, \sigma, C; \Downarrow_{\perp} \square} \\
\\
\frac{\Lambda, \sigma, L \Downarrow \Lambda', \sigma', t}{\Lambda, \sigma, \{L\} \Downarrow \Lambda', \sigma', t} \qquad \frac{\Lambda, \sigma, L \Downarrow_{\perp} t}{\Lambda, \sigma, \{L\} \Downarrow_{\perp} t} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{tt} \quad \Lambda, \sigma, S_1 \Downarrow \Lambda', \sigma', t}{\Lambda, \sigma, \text{if } (T) S_1 \text{ else } S_2 \Downarrow \Lambda', \sigma', t} \qquad \frac{\text{Val}(\sigma, T) = \mathbf{tt} \quad \Lambda, \sigma, S_1 \Downarrow_{\perp} t}{\Lambda, \sigma, \text{if } (T) S_1 \text{ else } S_2 \Downarrow_{\perp} t} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{ff} \quad \Lambda, \sigma, S_2 \Downarrow \Lambda', \sigma', t}{\Lambda, \sigma, \text{if } (T) S_1 \text{ else } S_2 \Downarrow \Lambda', \sigma', t} \qquad \frac{\text{Val}(\sigma, T) = \mathbf{ff} \quad \Lambda, \sigma, S_2 \Downarrow_{\perp} t}{\Lambda, \sigma, \text{if } (T) S_1 \text{ else } S_2 \Downarrow_{\perp} t} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{tt} \quad \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \quad \Lambda', \sigma', \text{while } (T) S \Downarrow \Lambda'', \sigma'', t'}{\Lambda, \sigma, \text{while } (T) S \Downarrow \Lambda'', \sigma'', t \# t'} \text{(LOOP-T)} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{ff}}{\Lambda, \sigma, \text{while } (T) S \Downarrow \Lambda, \sigma, \square} \text{(LOOP-F)} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{tt} \quad \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \quad \Lambda', \sigma', \text{while } (T) S \Downarrow_{\perp} t'}{\Lambda, \sigma, \text{while } (T) S \Downarrow_{\perp} t \# t'} \text{(LOOP-T1)} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{tt} \quad \Lambda, \sigma, S \Downarrow_{\perp} t}{\Lambda, \sigma, \text{while } (T) S \Downarrow_{\perp} t} \text{(LOOP-T2)} \\
\\
\frac{\text{Val}(\sigma, T) = \mathbf{ff}}{\Lambda, \sigma, \text{while } (T) S \Downarrow_{\perp} \square} \text{(LOOP-F1)}
\end{array}$$

Figure 4.4: Semantics of control flow

$$\begin{array}{c}
\frac{\Lambda, \sigma, S_i \Downarrow \Lambda', \sigma', t}{\Lambda, \sigma, S_0 \parallel \dots \parallel S_i \parallel \dots \parallel S_n \Downarrow \Lambda', \sigma', [S(i)] \# t} \\
\frac{\Lambda, \sigma, S_i \Downarrow_{\perp} t}{\Lambda, \sigma, S_0 \parallel \dots \parallel S_i \parallel \dots \parallel S_n \Downarrow_{\perp} [S(i)] \# t} \\
\hline
\Lambda, \sigma, S_0 \parallel \dots \parallel S_i \parallel \dots \parallel S_n \Downarrow_{\perp} \square
\end{array}$$

Figure 4.5: Semantics of parallel composition.

4.2 Elimination

In Figure 4.6, we define a syntactic transformation that allows the following eliminations of adjacent statements:

- elimination of a read preceded by a read from the same location (rule E-RAR),
- elimination of a read that follows a write to the same location (rule E-RAW),
- elimination of a write preceded by a read with the same value from the same location (E-WAR),
- elimination of a write that is overwritten by a subsequent write to the same location (E-WBW),
- elimination of a read, which is not used, because the subsequent operation destroys the read value (E-IR).

There is a small problem with this definition: it only allows eliminations at the end of a basic block, because the statement composition is right-associative (see rule SEQ in Figure 4.4). For instance, the judgement

$$r1:=x;r2:=x;print\ r2; \xrightarrow{e} r1:=x;r2:=r1;print\ r2;$$

cannot be derived from the rules in Figure 4.6. However, it is easy to prove by induction that replacing any code fragment of the form $S_1\ S_2\ S_3$ with $\{S_1\ S_2\}\ S_3$ (and vice versa) preserves the meaning in any context. This implies that we can always safely lift the pair of instructions to be eliminated to a basic block and

then perform the elimination. For example, starting with the program

```
Thread 0:  r1:=x;r2:=x;print r2;
```

we can rewrite $r1:=x;r2:=x;$ to $\{r1:=x;r2:=x;\}$ without changing the meaning of the program, and then eliminate:

```
Thread 0:\{r1:=x;r2:=x;\}print r2;  $\xrightarrow{e}$  Thread 0:\{r1:=x;r2:=r1;\}print r2;
```

Finally, we can remove the basic block

```
Thread 0:  r1:=x;r2:=r1;print r2;
```

If the elimination step is safe, then the entire transformation must be safe, because the first and last steps do not change the meaning of the program. In the rest of this section we establish that the elimination is indeed safe.

We only need to prove that the traceset of the transformed program is an elimination of the original program in the trace-semantic sense (Definition 3.3). Using our trace-semantic results, we will conclude that the syntactic elimination preserves data race freedom and cannot introduce new behaviours for data race free programs.

For reasoning about irrelevant read eliminations, it is useful to extend our judgements to wildcard traces:

Definition 4.2. Given a wildcard trace t^* , monitor states Λ, Λ' , register states σ, σ' and code fragment K , the judgement

$$\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t^*$$

means that for any instance t of t^* the judgement $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$ holds. Similarly,

$$\Lambda, \sigma, K \Downarrow_{\perp} t^*$$

holds if $\Lambda, \sigma, K \Downarrow_{\perp} t$ holds for any instance t of t^* .

Note that the semantic elimination (Definition 3.2) on traces is not composable in the sense that t'_1 being an elimination of t_1 and t'_2 being an elimination of t_2 does not necessarily imply that $t'_1 \# t'_2$ is an elimination of $t_1 \# t_2$. For example, $t'_1 = []$, i.e., the empty trace, is an elimination of $t_1 = [\text{Ext}(0)]$, and $t'_2 = [\text{Ext}(1)]$ is a trivial elimination of $t_2 = [\text{Ext}(1)]$, but $t'_1 \# t'_2 = [\text{Ext}(1)]$ is not an elimination of $t_1 \# t_2 = [\text{Ext}(0), \text{Ext}(1)]$. However, if we restrict our attention to a smaller set of proper eliminations below, we get this property:

$$\begin{array}{c}
\overline{C \xrightarrow{e} C} \qquad \qquad \qquad \overline{L \xrightarrow{e} L} \\
\\
\frac{L \xrightarrow{e} L' \quad S \xrightarrow{e} S'}{S L \xrightarrow{e} S' L'} \text{(E-SEQ)} \qquad \qquad \frac{L \xrightarrow{e} L'}{\{L\} \xrightarrow{e} \{L'\}} \\
\\
\frac{S_1 \xrightarrow{e} S'_1 \quad S_2 \xrightarrow{e} S'_2}{\text{if } (T) \text{ then } S_1 \text{ else } S_2 \xrightarrow{e} \text{if } (T) \text{ then } S'_1 \text{ else } S'_2} \text{(E-IF)} \\
\\
\frac{S \xrightarrow{e} S'}{\text{while } (T) S \xrightarrow{e} \text{while } (T) S'} \text{(E-WHILE)} \\
\\
\frac{\forall i \in \{0, \dots, n\}. S_i \xrightarrow{e} S'_i}{S_0 \parallel \dots \parallel S_n \xrightarrow{e} S'_0 \parallel \dots \parallel S'_n} \\
\\
\frac{x \text{ not volatile}}{r_1 := x; r_2 := x; \xrightarrow{e} r_1 := x; r_2 := r_1;} \text{(E-RRR)} \qquad \frac{x \text{ not volatile}}{x := r_1; r_2 := x; \xrightarrow{e} x := r_1; r_2 := r_1;} \text{(E-RAW)} \\
\\
\frac{x \text{ not volatile}}{r := x; x := r; \xrightarrow{e} r := x;} \text{(E-WAR)} \qquad \frac{x \text{ not volatile}}{x := r_1; x := r_2; \xrightarrow{e} x := r_2;} \text{(E-WBW)} \\
\\
\frac{x \text{ not volatile}}{r := x; r := i; \xrightarrow{e} r := i;} \text{(E-IR)}
\end{array}$$

Figure 4.6: Syntactic elimination.

Definition 4.3. Index i is *properly eliminable* in a wildcard trace t if i is a redundant read after read, or a redundant read after write, or an irrelevant read, or a redundant write after read or an overwritten write.

Given traces t and t' , the trace t' is a *proper elimination* of t if there is S such that $t' = t|_S$ and all $i \in \text{dom}(t) \setminus S$ are properly eliminable in t . We denote the proper elimination by $t \rightsquigarrow_e t'$.

Lemma 4.3. *Suppose that $t_1 \rightsquigarrow_e t'_1$ and $t_2 \rightsquigarrow_e t'_2$. Then $t_1 \# t_2 \rightsquigarrow_e t'_1 \# t'_2$.*

Proof. By Definition 4.3, there are sets S_1 and S_2 such that $t'_1 = t_1|_{S_1}$ and $t'_2 = t_2|_{S_2}$. Let

$$S' = S_1 \cup \{i + |S_1| \mid i \in S_2\}$$

It is straightforward to check that $t'_1 \# t'_2 = (t_1 \# t_2)|_{S'}$ and each $i \in S'$ is eliminable in $t_1 \# t_2$. \square

Lemma 4.4. *Suppose that K and K' are code fragments such that $K \xrightarrow{e} K'$. Then for any monitor states Λ, Λ' , any register states σ, σ' and any trace t' the following two statements hold:*

1. $\Lambda, \sigma, K' \Downarrow \Lambda', \sigma', t'$ implies that there is a wildcard trace t^* such that $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t^*$ and $t^* \rightsquigarrow_e t'$.
2. if $\Lambda, \sigma, K' \Downarrow_{\perp} t'$ then there is a wildcard trace t^* such that $\Lambda, \sigma, K \Downarrow_{\perp} t^*$ and $t^* \rightsquigarrow_e t'$.

Proof. By induction on the derivation of $K \xrightarrow{e} K'$:

1. E-RAR: assume that $\Lambda, \sigma, r_1 := x; r_2 := r_1; \Downarrow \Lambda', \sigma', t'$. By the operational semantics, there is a value v such that $t' = [\text{Rd}(x, v)]$, $\sigma' = \sigma[r_1 \mapsto v][r_2 \mapsto v]$, and $\Lambda' = \Lambda$. Take $t = [\text{Rd}(x, v), \text{Rd}(x, v)]$ and observe that t' is a proper elimination of t , because its second action is a redundant read after read in t , and $\Lambda, \sigma, r_1 := x; r_2 := x; \Downarrow \Lambda', \sigma', t$. Moreover, $\Lambda, \sigma, r_1 := x; r_2 := r_1; \Downarrow_{\perp} t'$ implies that t' is $[\text{Rd}(x, v)]$ or an empty list. In both cases, we have $\Lambda, \sigma, r_1 := x; r_2 := x \Downarrow_{\perp} t'$ and $t' \rightsquigarrow_e t'$.
2. E-RAW: $\Lambda, \sigma, x := r_1; r_2 := r_1; \Downarrow \Lambda', \sigma', t'$ implies $t' = [\text{Wr}(x, \sigma(r_1))]$, $\sigma' = \sigma[r_2 \mapsto \sigma(r_1)]$, $\Lambda' = \Lambda$. Let $t = [\text{Wr}(x, \sigma(r_1)), \text{Rd}(x, \sigma(r_1))]$. Then we have $t \rightsquigarrow_e t'$ and $\Lambda, \sigma, x := r_1; r_2 := x; \Downarrow \Lambda', \sigma', t$. If $\Lambda, \sigma, x := r_1; r_2 := r_1; \Downarrow_{\perp} t'$, then $t' = []$ or $t' = [\text{Wr}(x, \sigma(r_1))]$. In both cases, $\Lambda, \sigma, x := r_1; r_2 := x; \Downarrow_{\perp} t'$.

3. E-WAR and E-WBW are very similar to the cases E-RAR and E-RAW.
4. E-IR differs from the previous cases only in the usage of wildcard traces: If $\Lambda, \sigma, r := i \Downarrow \Lambda', \sigma', t'$, then $t' = []$, $\Lambda' = \Lambda$, and $\sigma' = \sigma[r \mapsto i]$. Take t to be $[\text{Rd}(x, *)]$. Note that $t \rightsquigarrow_e t'$ and $\Lambda, \sigma, r := x; r := i \Downarrow \Lambda', \sigma', t$. If $\Lambda, \sigma, r := i \Downarrow_{\perp} t'$ then $t' = []$, so $\Lambda, \sigma, r := x; r := i \Downarrow_{\perp} t'$.
5. E-SEQ: By the induction hypothesis, for any trace t' , wildcard trace t^* , monitor states Λ, Λ' , and register states σ, σ' we have

$$\Lambda, \sigma, S' \Downarrow \Lambda', \sigma', t' \text{ implies } \exists t^*. \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t^* \wedge t^* \rightsquigarrow_e t' \quad (4.1)$$

$$\Lambda, \sigma, S' \Downarrow_{\perp} t' \text{ implies } \exists t^*. \Lambda, \sigma, S \Downarrow_{\perp} t^* \wedge t^* \rightsquigarrow_e t' \quad (4.2)$$

$$\Lambda, \sigma, L' \Downarrow \Lambda', \sigma', t' \text{ implies } \exists t^*. \Lambda, \sigma, L \Downarrow \Lambda', \sigma', t^* \wedge t^* \rightsquigarrow_e t' \quad (4.3)$$

$$\Lambda, \sigma, L' \Downarrow_{\perp} t' \text{ implies } \exists t^*. \Lambda, \sigma, L \Downarrow_{\perp} t^* \wedge t^* \rightsquigarrow_e t' \quad (4.4)$$

Let us assume that $\Lambda, \sigma, S' L' \Downarrow \Lambda', \sigma', t'$. By the rule SEQ of the operational semantics, we have $t' = t'_1 \# t'_2$, $\Lambda, \sigma, S' \Downarrow \Lambda'', \sigma'', t'_1$ and $\Lambda'', \sigma'', L' \Downarrow \Lambda', \sigma', t'_2$. By 4.1 and 4.3, there are t_1 and t_2 such that $t_1 \rightsquigarrow_e t'_1$ and $t_2 \rightsquigarrow_e t'_2$ and $\Lambda, \sigma, S \Downarrow \Lambda'', \sigma'', t_1$ and $\Lambda'', \sigma'', L \Downarrow \Lambda', \sigma', t_2$. By Lemma 4.3 and the SEQ rule, $\Lambda, \sigma, S L \Downarrow \Lambda', \sigma', t_1 \# t_2$ and $t_1 \# t_2 \rightsquigarrow_e t'$.

Moreover, assume that $\Lambda, \sigma, S' L' \Downarrow_{\perp} t'$. By the operational semantics, there are two cases:

- (a) SEQ-2: $t' = t'_1 \# t'_2$, $\Lambda, \sigma, S' \Downarrow \Lambda', \sigma', t'_1$ and $\Lambda', \sigma', L' \Downarrow_{\perp} t'_2$. From 4.1 and 4.4, we obtain t_1 and t_2 such that $t_1 \rightsquigarrow_e t'_1$ and $t_2 \rightsquigarrow_e t'_2$ and $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t_1$ and $\Lambda', \sigma', L \Downarrow_{\perp} t_2$. By Lemma 4.3 and rule SEQ-2, $\Lambda, \sigma, S L \Downarrow_{\perp} t_1 \# t_2$ and $t_1 \# t_2 \rightsquigarrow_e t'$.

- (b) SEQ-1: Follows from 4.2.

6. E-IF: straightforward.
7. E-WHILE: By induction hypothesis, we have

$$\Lambda, \sigma, S' \Downarrow \Lambda', \sigma', t' \text{ implies } \exists t^*. \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t^* \wedge t^* \rightsquigarrow_e t' \quad (4.5)$$

$$\Lambda, \sigma, S' \Downarrow_{\perp} t' \text{ implies } \exists t^*. \Lambda, \sigma, S \Downarrow_{\perp} t^* \wedge t^* \rightsquigarrow_e t' \quad (4.6)$$

for any trace t' , wildcard trace t^* , monitor states Λ, Λ' , and register states σ, σ' .

Let us assume that $\Lambda, \sigma, \mathbf{while} (T) S' \Downarrow \Lambda', \sigma', t'$. We prove by induction on the derivation of $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t'$ that $K = \mathbf{while} (T) S'$ implies existence of t^* such that $t^* \rightsquigarrow_e t'$ and $\Lambda, \sigma, \mathbf{while} (T) S \Downarrow \Lambda', \sigma', t^*$. There are two relevant cases:

- LOOP-F: $\text{Val}(\sigma, T) = \mathbf{ff}$, $t' = []$, $\sigma' = \sigma$ and $\Lambda' = \Lambda$. Therefore,

$$\Lambda, \sigma, \mathbf{while} (T) S \Downarrow \Lambda', \sigma', t'.$$

- LOOP-T: $\text{Val}(\sigma, T) = \mathbf{tt}$ and $\Lambda, \sigma, S' \Downarrow \Lambda'', \sigma'', t'_1$ and $\Lambda'', \sigma'', \mathbf{while} (T) S' \Downarrow \Lambda', \sigma', t'_2$ and $t' = t'_1 \# t'_2$. From the induction hypothesis (of the derivation on the operational semantics) we obtain t_2 such that $\Lambda'', \sigma'', \mathbf{while} (T) S \Downarrow \Lambda', \sigma', t_2$ and $t_2 \rightsquigarrow_e t'_2$. From 4.5 we get t_1 such that $\Lambda, \sigma, S \Downarrow \Lambda'', \sigma'', t_1$ and $t_1 \rightsquigarrow_e t'_1$. Using Lemma 4.3, we have $t_1 \# t_2 \rightsquigarrow_e t'_1 \# t'_2$, and by the rule LOOP-T of the operational semantics, $\Lambda, \sigma, \mathbf{while} (T) S \Downarrow \Lambda', \sigma', t_1 \# t_2$.

Similarly, assume that $\Lambda, \sigma, \mathbf{while} (T) S' \Downarrow_{\perp} t'$. We show by induction on derivation of $\Lambda, \sigma, K \Downarrow_{\perp} t'$ that $K = \mathbf{while} (T) S'$ implies existence of t^* such that $t^* \rightsquigarrow_e t'$ and $\Lambda, \sigma, \mathbf{while} (T) S \Downarrow_{\perp} t^*$. The three relevant cases are:

- LOOP-F1: trivial.
- LOOP-T1: follows from the operational semantics and 4.6.
- LOOP-T2: Let us assume that $\text{Val}(\sigma, T) = \mathbf{tt}$ and $\Lambda, \sigma, S' \Downarrow \Lambda'', \sigma'', t'_1$ and $\Lambda'', \sigma'', \mathbf{while} (T) S' \Downarrow_{\perp} t'_2$ and also $t' = t'_1 \# t'_2$. From the induction hypothesis we have t_2 such that $\Lambda'', \sigma'', \mathbf{while} (T) S \Downarrow \Lambda', \sigma', t_2$ and $t_2 \rightsquigarrow_e t'_2$, and from 4.6 we get t_1 such that $\Lambda, \sigma, S \Downarrow_{\perp} t_1$ and $t_1 \rightsquigarrow_e t'_1$. Using Lemma 4.3, we have $t_1 \# t_2 \rightsquigarrow_e t'_1 \# t'_2$, and by the operational semantics, $\Lambda, \sigma, \mathbf{while} (T) S \Downarrow_{\perp} t_1 \# t_2$.

All the remaining induction cases are trivial. \square

Theorem 4.1. *Suppose that $P \xrightarrow{e} P'$ and $\llbracket P \rrbracket$ is data race free. Then $\llbracket P' \rrbracket$ is data race free, and any execution of $\llbracket P' \rrbracket$ has the same behaviour as some execution of $\llbracket P \rrbracket$.*

Proof. By Lemma 4.4, $\llbracket P' \rrbracket$ is an elimination of $\llbracket P \rrbracket$. Therefore, $\llbracket P' \rrbracket$ must be data race free (Theorem 3.1) and for any execution of $\llbracket P' \rrbracket$ there must be an execution of $\llbracket P \rrbracket$ with the same behaviour (Theorem 3.2). \square

We should remark that the syntactic elimination transformation we have considered in this section is in fact safe even under sequential consistency. However, with a bit more work, the technique can be extended to prove more interesting eliminations of non-adjacent memory accesses, such as the elimination in the first motivating example in Chapter 1.

4.3 Reordering

Figure 4.7 defines reordering judgement $P \xrightarrow{r} P'$ for programs P and P' . Although the definition only captures the simple case of reordering of two adjacent statements, it allows all cases of reordering from Section 1.4:

- Reordering of two independent non-conflicting non-volatile memory accesses,
- Reordering an acquire statement with a previous independent memory access,
- Reordering a release statement with a following independent memory access,
- Reordering a `print` statement with an independent memory access.

For the purposes of reordering we consider statements independent if they access different registers. Two memory accesses are non-conflicting if they access different shared-memory locations or if they are both reads.

We prove that our syntactic reordering is an instance of the semantic reordering, as described by Definition 3.7. First, we extend the notion of tracesets of programs to code fragments:

Definition 4.4. Let K be a code fragment, i.e., either C , L , S or P from Figure 4.1. The traceset of K in an initial state Λ, σ , denoted by $\llbracket K \rrbracket_{\Lambda, \sigma}$, is a set of all traces t such that:

1. $\Lambda, \sigma, K \Downarrow_{\perp} t$, or
2. there are Λ' and σ' such that $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$.

Observe that semantics reordering is monotone in the source traceset (see Definition 3.6):

$$\begin{array}{c}
\overline{C \xrightarrow{r} C} \qquad \qquad \qquad \overline{L \xrightarrow{r} L} \\
\\
\frac{L \xrightarrow{r} L' \quad S \xrightarrow{r} S'}{S L \xrightarrow{r} S' L'} \text{(R-SEQ)} \qquad \qquad \frac{L \xrightarrow{r} L'}{\{L\} \xrightarrow{r} \{L'\}} \\
\\
\frac{S_1 \xrightarrow{r} S'_1 \quad S_2 \xrightarrow{r} S'_2}{\text{if } (T) \text{ then } S_1 \text{ else } S_2 \xrightarrow{r} \text{if } (T) \text{ then } S'_1 \text{ else } S'_2} \\
\\
\frac{S \xrightarrow{r} S'}{\text{while } (T) S \xrightarrow{r} \text{while } (T) S'} \text{(R-WHILE)} \\
\\
\frac{\forall i \in \{0, \dots, n\}. S_i \xrightarrow{r} S'_i}{S_0 \parallel \dots \parallel S_n \xrightarrow{r} S'_0 \parallel \dots \parallel S'_n} \\
\\
\frac{r_1 \neq r_2 \quad x \text{ not volatile}}{r_1 := x; r_2 := y; \xrightarrow{r} r_2 := y; r_1 := x;} \qquad \frac{x \neq y \quad y \text{ not volatile}}{x := r_1; y := r_2; \xrightarrow{r} y := r_2; x := r_1;} \text{(R-WW)} \\
\\
\frac{r_1 \neq r_2 \quad x \neq y \quad x \text{ or } y \text{ not volatile}}{x := r_1; r_2 := y; \xrightarrow{r} r_2 := y; x := r_1;} \qquad \frac{r_1 \neq r_2 \quad x \neq y \quad x, y \text{ not volatile}}{r_1 := x; y := r_2; \xrightarrow{r} y := r_2; r_1 := x;} \text{(R-RW)} \\
\\
\frac{x \text{ not volatile}}{x := r; \text{lock } m; \xrightarrow{r} \text{lock } m; x := r;} \qquad \frac{x \text{ not volatile}}{r := x; \text{lock } m; \xrightarrow{r} \text{lock } m; r := x;} \\
\\
\frac{x \text{ not volatile}}{\text{unlock } m; x := r; \xrightarrow{r} x := r; \text{unlock } m;} \qquad \frac{x \text{ not volatile}}{\text{unlock } m; r := x; \xrightarrow{r} r := x; \text{unlock } m;} \\
\\
\frac{r_1 \neq r_2 \quad x \text{ not volatile}}{\text{print } r_1; r_2 := x; \xrightarrow{r} r_2 := x; \text{print } r_1;} \qquad \frac{x \text{ not volatile}}{\text{print } r_1; x := r_2; \xrightarrow{r} x := r_2; \text{print } r_1;}
\end{array}$$

Figure 4.7: Syntactic reordering.

Fact 4.5. *If a function f reorders a set of traces T into t and $T \subseteq T'$, then f reorders T' into t .*

The following two lemmata show how to compose traces for elimination and reordering.

Lemma 4.6. *Let trace t' be an elimination of some trace t . Then $t'' \# t'$ is an elimination of the trace $t'' \# t$.*

Proof. By Definition 3.2, there is a set S such that $t' = t|_S$. Let

$$S' = \{0, \dots, |t''| - 1\} \cup \{i + |t''| \mid i \in S\}$$

It is easy to see that $t'' \# t' = (t'' \# t)|_{S'}$ and each $i \in \text{dom}(t'' \# t) \setminus S'$ is eliminable in $t'' \# t$. \square

Lemma 4.7. *Suppose that function f reorders T into t , and function f' reorders T' into t' . Let*

$$\hat{T} = T \cup \{f^\rightarrow(t) \# t'' \mid t'' \in T'\}.$$

Then there is a function g that reorders \hat{T} into $t \# t'$. Moreover $g^\rightarrow(t \# t') = f^\rightarrow(t) \# f'^\rightarrow(t')$.

Proof. Let g be defined by

$$g(i) = \begin{cases} f(i) & \text{if } 0 \leq i < |t| \\ f'(i - |t|) + |t| & \text{if } |t| \leq i < |t| + |t'| \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is straightforward to check that g is a reordering function for $t \# t'$.

It remains to be shown that for each $n \leq |t| + |t'|$ we have

$$[(t \# t')_{g^{-1}(i)} \mid i \leftarrow \text{l dom}(t' \# t). g^{-1}(i) < n] \in \hat{T}$$

Let us denote

$$\tau_n =_{\text{def}} [(t \# t')_{g^{-1}(i)} \mid i \leftarrow \text{l dom}(t' \# t). g^{-1}(i) < n]$$

For $n \leq |t|$ we have

$$\tau_n = [t_{f^{-1}(i)} \mid i \leftarrow \text{l dom}(t). f^{-1}(i) < n]$$

and for $n > |t|$ we obtain

$$\tau_n = f^\rightarrow(t) \# \left[t'_{f^{-1}(i)} \mid i \leftarrow \text{ldom}(t'). f'^{-1}(i) < n \right]$$

In the first case, τ_n must be in $T \subseteq \hat{T}$, because f reorders T to t (Definition 3.6).

In the second case,

$$\left[t'_{f^{-1}(i)} \mid i \leftarrow \text{ldom}(t'). f'^{-1}(i) < n \right] \in T'$$

because f' reorders T' to t' . As a result, we have $\tau_n \in \hat{T}$. Moreover, $g^\rightarrow(t \# t') = \tau_{|t|+|t'|} = f^\rightarrow(t) \# f'^\rightarrow(t')$. \square

Fact 4.8. *Let S be a statement, L be a list of statements, Λ a monitor state and σ a register state. Then $\llbracket S \rrbracket_{\Lambda, \sigma} \subseteq \llbracket S L \rrbracket_{\Lambda, \sigma}$.*

Proof. By Lemma 4.1. \square

The next lemma captures the observation from Example 3.6 in Chapter 3 that a syntactic reordering can be expressed as a semantic elimination followed by a semantic reordering.

Lemma 4.9. *Assume that $K \xrightarrow{r} K'$. Then for each Λ and σ there is a prefix closed set of traces T satisfying these conditions:*

1. *the set of traces $\llbracket K \rrbracket_{\Lambda, \sigma}$ is a subset of T ,*
2. *each trace from T is an elimination of some trace from $\llbracket K \rrbracket_{\Lambda, \sigma}$,*
3. *for each trace t' , if $\Lambda, \sigma, K' \Downarrow_{\perp} t'$ holds then there is a function f that reorders T into t' ,*
4. *for each trace t' , if there are Λ' and σ' such that $\Lambda, \sigma, K' \Downarrow \Lambda', \sigma', t'$ then there is a function f that reorders T into t' and $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', f^\rightarrow(t')$.*

Proof. By induction on the derivation of $K \xrightarrow{r} K'$. We only analyse the interesting cases of the derivation:

1. R-WW: we have

$$\begin{aligned} K &= x := r_1; y := r_2 \\ K' &= y := r_2; x := r_1 \end{aligned}$$

Suppose we have a fixed monitor state Λ and register state σ . From the operational semantics we obtain

$$\llbracket K \rrbracket_{\Lambda, \sigma} = \{[], [\text{Wr}(x, \sigma(r_1))], [\text{Wr}(x, \sigma(r_1)), \text{Wr}(y, \sigma(r_2))]\}$$

$$\llbracket K' \rrbracket_{\Lambda, \sigma} = \{[], [\text{Wr}(y, \sigma(r_2))], [\text{Wr}(y, \sigma(r_2)), \text{Wr}(x, \sigma(r_1))]\}$$

Let

$$T = \{[], [\text{Wr}(x, \sigma(r_1))], [\text{Wr}(x, \sigma(r_1)), \text{Wr}(y, \sigma(r_2))], [\text{Wr}(y, \sigma(r_2))]\}$$

The set T is obviously prefix closed and it satisfies the first condition. The second condition is also satisfied, the only non-trivial elimination is the trace $[\text{Wr}(y, \sigma(r_2))]$, which can be obtained from $[\text{Wr}(x, \sigma(r_1)), \text{Wr}(y, \sigma(r_2))]$ by eliminating the write to x , because the write is a redundant last write if x is not a volatile location, or a redundant release if x is volatile. Moreover, for each trace from $\llbracket K' \rrbracket_{\Lambda, \sigma}$ there is an appropriate reordering function from T : for the empty trace it is the empty function, for $[\text{Wr}(y, \sigma(r_2))]$ it is the function $\{\langle 0, 0 \rangle\}$, and for $[\text{Wr}(y, \sigma(r_2)), \text{Wr}(x, \sigma(r_1))]$ the function $f = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\}$. Finally, we have $\Lambda, \sigma, K \Downarrow \Lambda, \sigma, f^{\rightarrow}([\text{Wr}(y, \sigma(r_2)), \text{Wr}(x, \sigma(r_1))])$, because

$$f^{\rightarrow}([\text{Wr}(y, \sigma(r_2)), \text{Wr}(x, \sigma(r_1))]) = [\text{Wr}(x, \sigma(r_1)), \text{Wr}(y, \sigma(r_2))].$$

2. R-RW: let us fix Λ and σ . We have

$$K = r_1 := x; \quad y := r_2$$

$$K' = y := r_2; \quad r_1 := x$$

$$\llbracket K \rrbracket_{\Lambda, \sigma} = \{[]\} \cup \{[\text{Rd}(x, v)] \mid v \in \mathbb{N}\} \cup \{[\text{Rd}(x, v), \text{Wr}(y, \sigma(r_2))] \mid v \in \mathbb{N}\}$$

$$\llbracket K' \rrbracket_{\Lambda, \sigma} = \{[], [\text{Wr}(y, \sigma(r_2))]\} \cup \{[\text{Wr}(y, \sigma(r_2)), \text{Rd}(y, v)] \mid v \in \mathbb{N}\}$$

Let

$$T = \llbracket K \rrbracket_{\Lambda, \sigma} \cup \{[\text{Wr}(y, \sigma(r_2))]\}.$$

It is easy to check that T satisfies the requirements: Note that the trace $[\text{Wr}(y, \sigma(r_2))]$ can be obtained by irrelevant read elimination from $[\text{Rd}(x, *), \text{Wr}(y, \sigma(r_2))]$, and $[\text{Rd}(x, *), \text{Wr}(y, \sigma(r_2))]$ belongs to $\llbracket K \rrbracket_{\Lambda, \sigma}$.

3. R-SEQ: assume that there are S, L, S' and L' such that

$$K = S L \quad (4.7)$$

$$K' = S' L' \quad (4.8)$$

$$L \xrightarrow{r} L' \quad (4.9)$$

$$S \xrightarrow{r} S' \quad (4.10)$$

Let Λ and σ be a fixed monitor state and a fixed register state. From the induction hypothesis we obtain a set T_S that satisfies the conditions from the statement of this lemma for $K = S$ and $K' = S'$ in state Λ and σ . Moreover, for any state Λ', σ' we have a set $T_L(\Lambda', \sigma')$ satisfying the statement for $K = L, K' = L'$ and state Λ', σ' . We define the set T as

$$T = T_S \cup \{t \# t' \mid \exists \Lambda' \sigma'. \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \wedge t' \in T_L(\Lambda', \sigma')\}$$

Since T_S and $T_L(\Lambda', \sigma')$ are prefix closed, T must be also prefix-closed. Set T satisfies the required conditions:

- (a) $\llbracket S L \rrbracket_{\Lambda, \sigma}$ is a subset of T : Let us assume that $t \in \llbracket S L \rrbracket_{\Lambda, \sigma}$. By Definition 4.4 and the operational semantics, this can happen in three ways:
- i. SEQ-1: $\Lambda, \sigma, S \Downarrow_{\perp} t$. Then $t \in T_S \subseteq T$.
 - ii. SEQ-2: There are t' and t'' such that $t = t' \# t''$, $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t'$ and $\Lambda', \sigma', L \Downarrow_{\perp} t''$. Hence, $t'' \in T_L(\Lambda', \sigma')$ and $t' \# t'' \in T$.
 - iii. SEQ: Analogous to the previous case.
- (b) $\forall t \in T. \exists t' \in \llbracket S L \rrbracket_{\Lambda, \sigma}. t$ is an elimination of t' . Let $t \in T$. We analyse the possible cases:
- i. $t \in T_S$. By the induction hypothesis, there is a wildcard trace t' from $\llbracket S \rrbracket_{\Lambda, \sigma}$ such that t is an elimination of t' . By Fact 4.8, the trace t' is in $\llbracket S L \rrbracket$.
 - ii. there are trace t_0 , wildcard trace t_1 , monitor state Λ' and register state σ' such that $t = t_0 \# t_1$, $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t_0$ and t_1 is in $T_L(\Lambda', \sigma')$. By the induction hypothesis, there is t'_1 in $\llbracket L \rrbracket_{\Lambda', \sigma'}$ such that t_1 is an elimination of t'_1 . From Lemma 4.6, the trace $t_0 \# t'_1$ is an elimination of t . Using the rules SEQ-1 and SEQ of the operational semantics, $t_0 \# t'_1$ is in $\llbracket S L \rrbracket_{\Lambda, \sigma}$.

- (c) $\forall t'. (\Lambda, \sigma, S' L' \Downarrow_{\perp} t') \Rightarrow \exists f. f \text{ reorders } T \text{ into } t'$. Let us assume that $\Lambda, \sigma, S' L' \Downarrow_{\perp} t'$. From the operational semantics, there are two cases
- i. SEQ-1: if $t' \in \llbracket S' \rrbracket$, then we use the induction hypothesis for $S \xrightarrow{r} S'$ to get the function f .
 - ii. SEQ-2, i.e., there are t'_0, t'_1, Λ' and σ' such that $t' = t'_0 \# t'_1$, $\Lambda, \sigma, S' \Downarrow \Lambda', \sigma', t'_0$ and $\Lambda', \sigma', L' \Downarrow_{\perp} t'_1$. Using the induction hypothesis, we obtain f_0 and f_1 such that f_0 reorders T_S into t'_0 and f_1 reorders $T_L(\Lambda', \sigma')$ into t'_1 , and $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', f_0^{-1}(t'_0)$. Using Lemma 4.7 we obtain function g that reorders $T_S \cup \{f_0^{-1}(t'_0) \# t'' \mid t'' \in T_L(\Lambda', \sigma')\}$ into $t'_0 \# t'_1$. By Fact 4.5, we establish that g reorders T into t' .
- (d) $\forall t' \Lambda' \sigma'. (\Lambda, \sigma, S' L' \Downarrow \Lambda', \sigma', t') \Rightarrow \exists f. f \text{ reorders } T \text{ into } t' \text{ and } \Lambda, \sigma, S L \Downarrow \Lambda', \sigma', f^{-1}(t')$. Let us assume that $\Lambda, \sigma, S' L' \Downarrow \Lambda', \sigma', t'$. By the rule (SEQ) of the operational semantics, there are t'_0, t'_1, Λ'' and σ'' such that $t' = t'_0 \# t'_1$ and $\Lambda, \sigma, S' \Downarrow \Lambda'', \sigma'', t'_0$ and $\Lambda'', \sigma'', L' \Downarrow \Lambda', \sigma', t'_1$. Using the same reasoning as in the case 3(c)ii we obtain function g that reorders T into $t'_0 \# t'_1$ and $g^{-1}(t'_0 \# t'_1) = f^{-1}(t'_0) \# f'^{-1}(t'_1)$. By the induction hypothesis we have $\Lambda, \sigma, S \Downarrow \Lambda'', \sigma'', f^{-1}(t'_0)$ and $\Lambda'', \sigma'', L \Downarrow \Lambda', \sigma', f'^{-1}(t'_1)$. With the rule (SEQ) we obtain $\Lambda, \sigma, S L \Downarrow \Lambda', \sigma', g^{-1}(t')$.

4. R-WHILE: $K = \mathbf{while} (C) S, K' = \mathbf{while} (C) S'$. By the induction hypothesis, for any state Λ', σ' we have a set $T_S(\Lambda', \sigma')$ satisfying the statement of the lemma for $K = S, K' = S'$ and state Λ', σ' . We will build the set T in stages:

$$\begin{aligned}
 T_0(\Lambda, \sigma) &= \emptyset \\
 T_{i+1}(\Lambda, \sigma) &= \begin{cases} \{\emptyset\} & \text{if } \text{Val}(\sigma, C) = \mathbf{ff} \\ T_S(\Lambda, \sigma) \cup \{t \# t' \mid \exists \Lambda' \sigma'. \Lambda, \sigma, S \Downarrow \Lambda', \sigma', t \wedge t' \in T_i(\Lambda', \sigma')\} & \\ \text{otherwise} & \end{cases} \\
 T(\Lambda, \sigma) &= \bigcup_{i \in \mathbb{N}} T_i(\Lambda, \sigma)
 \end{aligned}$$

By induction on i and prefix closedness of $T_S(\Lambda, \sigma)$ and Lemma 4.1, T is prefix closed. Let Λ and σ be fixed monitor and register states. The set $T = T(\Lambda, \sigma)$ has the required properties:

- (a) $\llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$ is a subset of T . If $t \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$ then

either $\Lambda, \sigma, \mathbf{while} (C) S \Downarrow \Lambda', \sigma', t$ or $\Lambda, \sigma, \mathbf{while} (C) S \Downarrow_{\perp} t$. We will show that $t \in T$ only for the case when $\Lambda, \sigma, \mathbf{while} (C) S \Downarrow_{\perp} t$. The other case is very similar. For arbitrary Λ and σ , we show by induction on the derivation of $\Lambda, \sigma, K \Downarrow_{\perp} t$ that $K = \mathbf{while} (C) S$ implies existence of i such that $t \in T_i(\Lambda, \sigma)$. The relevant rules of the operational semantics are:

- i. LOOP-F1: assume that $t = []$ and $\text{Val}(\sigma, C) = \mathbf{ff}$. Then $t \in T_1(\Lambda, \sigma)$.
- ii. LOOP-T1: let us assume that $\text{Val}(\sigma, C) = \mathbf{tt}$ and $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t_1$ and $\Lambda', \sigma', \mathbf{while} (C) S \Downarrow_{\perp} t_2$ and $t = t_1 \# t_2$. From the induction hypothesis we obtain i such that $t_2 \in T_i(\Lambda', \sigma')$. By the definition of T_{i+1} , $t_1 \# t_2 \in T_{i+1}(\Lambda, \sigma)$.
- iii. LOOP-T2: assume $\text{Val}(\sigma, C) = \mathbf{tt}$ and $\Lambda, \sigma, S \Downarrow_{\perp} t$. By the induction hypothesis from the outer induction on the derivation $K \xrightarrow{r} K', t \in T_S(\Lambda, \sigma) \subseteq T_1(\Lambda, \sigma)$.

(b) $\forall t' \in T. \exists t \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}. t'$ is an elimination of t . We prove by induction on i that for all Λ and σ , if $t' \in T_i(\Lambda, \sigma)$, then there is $t \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$ such that t' is an elimination of t . The case $i = 0$ is trivial. Let $t' \in T_{i+1}(\Lambda, \sigma)$. There are three cases:

- i. $t' = []$ and $\text{Val}(\sigma, C) = \mathbf{ff}$. Then we take $t = []$.
- ii. $\text{Val}(\sigma, C) = \mathbf{tt}$ and $t' \in T_S(\Lambda, \sigma)$. By the induction hypothesis on the derivation of $K \xrightarrow{r} K'$, there is $t \in \llbracket S \rrbracket_{\Lambda, \sigma}$ such that t' is an elimination of t . By the rules of the operational semantics and Lemma 4.1, $t \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$.
- iii. $\text{Val}(\sigma, C) = \mathbf{tt}$ and there are $t'_0, t'_1, \Lambda', \sigma'$ such that $t' = t'_0 \# t'_1$, $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', t'_0$ and $t'_1 \in T_i(\Lambda', \sigma')$. By the induction hypothesis, there is $t_1 \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$ such that t'_1 is an elimination of t_1 . By Lemma 4.6, the trace $t'_0 \# t_1$ is an elimination of t' . By the rules LOOP-T and LOOP-T1, $t'_0 \# t_1 \in \llbracket \mathbf{while} (C) S \rrbracket_{\Lambda, \sigma}$.

(c) $\forall t'. (\Lambda, \sigma, \mathbf{while} (C) S' \Downarrow_{\perp} t') \Rightarrow \exists f. f$ reorders T into t' . We will prove by induction on the derivation of $\Lambda, \sigma, K' \Downarrow_{\perp} t'$ that $K' = \mathbf{while} (C) S'$ implies that there is i and f such that f reorders $T_i(\Lambda, \sigma)$ into t' . The applicable rules are:

- i. LOOP-F1: assume that $\text{Val}(\sigma, C) = \mathbf{ff}$ and $t' = []$. Then the empty function reorders $T_1(\Lambda, \sigma)$ into t' .
 - ii. LOOP-T1: let us assume that $\text{Val}(\sigma, C) = \mathbf{tt}$ and $\Lambda, \sigma, S' \Downarrow \Lambda', \sigma', t'_0$ and $\Lambda', \sigma', \mathbf{while} (C) S' \Downarrow_\perp t'_1$ and $t' = t'_0 \# t'_1$. From the induction hypothesis of the outer induction on the derivation $K \xrightarrow{x} K'$ we obtain function f_S that reorders $T_S(\Lambda, \sigma)$ into t'_0 and $\Lambda, \sigma, S \Downarrow \Lambda', \sigma', f_S^{-1}(t'_0)$. From the induction hypothesis of the inner induction on the operational semantics we obtain f_w and i such that f_w reorders $T_i(\Lambda'', \sigma'')$ into t'_1 . By Lemma 4.7, there is g that reorders $T_S(\Lambda, \sigma) \cup \{f_S^{-1}(t'_0) \# t'' \mid t'' \in T_i(\Lambda'', \sigma'')\}$ into $t'_0 \# t'_1$. Since $T_S(\Lambda, \sigma) \cup \{f_S^{-1}(t'_0) \# t'' \mid t'' \in T_i(\Lambda'', \sigma'')\}$ is a subset of $T_{i+1}(\Lambda, \sigma)$, the function g reorders $T_{i+1}(\Lambda, \sigma)$ into t' .
 - iii. LOOP-T2: let us assume that $\text{Val}(\sigma, C) = \mathbf{tt}$ and $\Lambda, \sigma, S' \Downarrow_\perp t'$. From the induction hypothesis of the outer induction we obtain f that reorders $T_S(\Lambda, \sigma)$ into t' . Since $T_S(\Lambda, \sigma) \subseteq T_1(\Lambda, \sigma)$, we are done.
- (d) $\forall t' \Lambda' \sigma'. (\Lambda, \sigma, \mathbf{while} (C) S' \Downarrow \Lambda', \sigma', t') \Rightarrow \exists f. f \text{ reorders } T \text{ into } t' \text{ and } \Lambda, \sigma, \mathbf{while} (C) S \Downarrow \Lambda', \sigma', f^{-1}(t')$. By induction on the derivation of $\Lambda, \sigma, K' \Downarrow \Lambda', \sigma', t'$ we will show that $K' = \mathbf{while} (C) S'$ implies that there is i and f such that f reorders T_i into t' and $\Lambda, \sigma, \mathbf{while} (C) S \Downarrow \Lambda', \sigma', f^{-1}(t')$. The two relevant rules of the operational semantics are:

- i. LOOP-F: if $\text{Val}(\sigma, C) = \mathbf{ff}$ and $t' = []$, then the empty function reorders T_1 into t' . Moreover, $\Lambda, \sigma, \mathbf{while} (C) S \Downarrow \Lambda', \sigma', []$.
- ii. LOOP-T: let us assume that $\text{Val}(\sigma, C) = \mathbf{tt}$ and $\Lambda, \sigma, S' \Downarrow \Lambda'', \sigma'', t'_0$ and $\Lambda'', \sigma'', \mathbf{while} (C) S' \Downarrow \Lambda', \sigma', t'_1$ and $t' = t'_0 \# t'_1$. From the induction hypothesis of the outer induction on the derivation $K \xrightarrow{x} K'$ we obtain function f_S that reorders $T_S(\Lambda, \sigma)$ into t'_0 and

$$\Lambda, \sigma, S \Downarrow \Lambda'', \sigma'', f_S^{-1}(t'_0). \quad (4.11)$$

From the induction hypothesis of the inner induction on operational semantics we obtain f_w and i such that f_w reorders $T_i(\Lambda'', \sigma'')$ into t'_1 and

$$\Lambda'', \sigma'', \mathbf{while} (C) S \Downarrow \Lambda', \sigma', f_w^{-1}(t'_1). \quad (4.12)$$

By Lemma 4.7, there is g that reorders $T_S(\Lambda, \sigma) \cup \{f_S^{-1}(t'_0) \# t'' \mid t'' \in T_i(\Lambda'', \sigma'')\}$ to $t'_0 \# t'_1$ and

$$g^{-1}(t'_0 \# t'_1) = f_S^{-1}(t'_0) \# f_w^{-1}(t'_1). \quad (4.13)$$

Since $T_S(\Lambda, \sigma) \cup \{f_S^{-1}(t'_0) \# t'' \mid t'' \in T_i(\Lambda'', \sigma'')\} \subseteq T_{i+1}(\Lambda, \sigma)$, the function g reorders $T_{i+1}(\Lambda, \sigma)$ into t' . From 4.11, 4.12, 4.13 and rule LOOP-T we get

$$\Lambda, \sigma, \text{while } (C) \ S \Downarrow \Lambda', \sigma', g^{-1}(t').$$

The remaining rules for reordering are straightforward or similar to the above. \square

Corollary 4.10. *Suppose that $P \xrightarrow{r} P'$. Then there is a traceset T such that $\llbracket P' \rrbracket$ is a reordering of T and T is an elimination $\llbracket P \rrbracket$.*

Proof. By Lemma 4.9, there is a prefix closed T such that each trace from T is an elimination of a trace from $\llbracket P \rrbracket$, and for each trace $t' \in \llbracket P' \rrbracket$ there is a function that reorders T into t' .

Since each trace $t \in T$ is an elimination of some properly started t' from the properly locked set $\llbracket P \rrbracket$ and eliminations cannot eliminate acquire and thread start actions, T must be properly locked and properly started. T is properly typed, because each action from t must be also in t' . Therefore, T is a traceset, and by Definitions 3.7 and 3.3, T is an elimination of $\llbracket P \rrbracket$ and $\llbracket P' \rrbracket$ is a reordering of T . \square

Theorem 4.2. *Suppose that $P \xrightarrow{r} P'$ and $\llbracket P \rrbracket$ is data race free. Then $\llbracket P' \rrbracket$ is data race free, and any execution of $\llbracket P' \rrbracket$ has the same behaviour as some execution of $\llbracket P \rrbracket$.*

Proof. From Corollary 4.10, there is a traceset T such that $\llbracket P' \rrbracket$ is a reordering of T and T is an elimination $\llbracket P \rrbracket$. Both T and $\llbracket P' \rrbracket$ must be data race free by Theorems 3.1 and 3.4. Let I' be an execution of $\llbracket P' \rrbracket$. Then there is some execution \hat{I} of T with the same behaviour as I' (Theorem 3.3), and \hat{I} has the same behaviour as some execution of $\llbracket P \rrbracket$ (Theorem 3.2). \square

4.4 Out-of-thin-air

To show the out-of-thin-air guarantee for our toy language, let us make a simple observation:

Lemma 4.11. *Let v be a value such that v is not a default value for any location, i.e., $v \neq 0$. Let P be a program without any statement of the form $r := v$, where r is a register name. Then no trace in the traceset of P is an origin for the value v .*

Proof. It is straightforward to show by induction on the derivation of $\Lambda, \sigma, K \Downarrow \Lambda', \sigma', t$ and $\Lambda, \sigma, K \Downarrow_{\perp} t$ that whenever K does not contain the statement of the form $r := v$ and $\sigma(r) \neq v$ for all registers r , the trace t is not an origin for v . Since $\sigma_{\text{init}}(r) = 0 \neq v$ for all registers r , we have that for any $t \in \llbracket P \rrbracket$, t is not an origin for v . \square

Let us summarise the out-of-thin-air guarantee:

Theorem 4.3. *Suppose that c is a constant different from 0, and P a program that does not contain a statement of the form $r := c$, where r is a register. Let P' be a program obtained from P by reordering or by an elimination, i.e., either $P \xrightarrow{r} P'$ or $P \xrightarrow{e} P'$. Then P' cannot output c .*

Proof. Direct consequence of Lemmata 4.11, 3.25, 3.24 and 3.26. \square

Chapter 5

Java Memory Model

5.1 Introduction

The first edition of the Java language specification defined a multi-threaded semantics that was intentionally weaker than sequential consistency to allow common compiler and hardware optimisations (Gosling et al., 1996). However, Pugh (2000) found that this semantics did not allow many compiler transformations and the consensus was that the Java Memory Model was broken. As a result of this, Manson et al. (2005) designed a new memory model and claimed to have proved that it satisfies the DRF guarantee, provides out-of-thin-air guarantees and allows reordering of independent statements. This memory model became part of the current edition of the Java language specification (Gosling et al., 2005a). Recently, Cenciarelli et al. (2007) found a counterexample to validity of the reordering transformation, and suggested that the problem could be fixed. To fix the memory model, we proposed a small change to the JMM in Aspinall and Ševčík (2007). We provided a machine-checked proof of the DRF guarantee and conjectured validity of reordering for our alternative JMM.

In this chapter, we will analyse validity of the transformations from Section 1.4 in the current Java Memory Model and the alternative JMM. We devise an operational view of the JMM (Section 5.2), and use this interpretation to explain the counterexamples for the invalid transformations, such as read after read elimination, irrelevant read introduction or roach motel reordering (Section 5.3). In Section 5.4, we prove that our alternative definition of the JMM indeed allows reordering and both versions of the JMM also allow eliminations of redundant reads after writes, writes after writes and irrelevant reads. Finally, we discuss

the practical impact of these results in Section 5.5, and show that even Sun’s HotSpot Java Virtual Machine does not comply with the specification.

This work has been reported on in Ševčík and Aspinall (2008) and most of the text in this chapter originates from that paper.

5.2 JMM Operationally

To reason about the Java Memory Model, we first describe the formal definition and then introduce an intuitive operational interpretation, based on some observations about the construction of the formal definition¹. This re-interpretation will allow us to explain key counterexamples in a direct way in the next section.

Unlike the interleaved semantics studied in the previous chapters, the Java Memory Model has no explicit global ordering of all actions by time consistent with each thread’s perception of time, and has no global store. Instead, executions are described in terms of memory related actions, partial orders on these actions, and a visibility function that assigns a write action to each read action. Therefore, we redefine some of the previously used notation to capture the foundations of the JMM. We first explain the building blocks of Java executions, then we show how Java builds *legal executions* out of simple “well-behaved” executions.

5.2.1 JMM actions and orders

An *action* in the Java Memory Model is a memory-related operation. Note that the actions have their own unique identity. This is different from the actions in the interleavings, as defined in Chapter 2, where the action’s implicit identity was its position in an interleaving or a trace.

Definition 5.1. Actions are modelled by an abstract type \mathcal{A} with the following properties:

1. Each action belongs to one thread, we will denote it by $T(a)$.
2. An action has one of the following *action kinds*:

¹Jeremy Manson made essentially the same observations in his description of his model checker for the JMM (Manson, 2004).

- *volatile read* $Rd_v(v)$,
- *volatile write* $Wr_v(v)$,
- *normal read* $Rd(x)$,
- *normal write* $Wr(x)$,
- *external action* $Ext(i)$,
- *lock* $L(m)$,
- *unlock* $U(m)$,
- *thread start* $S(\theta)$,
- *thread finish* $Fin(\theta)$,

where x is a non-volatile memory location, v is a volatile memory location, m is a synchronisation monitor, and θ is a thread identifier. In the spirit of the JMM, we consider an external action to be an output of a value. The volatile read/write and lock/unlock actions are called *synchronisation actions*. We denote the action kind of a by $K(a)$.

An execution of a program is a set of actions that it performs, partial orders that restrict the visibility of the actions, and the visibility functions:

Definition 5.2. An *execution* E is a tuple $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$, where

- $A \subseteq \mathcal{A}$ is a set of actions,
- P is a program, which is represented as a set of memory traces,
- the partial order $\leq_{po} \subseteq A \times A$ is the program order, which is a union of total orders on actions of each thread,
- the partial order $\leq_{so} \subseteq A \times A$ is the synchronisation order, which is a total order on all synchronisation actions in A ,
- $V :: \mathcal{A} \Rightarrow \mathcal{V}$ is a *value-written* function that assigns a value to each write from A , $V(a)$ is unspecified for non-write actions a ,
- $W :: \mathcal{A} \Rightarrow \mathcal{A}$ is a *write-seen* function. It assigns a write to each read action from A , the $W(r)$ denotes the write seen by r , i.e., the value read by r is $V(W(r))$. The value of $W(a)$ for non-read actions a is unspecified.

To relate a program to a sequence of actions performed by one thread we must define a notion of *sequential validity*. We consider programs as sets of sequences of pairs of an action kind and a value, which we call *traces*.

Definition 5.3. Given an execution $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$, the *action trace* of thread θ in E , denoted $\overline{\text{Tr}}_E(\theta)$, is the list of actions of thread θ in the order \leq_{po} . The *trace* of thread θ in E , written $\text{Tr}_E(\theta)$ is the list of action kinds and corresponding values obtained from the action trace (i.e., $V(W(a))$ if a is a read, $V(a)$ otherwise).

By writing $t \leq t'$ we mean that t is a prefix of t' , $\text{set}(t)$ is the set of elements of the list t , $\iota(t, a)$ is the first index i such that $t_i = a$, or 0 if $a \notin \text{set}(t)$. For an action kind-value pair $p = \langle k, v \rangle$ we will use the notation $\mathcal{K}(p)$ for the action kind k and $\mathcal{V}(p)$ for the value v . We say that a sequence s of action kind-value pairs is *sequentially valid* with respect to a set of traces P if $t \in P$. The operator $\#$ stands for trace concatenation.

A multi-thread program is a set of single-thread traces. We assume some reasonable properties of programs, and we call the programs satisfying these properties tracesets:

Definition 5.4. We say that a set of traces P is a *traceset*² if sequential validity of trace t in P implies:

1. any trace $t' \leq t$ is sequentially valid (prefix closedness),
2. if the last action of t is a read with value v , then the trace obtained from t by replacing the value in the last action by any value v' is also sequentially valid in P (final read value independence),
3. $|t| > 0$ implies $\mathcal{K}(t_0) = \text{S}(\theta)$ for some θ ; moreover, $0 < i < |t|$ implies $\mathcal{K}(t_i) \neq \text{S}(\theta)$ (start action first),
4. $\mathcal{K}(t_i) = \text{Fin}(\theta)$ implies $i = |t| - 1$ (finish action last),
5. $|t| > 0$ and $t_0 = \text{S}(\theta_{init})$ implies $\forall i. 1 \leq i < |t| - 1 \rightarrow \exists v. \mathcal{K}(t_i) = \text{Wr}(v) \vee \mathcal{K}(t_i) = \text{Wr}_v(v) \vee \mathcal{K}(t_i) = \text{Fin}(\theta_{init})$ (initialisation thread only contains writes).

Definition 5.5. Pair of synchronisation actions a, b is a *release-acquire* pair if either

- a is an unlock of monitor m and b is a lock of m , or

²This notion is slightly stronger than the one we used in previous chapter, c.f. Definition 2.3. The main addition is the final read value independence condition.

- a is a volatile write to v and b is a volatile read from v , or
- a is a finish action of the initialisation thread and b is a start of a non-initialisation thread.

Action a *synchronises-with* action b if $a \leq_{so} b$ and a, b are a release-acquire pair.

From the program order and synchronises-with orders we construct a *happens-before* order of the execution.

Definition 5.6. The *happens-before* order of an execution is the transitive closure of the composition of its synchronises-with order and its program order, i.e. $\leq_{hb} = (\leq_{sw} \cup \leq_{po})^+$.

The happens-before order determines an upper bound on the visibility of writes—a read happening before a write should never see that write, and a read r should not see a write w if there is another write happening “in-between”, i.e., if $w \leq_{hb} w' \leq_{hb} r$ and $w \neq w'$, then r cannot see w .

The next definition summarises the restrictions on well-formed executions.

Definition 5.7. We say that an *execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ is *well-formed* if

1. A is finite.
2. P is a traceset,
3. \leq_{po} restricted on actions of one thread is a total order, \leq_{po} does not relate actions of different threads.
4. \leq_{so} is total on synchronisation actions of A .
5. \leq_{so} is consistent with \leq_{po} .
6. W is properly typed: for every non-volatile read $r \in A$, $W(r)$ is a non-volatile write; for every volatile read $r \in A$, $W(r)$ is a volatile write.
7. Locking is proper: for all lock actions $l \in A$ on monitors m and all threads θ different from the thread of l , the number of locks in θ before l in \leq_{so} is the same as the number of unlocks in θ before l in \leq_{so} .

8. Program order is intra-thread consistent: for each thread θ , the trace of θ in E is sequentially valid for P ; moreover, for any $a \in A$, if $K(a) = S(\theta)$ or $K(a) = \text{Fin}(\theta)$ then $T(a) = \theta$.
9. \leq_{so} is consistent with W : for every volatile read r of a variable v we have $W(r) \leq_{so} r$ and for any volatile write w to v , either $w \leq_{so} W(r)$ or $r \leq_{so} w$.
10. \leq_{hb} is consistent with W : for all reads r of v it holds that $r \not\leq_{hb} W(r)$ and there is no intervening write w to v , i.e. if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then $W(r) = w$.
11. The initialisation thread θ_{init} finishes before any other thread starts, i.e.,

$$\forall \theta. \forall a, b \in A. K(a) = \text{Fin}(\theta_{init}) \wedge K(b) = S(\theta) \wedge \theta \neq \theta_{init} \rightarrow a \leq_{so} b.$$

We say that an execution is *sequentially consistent* if there is a total order consistent with the program order, such that for each read r , the write $W(r)$ is the most recent write to the same variable in that total order. A pair of non-volatile memory accesses to the same variable is called a *data race* if at least one of the accesses is a write and they are not ordered by the happens-before order. A program is *correctly synchronised* (or *data-race-free*) if no sequentially consistent execution contains a data race.

A thorny issue is initialisation of variables. The JMM says

The write of the default value (zero, false, or null) to each variable synchronises-with to the first action in every thread (Gosling et al., 2005b)

However, normal writes are not synchronisation actions and synchronises-with only relates synchronisation actions, so normal writes cannot synchronise-with any action. For this study, we assume that all default writes are executed in a special initialisation thread and the thread is finished before all other threads start.

5.2.2 Committing semantics

The basic building blocks are *well-behaved* executions, in which reads are only allowed to see writes that happen before them, i.e., for each read $r \in A$ we have $W(r) \leq_{hb} r$. In other words, in these executions reads cannot see writes

		initially $x = y = 0$			
		lock m1	lock m2		
		r1=x	r2=y		
initially $x = y = 0$		unlock m1	unlock m2	initially $x = y = 0$	
r1 = x	r2 = y	lock m2	lock m1	r1 = x	r2 = y
y = r1	x = 1	y=r1	x=1	y = r1	x = r2
		unlock m2	unlock m1		

A. (allowed)

B. (prohibited)

C. (prohibited)

Is it possible to get $r1 = r2 = 1$ at the end of an execution?

Figure 5.1: Examples of legal and illegal executions from Sections 1.1 and 1.2.

through data races, and threads can only communicate through synchronisation. For example, the programs³ A and C in Figure 5.1 have just one such execution—the one where $r1 = r2 = 0$. On the other hand, the behaviours of program B are exactly the behaviours that could be observed by the interleaved semantics, i.e. $r1 = r2 = 0$, or $r1 = 1$ and $r2 = 0$. In fact, if a program is correctly synchronised then its execution is well-behaved if and only if it is sequentially consistent (Manson et al., 2005, Lemma 2). This does not hold for incorrectly synchronised programs (e.g., see the first counterexample in Section 5.3).

The Java Memory Model starts from a well-behaved execution and *commits* one or more read-write data races from the well-behaved execution. After committing the actions involved in the data races it “restarts” the execution, but this time it must execute the committed actions. This means that each read in the execution must be either committed and see the value through the race, or it must see the write that happens-before it. Similarly, all committed writes must be executed in the restarted execution and must write the same value as in the execution where they were committed. The JMM can repeat the process, i.e., it may choose some non-committed reads involved in a data race, commit the writes involved in these data races if they are not committed already, commit the chosen reads, and restart the execution. The executions constructed using this procedure are called *legal executions*.

The JMM imposes several requirements on the committing sequence:

³The programs in Figure 5.1 are the motivating examples from Sections 1.1 and 1.2.

1. All subsequent (restarted) executions must preserve happens-before ordering restricted to the set of the committed actions. Cenciarelli et al. (2007) observed that this requirement makes reordering of independent statements invalid. In earlier work (Aspinall and Ševčík, 2007), we suggested that the happens-before ordering should be preserved only between a read and the write it sees. We showed there that this revision still satisfies the DRF guarantee; in this chapter we further establish that validity of reordering is indeed rescued in this version.
2. If some synchronisation happens-before the committed data race(s), the synchronisation must be preserved in all subsequent executions⁴.
3. All external actions that happen-before any committed action must be committed, as well.

This committing semantics imposes a causality order on races—the outcome of a race must be explained in terms of previously committed races. This prevents causality loops, where the outcome of a race depends on the outcome of the very same race, e.g., the outcome `r1 = 1` in program C in Figure 5.1. The DRF guarantee is a simple consequence of this procedure. If there are no data races in the program, there is nothing to commit, and we can only generate well-behaved executions, which are sequentially consistent for data race free programs.

In fact, the JMM, as defined in Gosling et al. (2005b), actually commits all actions in an execution, but committing a read that sees a write that happens-before it does not create any opportunities for committing new races, because reads can see writes that happen-before them in a well-behaved execution. This is why we need to consider only read-write races and not write-write races. Similarly, committing synchronisation actions does not create any committing opportunities and can be always performed in the last step. Therefore, the central issue is committing read-write data races, and we explain our examples using this observation.

The formal definition of the legal execution follows. In our work, we use a weakened version of the memory model that we suggested in Aspinall and Ševčík (2007) and which permits more transformations than the original version.

⁴For a formal definition, see rule 8 in the list that follows Definition 5.8.

Definition 5.8. A well-formed execution $\langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ with happens before order \leq_{hb} is *legal* if there is a finite sequence of sets of actions C_i and well-formed executions $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i \rangle$ with happens-before \leq_{hb_i} and synchronises-with $<_{sw_i}$ such that $C_0 = \emptyset$, $C_{i-1} \subseteq C_i$ for all $i > 0$, $\bigcup C_i = A$, and for each $i > 0$ the following rules are satisfied:

1. $C_i \subseteq A_i$.
2. For all reads $r \in C_i$ we have $W(r) \leq_{hb} r \iff W(r) \leq_{hb_i} r$, and $r \not\leq_{hb_i} W(r)$,
3. $V_i|_{C_i} = V|_{C_i}$.
4. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$.
5. For all reads $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$.
6. For all reads $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$.
7. If $y \in C_i$ is an external action and $x \leq_{hb} y$ then $x \in C_i$.

The original definition of legality from Gosling et al. (2005b); Manson et al. (2005) differs in rules 2 and 6, and adds rule 8:

2. $\leq_{hb_i}|_{C_i} = \leq_{hb}|_{C_i}$.
6. For all reads $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$ and $W_i(r) \in C_{i-1}$.
8. If $x <_{ssw_i} y \leq_{hb_i} z$ and $z \in C_i - C_{i-1}$, then $x <_{sw_j} y$ for all $j \geq i$, where $<_{ssw_i}$ is the transitive reduction of \leq_{hb_i} without any \leq_{po_i} edges, and the transitive reduction of \leq_{hb_i} is a minimum relation such that its transitive closure is \leq_{hb_i} .

The reasons for weakening the rules are invalidity of reordering of independent statements, broken JMM causality tests 17–20 Pugh and Manson (2004), and redundancy. For details, see Aspinall and Ševčík (2007); Cenciarelli et al. (2007).

We describe the operational view formally in Definition 5.9. Note that the operational interpretation builds the justification incrementally—it never refers to the final execution in the justification conditions. Theorem 5.1 states the equivalence of the operational justification with the previous definition of legal executions (Definition 5.8).

Definition 5.9. The sequence of sets of actions $\{C_i\}_{i=0}^n$ and the sequence of well-formed executions $\{E_i\}_{i=0}^n$ where $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i \rangle$ and $C_0 = \emptyset$ are a *data-race-justification* for execution E_n if for each $i \in \{1, \dots, n\}$ we have:

1. $C_i \subseteq A_i$.
2. $V_i|_{C_i} = V_{i-1}|_{C_i}$.
3. $W_i|_{C_{i-1}} = W_{i-1}|_{C_{i-1}}$.
4. $a \in C_i$ implies that $a \in A_{i-1}$ and either:
 - (a) a is a read such that $W_i(a) \in C_i$, and a and $W_i(a)$ are not ordered by happens-before in both E_{i-1} and E_i , i.e., a and $W_i(a)$ are a data race in E_{i-1} and E_i , or
 - (b) a is a write such that there is a read r in $C_i \setminus C_{i-1}$ with $W_i(r) = a$.
5. for all $j \in \{0, \dots, n\}$ and for all reads $r \in A_j - C_j$ we have $W_j(r) \leq_{hb_j} r$, where \leq_{hb_j} is the happens-before order of E_j .

Before we prove the equivalence of this definition with Definition 5.8, let us observe that we need not commit all actions in an otherwise well-behaved execution to show its legality:

Lemma 5.1. *Let $\{C_i\}_{i=0}^n$ be a finite sequence of sets of actions, $\{E_i\}_{i=0}^n$ be a finite sequence of well-formed executions such that $C_0 = \emptyset$, $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i \rangle$, E_i has happens-before order \leq_{hb_i} for all $i \in \{0, \dots, n\}$, and the following rules are satisfied for all $i \in \{1, \dots, n\}$:*

1. $C_i \subseteq A_i$.
2. For all reads $r \in C_i$ we have $W_n(r) \leq_{hb_n} r \iff W_n(r) \leq_{hb_i} r$, and $r \not\leq_{hb_i} W(r)$,
3. $V_i|_{C_i} = V_n|_{C_i}$.
4. $W_i|_{C_{i-1}} = W_n|_{C_{i-1}}$.
5. For all reads $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$.
6. For all reads $r \in C_i - C_{i-1}$ we have $W_n(r) \in C_{i-1}$.

7. If $y \in C_i$ is an external action and $x \leq_{hb_n} y$ then $x \in C_i$.

Then E_n is a legal execution.

Proof. Let C_{n+1} be the set of all writes that are in $A_n \setminus C_n$, let $C_{n+2} = A_n \setminus C_{n+1}$ and $E_{n+2} = E_{n+1} = E_n$. Then $\{C_i\}_{i=0}^{n+2}$ and $\{E_i\}_{i=0}^{n+2}$ are justifying sequences for legality of the execution E_n . \square

Theorem 5.1. *There is a data-race-justification for E if and only if E is a legal execution.*

Proof. “ \Rightarrow ”: let $\{C_i\}_{i=0}^n, \{E_i\}_{i=0}^n$ be a data-race-justification for $E = E_n$. We will build a justifying sequence $\{C'_k\}_{k=0}^{2n+1}, \{E'_k\}_{k=0}^{2n+1}$ in the sense of Lemma 5.1 in the following way:

$$\begin{array}{ll} C'_0 = \emptyset & E'_0 = E_0 \\ C'_{2i-1} = \{w \mid w \in C_i \text{ is a write}\} \cup C_{i-1} & E'_{2i-1} = E_{i-1} \\ C'_{2i} = C_i & E'_{2i} = E_{i-1} \\ C'_{2n+1} = C_n & E'_{2n+1} = E_n \end{array}$$

where $i \in \{1, \dots, n\}$. It is straightforward to check that the sequence satisfies the required properties.

“ \Leftarrow ”: let $\{C_i\}_{i=0}^n, \{E_i\}_{i=0}^n$ be the justification in the sense of Definition 5.8. We define

$$C'_i = \bigcup_{\substack{r \in C_i \text{ is a read} \\ W_i(r) \not\leq_{hb_i} r}} \{r, W_i(r)\} \quad E'_i = \begin{cases} E_{i+1} & \text{if } i < n \\ E & \text{if } i = n \end{cases}$$

for all $i \in \{0, \dots, n\}$. Then $\{C'_i\}_{i=0}^n, \{E'_i\}_{i=0}^n$ is a data-race-justification for E . \square

5.2.3 Example

An example should help make the operational interpretation clearer. First, we demonstrate the committing semantics on program A in Figure 5.1. In the well-behaved execution of this program, illustrated by the first diagram in Figure 5.2, the reads of x and y can only see the default writes of 0, because there is no synchronisation. This results in $r1 = r2 = 0$.

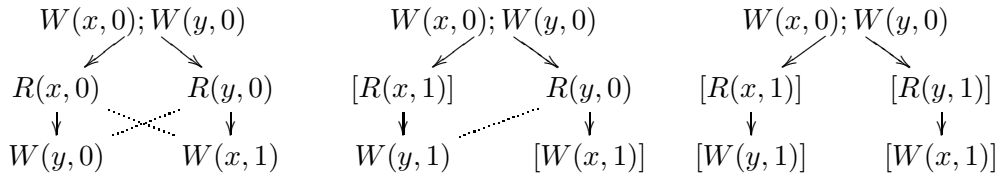


Figure 5.2: Justifying executions of program A from Figure 5.1.

There are two data races in this execution (depicted by the dotted lines, the solid lines represent the happens-before order)—one on x and one on y . We can commit either one of the races or both of them. Suppose we commit the race on x . In the second diagram we show the only restarted execution that uses this data race; the committed actions are in brackets and the committed read sees the value of (the write in) the data race. The non-committed read sees the write that happens-before it, i.e., the default write. This execution gives the result $r1 = 1$ and $r2 = 0$. The JMM can again decide to commit a data race from the execution. There is only one such data race. Committing the data race on y gives the last diagram, and results in $r1 = r2 = 1$.

Observe that the result $r1 = r2 = 1$ is not possible for program C in Figure 5.1, because we can only commit data races with value 0.

5.3 Invalid Transformations

In this section, we show and explain our counterexamples for the invalid transformations. All invalidity arguments apply to the finite version⁵ of the Java Memory Model as described in Gosling et al. (2005b) and to our alternative weaker memory model proposed in Aspinall and Ševčík (2007). The examples follow the same pattern—at first we list a program where a certain behaviour is not possible in the JMM, and then we show that after the transformation the behaviour becomes possible (in the JMM). This shows that the transformation in question is invalid, because any run of the transformed program should be indistinguishable from some run of the original program. In the Java Memory Model, the behaviour of a program is essentially the set of external actions, such as printing, performed by the program⁶. In our examples, we will consider final contents of registers being

⁵We use the finite version because the infinite JMM contains subtle inconsistencies (Aspinall and Ševčík, 2007).

⁶The definition in Manson et al. (2005) is slightly more complex because of non-terminating executions and ordering, see Definition 5.11 for details. Our examples are always terminating.

part of the program's behaviour, because we could observe them by printing them at the end of each thread.

5.3.1 Redundant Write after Read Elimination

initially $x = 0$		
lock m1 x=2 unlock m1	lock m2 x=1 unlock m2	lock m1 lock m2 r1=x x=r1 r2=x unlock m2 unlock m1

First note that no well-behaved execution of this program contains a read-write data race, so all legal executions of this program are well-behaved. Moreover, in all executions the read $r2=x$ must see the write $x=r1$, because it overwrites any other write. As the write $x=r1$ always writes the value that is read by $r1=x$, we have that $r1 = r2$.

On the other hand, if a compiler removes the redundant write $x=r1$, the reads $r1=x$ and $r2=x$ can see different values in a well-behaved execution, e.g., we might get the outcome $r1 = 1$ and $r2 = 2$.

Note that the program contains a write-write data race between the first two threads, so the unexpected behaviour is not prohibited by the DRF guarantee of the Java Memory Model.

5.3.2 Redundant Read after Read Elimination

The counterexample for the elimination of a read after a read uses a trick with switching the branches of an `if` statement—in the first well-behaved execution we take one branch, and then we commit a data race so that we can take the other branch after we restart. Let us examine the program below.

$x = y = 0$	
r1=x y=r1	r2=y if (r2==1) { r3=y x=r3 } else x=1

The question is whether we can observe the result $r2 = 1$. This result is not possible in this program, but it becomes possible after rewriting $r3=y$ to $r3=r2$.

First we show that this is not possible with the original program: With the initially empty commit set we can get just one well-behaved execution—the one where $r1 = r2 = 0$. In this well-behaved execution, we have two data races: (i) between the actions performed by $y=r1$ and $r2=y$ with value 0, (ii) between the actions performed by $r1=x$ and $x=1$ with value 1. If we commit (i), we are stuck with $r2 = 0$, because all subsequent restarted executions must perform the committed read of y with the value 0. If we commit (ii) and restart, we get an execution, where $r1 = 1$, so we can now commit the data race between $y=r1$ and $r2=y$ with value 1. After we restart the execution, suppose we were to read $r1 = r2 = 1$. Then $r3=y$ must read a value that happens-before it; the only such value is the default value 0, but then $x=r3$ must write 0, which contradicts the commitment to perform the write of 1 to x .

On the other hand, if JVM transforms the read $r3=y$ into $r3=r2$, we can obtain the result $r2 = 1$ by committing the data race between $r1=x$ and $x=1$, restarting, committing the data race between $y=r1$ and $r2=y$, and restarting again. As opposed to the original program, now we can keep the commitment to write 1 to x , because $r3 = r2 = 1$ in the transformed program.

5.3.3 Roach Motel Semantics

We demonstrate the invalidity of roach motel semantics on the program:

initially $x = y = z = 0$			
lock m x=2 unlock m	lock m x=1 unlock m	r1=x lock m r2=z if (r1==2) y=1 else y=r2 unlock m	r3= y z=r3

This program cannot result in $r1 = r2 = r3 = 1$ in the JMM: In all well-behaved executions of this program, we have $r1 = r2 = r3 = 0$, and four data races—two on x with values 1 and 2, then on y and z with value 0. If we commit the data race on y (resp. z , resp. x with value 2) we would be stuck with $r3 = 0$ (resp. $r2 = 0$, resp. $r1 = 2$), so we can only commit a race on x . However, if we commit the race with $x=1$ and restart, we are only left with races on z and y with value 0. Committing any of these races would result in $r2$ and $r3$ being 0.

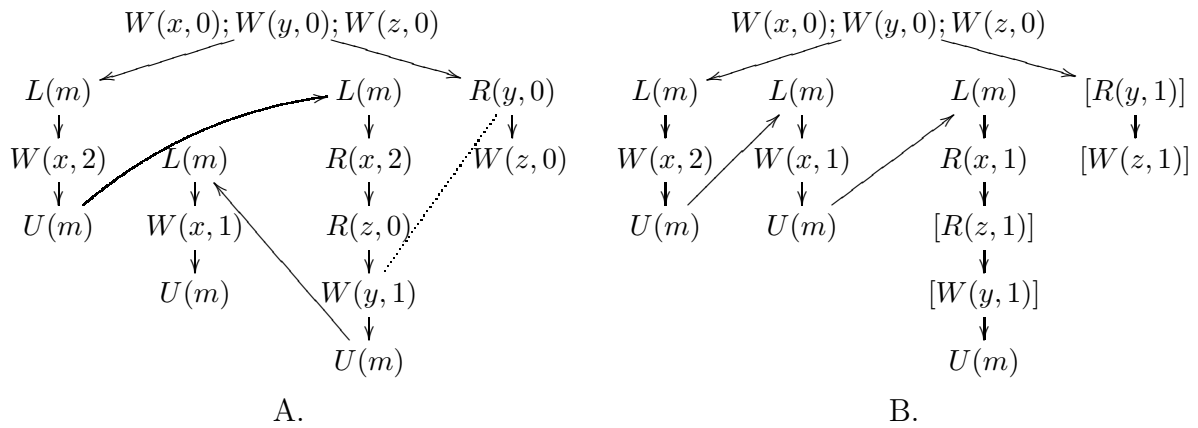


Figure 5.3: Justifying and final executions for the roach motel semantics counterexample.

However, after swapping $r1=x$ and $lock\ m$ the program offers more freedom to well-behaved executions, e.g., the read $r1=x$ can see value 2 (without committing any action on $x!$), and we can commit the data race on y with value 1 (see execution A from Figure 5.3). After restarting, we can commit data race on z with value 1. After another restart, we change the synchronisation order so that the write $x=1$ overwrites the write $x=2$, and the read $r1=x$ sees value 1 (see execution B from Figure 5.3). In this execution, we have $r1 = r2 = r3 = 1$.

Note that this committing sequence respects the rule that all the subsequent restarted executions must preserve synchronisation that was used to justify the previous data races, because our committing sequence only introduces new synchronisation that in effect overwrites the write $x=2$ with the write $x=1$. This problem seems to be hard to solve in a committing semantics based on well-behaved executions, because more synchronisation gives more freedom to well-behaved executions and allows more actions to be committed.

5.3.4 Irrelevant Read Introduction

The counterexample for irrelevant read introduction uses the trick with switching branches again. The program

x = y = z = 0	
<pre> r1 = z if (r1==0) { r3 = x if (r3==1) y = 1 } else { //r4 = x r4 = 1 y = r1 } </pre>	<pre> x = 1 r2 = y z = r2 </pre>

cannot result in $r1 = r2 = 1$: its only well-formed execution has data races on x with value 1 and z with value 0. We cannot commit the data race on z , because then $r1$ would remain 0. If we commit the data race on x and restart, we have a new data race between $y=1$ and $r2=y$. After committing it and restarting, we can try to commit the data race on z with value 1. However, after this commit and restart, we cannot fulfil the commitment to perform the data race on x .

On the other hand, if we introduce the irrelevant read $r4=x$ by uncommenting the commented-out line, we can keep the commitment to perform the committed read on x , and the program can result in $r1 = r2 = 1$. This seems to be another deep problem with committing semantics—even introducing a benign irrelevant read may validate some committing sequence that was previously invalid.

5.3.5 Reordering with external actions

The program

x = y = 0	
<pre> r1=y if (r1==1) x=1 else {print "!"; x=1} </pre>	<pre> r2=x y=r2 </pre>

cannot result in $r1 = r2 = 1$ in the JMM, because to have $r2 = 1$ we must commit the data race on x and, by the rule for committing external actions, also the external printing action. To get $r1 = 1$ we must also commit the race on y , but then we are not able to keep the commitment to perform the committed printing action.

However, if we swap `print "!"` with `x=1` in the else-branch, the rule for external actions does not apply, and we can commit the race on x , and then the race on y , resulting in $r1 = r2 = 1$.

5.4 Valid Transformations

In this section we prove the validity of irrelevant read elimination, read after write elimination, write after write elimination, and reordering of independent non-volatile memory accesses in the weaker version of the Java memory model. Using the same method one could also prove that the first three of these transformations are valid in the standard JMM Gosling et al. (2005b).

The validity of a transformation says that any behaviour of the transformed program is a behaviour of the original program. We prove it using the same direct approach as in the proofs of safety in Chapter 3—we take an execution of the transformed program that exhibits the behaviour in question, then we apply an ‘inverse’ transformation to the execution, and finally we show that the untransformed execution has the same behaviour as the one of the transformed program. Note that our proof technique does not consider non-termination being a behaviour; we only prove safety of transformations with respect to external output behaviours. We leave the preservation of termination for future work.

The main idea of the proof is that we describe transformations using their ‘inverse’ transformations. We will say that P' is a transformation of P if for any trace $t' \in P'$ there is an *untransformation* in P . By the untransformation we mean a trace t of P together with an injective function f that describes a valid reordering of the actions of t' . Moreover, each action of t that is not in $\text{rng}(f)$ must be either

1. a redundant read after write, i.e., it must be a read of the same value as the last write to the same variable in the trace, and there cannot be any synchronisation or read from the same variable in between, or
2. a redundant write before write, i.e., the write must precede another write to the same variable such that there is no read from the same location or synchronisation in between, or
3. an irrelevant read, i.e., the value of the read cannot affect validity of the trace t in P .

For formal details, see Definition 5.12. Using the techniques described in Chapter 4, we can show that the program transformations on the syntax level implies the existence of an untransformed trace and an untransformation function for each trace of the transformed program.

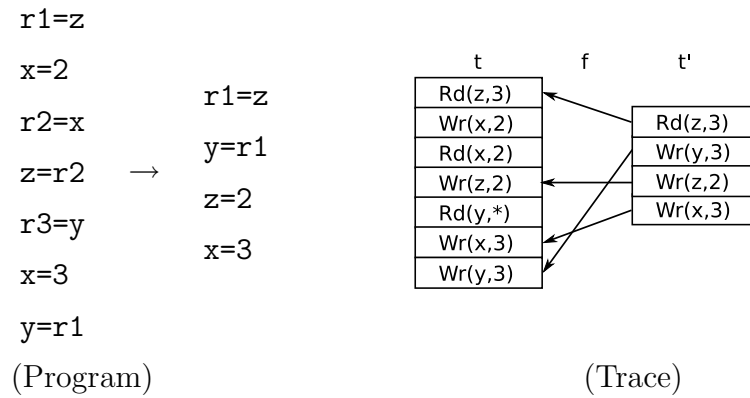


Figure 5.4: Transformation of a program as a transformation on traces.

For example, the program on the left in Figure 5.4 can be transformed to the program on the right of the arrow, because for each trace of the transformed program there is its untransformation. For instance, for the trace t' (on the right of Figure 5.4) of the transformed program there is a trace t of the original program, and a function f that determines the reordering of the actions. Moreover, $Wr(x, 2)$ is a redundant write before write, $Rd(x, 2)$ is a redundant read after write, and $Rd(y, *)$ is an irrelevant read, i.e., t is a valid trace of P if we replace $*$ by any value.

Having this definition, the proof is technical, but straightforward—given an execution of the transformed program we construct an execution of the original program by untransforming the traces of all its threads, while preserving the synchronisation order. This is possible because the definition of program transformation preserves ordering of synchronisation actions, thus guaranteeing consistency of the program order with the synchronisation order.

We also observe that the untransformed execution is legal: if we take the committing sequence of data races and justifying executions, and untransform the justifying executions, we get a legal committing sequence for the untransformed program (Lemma 5.4). We conclude that any behaviour of the transformed program is a behaviour of the original program (Theorem 5.2).

5.4.1 Proof of Validity

In this subsection, we describe the proof of the validity of transformations in greater detail. For formal description of effects of transformations, we need to define the notion of observable behaviours of a program:

Definition 5.10. An execution $\langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ with happens-before order \leq_{hb} has a set of *observable behaviours* O if for all $x \in O$ we have $y \leq_{hb} x$ or $y \leq_{so} x$ implies $y \in O$ or $T(y) = \theta_{init}$. Moreover, there is no $x \in O$ such that $T(x) = \theta_{init}$.

The allowable behaviours may contain a special external *hang* action if the execution does not terminate. We will use the notation $\mathcal{X}(A)$ for the set of all external actions from set A , i.e., $\mathcal{X}(A) = \{a \mid \exists v. K(a) = \text{Ext}(v)\}$.

Definition 5.11. A finite set of actions B is an *allowable behaviour* of a program P if either

- There is a legal execution E of P with a set of observable behaviours O such that $B = \mathcal{X}(O)$, or $B = \mathcal{X}(O) \cup \{\text{hang}\}$ and E is hung.
- There is a set O such that $B = \mathcal{X}(O) \cup \{\text{hang}\}$, and for all $n \geq |O|$ there must be a legal execution E of P with set of actions A , and a set of actions O' such that (i) O and O' are observable behaviours of E , (ii) $O \subseteq O' \subseteq A$, (iii) $n \leq |O'|$, and (iv) $\mathcal{X}(O') = \mathcal{X}(O)$.

5.4.1.1 Effects of Transformations on Traces.

First, we define the notion of transformed program loosely enough so that redundant read/write elimination, irrelevant read elimination and reordering fit our definition. The idea is that for any trace of the transformed program there should be a trace of the original program that is just reordered with the redundant and irrelevant operations added.

To describe the effects of irrelevant read elimination formally we define *wildcard traces*⁷ that may contain star $*$ symbols instead of some values. For example, the sequence $[\langle \text{Wr}(x), 2 \rangle, \langle \text{Rd}(y), * \rangle, \langle \text{Rd}(x), 3 \rangle]$ is a wildcard trace. If \hat{t} is a wildcard trace, then $[\hat{t}]_*$ stands for the family of all (normal) traces with the $*$ symbols replaced by some values.

Given a wildcard trace \hat{t} , we say its i^{th} component $\hat{t}_i = \langle a, v \rangle$ is an

- *irrelevant read* if a is a read and v is the wildcard symbol $*$,

⁷We used the same idea for the definition of transformations for our interleaved semantics in Section 3.1.

- *redundant read* if a is a read of some x and the most recent access of x is a write of the same value, and there is no synchronisation or external action in between; formally, there must be $j < i$ such that $\hat{t}_j = \langle \text{Wr}(x), v \rangle$ and for each k such that $j < k < i$ it must be that $\hat{t}_k = \langle \text{Wr}(y), v' \rangle$ or $\hat{t}_k = \langle \text{Rd}(y), v' \rangle$ for some $y \neq x$ and some v' ,
- *redundant write* if a is a write to some x and one of these two cases holds: (i) the write is overwritten by a subsequent write to the same variable and there are no synchronisation or external actions, and no read of x in between, or (ii) \hat{t}_i is the last access of the variable in the trace and there are no synchronisation or external actions in the rest of the trace.

Definition 5.12. We will say that P' is a *transformed traceset* from P if for any trace t' in P' there is a wildcard trace \hat{t} and a function $f :: \{0, \dots, |t'| - 1\} \rightarrow \{0, \dots, |\hat{t}| - 1\}$ such that:

1. all traces in $[\hat{t}]_*$ are sequentially valid in P .
2. function f is injective,
3. the action kind-value pair t'_i is equal to $\hat{t}_{f(i)}$,
4. for $0 \leq i \leq j < |t'|$ we have that $f(i) \leq f(j)$ if any of the following reordering restrictions holds:
 - (a) t'_i or t'_j is a synchronisation or external actions, or
 - (b) t'_i and t'_j are conflicting memory accesses, i.e., accesses to the same variables such that at least one is a write,
5. if there is an index $j < |\hat{t}|$ such that $f(i) \neq j$ for any i , then \hat{t}_j must be a redundant read, a redundant write, or an irrelevant read.

The link between this semantic definition and concrete syntactic transformations can be established using the techniques from Chapter 4.

5.4.1.2 Transforming Executions.

Let P' be a traceset transformed from P , and $E' = \langle A', P', \leq'_{po}, \leq'_{so}, W', V' \rangle$ be a legal execution of P' . Our goal is to construct a legal execution E of P with the same observable behaviours.

The main idea of the construction is to take the memory trace of each thread in E' and use Definition 5.12 to obtain a corresponding trace from P , and a mapping of the actions and the program order of E' to the actions and the program order of our newly constructed execution. We will also need to restore all the actions that were eliminated by the transformation and construct the visibility functions W and V for the reconstructed actions.

Given an execution $E' = \langle A', P', \leq'_{po}, \leq'_{so}, W', V' \rangle$ we construct *untransformed execution* E of P : for each thread $\theta \neq \theta_{init}$ let $\text{Tr}_{E'}(\theta)$ be the trace of θ in E' . By the definition of transformed program (Definition 5.12), there must be a wildcard trace of P , let us denote it by \hat{t}^θ , and a corresponding transformation function f_θ .

For the initialisation thread θ_{init} we define

$$\hat{t}^{\theta_{init}} = [\langle S(\theta_{init}), 0 \rangle] \# \text{Tr}_{E'}(\theta_{init})|_W \# \text{Init}_E \# [\langle \text{Fin}, 0 \rangle],$$

where $\text{Tr}_{E'}(\theta_{init})|_W$ is the trace of the initialisation thread of E' restricted to (possibly volatile) write actions, and Init_E is any sequence of initialisation writes for all variables that appear in any component of \hat{t}^θ ($\theta \neq \theta_{init}$), but are not initialised in E' . We set $f_{\theta_{init}}(i) = i$ if $0 \leq i < |\text{Tr}_{E'}(\theta_{init})| - 1$, and $f_{\theta_{init}}(|\text{Tr}_{E'}(\theta_{init})| - 1) = |\hat{t}^{\theta_{init}}| - 1$.

From the traces \hat{t}^θ we build action traces t^θ of the same length. For $0 \leq i < |\hat{t}^\theta|$, we set the i -th component of t^θ to be

- $f_\theta^{-1}(i)$ -th element of $\text{Tr}_{E'}(\theta)$ if $f_\theta^{-1}(i)$ exists, or
- fresh action a such that $K(a) = \hat{t}_i^\theta$ and $T(a) = \theta$, if there is no j such that $i = f_\theta(j)$.

We construct our *untransformed execution* $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ from the action traces t^θ :

1. $A = \{t_i^\theta \mid 0 \leq i < |\hat{t}^\theta|\}$,
2. the order \leq_{po} is the order induced by the traces t^θ , i.e.,

$$\leq_{po} = \{(a, b) \mid T(a) = T(b) \wedge \iota(t^{T(a)}, a) \leq \iota(t^{T(a)}, b)\}$$

3. the order \leq_{so} is equal to \leq'_{so} ,
4. the write-seen function $W(a)$ is

- $W'(a)$ if $a \in A'$,
- most recent write⁸ to x in \leq_{hb} if $a \notin A'$ and a is a read from x ,
- a otherwise, i.e., if a is not a read,

5. $V(a)$ is the corresponding value in the wildcard trace \hat{t}^θ , i.e., $V(a) = \mathcal{V}(\hat{t}_{\iota(t^{T(a)}, a)}^{T(a)})$.

Lemma 5.2. *Let P' be a transformation of P , E' be a well-formed execution of P' with happens-before order \leq'_{hb} and E be the untransformed execution of P with happens-before order \leq_{hb} . Let x and y be two actions from E' such that any of them is synchronisation action, or they are conflicting memory accesses⁹, or $T(x) \neq T(y)$. Then $x \leq_{hb} y$ if and only if $x \leq'_{hb} y$.*

Proof. Observe that by point 4 of Definition 5.12 we have $x \leq_{po} y$ iff $x \leq'_{po} y$ for all x and y from E' such that x or y is a synchronisation or external action, or x and y are conflicting memory accesses. By induction on the transitive closure definition of \leq_{hb} we get that for any $z \leq_{hb} y$ either $z \leq_{po} y$ or there is a synchronisation action s such that $z \leq_{po} s \leq'_{hb} y$. With the observation above we conclude that $x \leq_{hb} y$ implies $x \leq'_{hb} y$ if x is in E' and x or y is a synchronisation action, or x and y are conflicting memory accesses, or $T(a) \neq T(b)$. On the other hand, we prove that $z \leq'_{hb} x$ implies that either $z \leq'_{po} x$ or there is a synchronisation action s such that $z \leq'_{po} s \leq_{hb} x$ by induction on the definition of \leq'_{hb} . This implies the other direction of the equivalence. □

Lemma 5.3. *Let P' be a transformation of P , E' be a well-formed execution of P' and E be the untransformed execution of P . Then E is a well-formed execution of P .*

Proof. Properties 1–9 and 11 of well-formedness (Definition 5.7) are satisfied directly by our construction. We prove property 10, the hb-consistency, i.e., that for all reads in E , $r \not\leq_{hb} W(r)$ and there is no write w to the same variable as $W(r)$ such that $W(r) <_{hb} w \leq_{hb} r$. There are two cases: (i) for r being an irrelevant read or a redundant read the hb-consistency is satisfied trivially by construction, (ii) for $r \in E'$, we get the result using hb-consistency of E' and Lemma 5.2. □

⁸Note that the initialisation writes in thread θ_{init} happen-before any read action, so a most recent write always exists.

⁹I.e. a read and a write to the same variable, or two writes to the same variable.

Lemma 5.4. *Let P' be a transformation of P , E' be a legal execution of P' and E be the untransformed execution of P . Then E is a legal execution of P .*

Proof. Let $\{C_i\}_{i=0}^n$ be a sequence of committing sets and $\{E'_i\}_{i=0}^n$ the corresponding justifying executions for E' . Let E_i be the untransformed executions of E'_i . Let us define C_{n+1} as the set of actions of E and $E_{n+1} = E$. Then it is straightforward to show that the committing sequence $\{C_i\}_{i=0}^{n+1}$ with the justifying executions $\{E_i\}_{i=0}^{n+1}$ satisfies the conditions (1), (3), (4) and (6) of Definition 5.8. To establish the rules (2), (5) and (7) we use Lemma 5.2 and legality of E' . □

In the following we will write $C_{\leq_{so}, \leq_{po}}(X)$ to denote the \leq_{so} and \leq_{po} downward closure of X without the initialisation actions, i.e.,

$$C_{\leq_{so}, \leq_{po}}(X) = \{y \mid \exists x \in X. y \leq_{po \cup so} x \wedge T(y) \neq \theta_{init}\},$$

where $\leq_{po \cup so} = (\leq_{po} \cup \leq_{so})^+$. We will often use $C_E(X)$ for $C_{\leq_{so}, \leq_{hb}}(X)$, where E has the synchronisation order \leq_{so} and the happens-before order \leq_{hb} . The set $C_E(X)$ is an observable behaviour of execution E with actions A for any $X \subseteq A$.

Lemma 5.5. *Let P' be a transformation of P , E' be a legal execution of P' with observable behaviour O' , and E be the untransformed execution of P . Then $\mathcal{X}(C_E(O')) = \mathcal{X}(O')$.*

Proof. The direction \supseteq is trivial, because $\mathcal{X}(-)$ is monotone and $C_E(O') \supseteq O'$.

On the other hand, if an external action $x \leq_{po \cup so} y \in O'$, then for any $z \leq_{po \cup so} y$ there is s such that $z \leq_{po} s \leq'_{po \cup so} y$ by induction on the transitive definition of $\leq_{po \cup so}$. By Lemma 5.2 we get $x \leq'_{po \cup so} y$, thus $x \in O'$. □

The main theorem says that transforming a program using Definition 5.12 cannot introduce any new behaviour, except hanging.

Theorem 5.2. *Let P' be a program transformed from P . If B is an allowable behaviour of P' then $B \setminus \{hang\}$ is an allowable behaviour of P .*

Proof. By Definition 5.11, there must an execution E' of P' with observable behaviour O' such that $B = \mathcal{X}(O')$ or $B = \mathcal{X}(O') \cup \{hang\}$.

Let us take an untransformation E of E' and let $O = C_E(O')$. Using Lemma 5.5, we have $\mathcal{X}(O) = \mathcal{X}(O')$. Since O is an observable behaviour of E and E is a legal

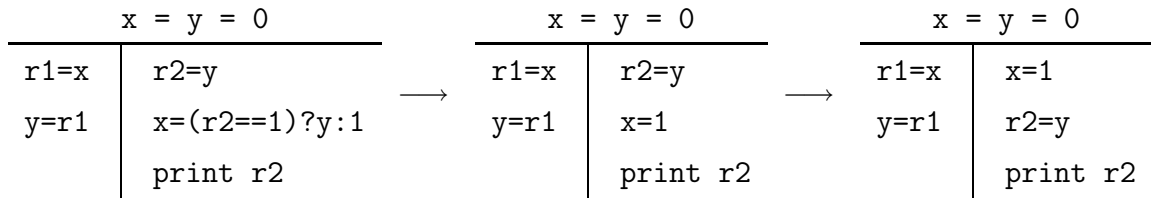


Figure 5.5: Hotspot JVM's transformations violating the JMM.

execution of P (Lemma 5.4), the set $B \setminus \{hang\} = \mathcal{X}(O)$ is an allowable behaviour of P .

□

5.5 Practical Impact

The flaw in the memory model is important in theory, but it is conceivable that it might not be manifested in practical implementations, because JVMs compile to stricter memory models than the JMM. It is natural to ask whether some widely used JVM actually implements optimisations that lead to forbidden behaviours. In fact, this is indeed the case. We have experimented with the Sun Hotspot JVM (Paleczny et al., 2001) to discover this.

For example, the first program in Figure 5.5 cannot print 1 under the JMM (for details, see the counterexample for redundant read after read elimination in Section 5.3). A typical optimising compiler may reuse the value of y in $r2$ and transform

$$x=(r2==1)?y:1 \quad \rightarrow \quad x=(r2==1)?r2:1$$

which is equivalent to the second program from Figure 5.5. Then it may reorder the write to x with read of y , yielding the last program in Figure 5.5. Observe that this transformed program can print 1 using the interleaving $x=1, r1=x, y=r1, r2=y, \text{print } r2$. After minor modifications to the program, Sun Hotspot JVMs will perform these transformations, so it does not comply with the JMM¹⁰.

The program in Figure 5.5 is not data-race-free. Should we worry about behaviours of correctly synchronised programs after optimisations? Although we do not have a formal proof, we believe that the transformations performed by

¹⁰Tested on Java HotSpot(TM) Tiered VM (build 1.7.0-ea-fastdebug-b16-fastdebug, mixed mode), Linux x86. Further details are in a short technical report (Ševčík, 2008).

the Hotspot JVM can be explained by the framework described in Chapters 2 and 3. This means that Sun's implementation of Java might be in fact correct, i.e., satisfy the DRF guarantee, and it is only the JMM specification that needs fixing.

Chapter 6

Conclusion

This chapter discusses possible applications of our work, relates our results to the existing research on weak memory models and gives an overview of interesting open questions and future research.

6.1 Towards Applications

Our results can be used in two ways. For languages that enforce data race freedom and guarantee sequential consistency, such as Guava (Bacon et al., 2000) or Ada (Ada95, 1995), or languages with undefined behaviours for programs with data races, e.g., C++0x (Boehm and Adve, 2008), we have established that a broad class of transformations can be safely performed by their compilers. We compare this result to the existing work on compilers and hardware in Sections 6.2.1 and 6.2.4.

On the other hand, languages that allow data races, but still need some safety guarantees, such as Java (Gosling et al., 2005a) or C# (ECMA, 2006a,b), could use our semantic transformations as a basis for their memory models. Although modern processors (Intel, 2002, 2007; Sparc International, 2000) use memory model specifications that are similar to ours, these specifications are not usable by compilers for several reasons. First, processor transformations are defined as relaxations of orders on actions. Such definitions are not composable: in contrast to compilers, it does not even make sense to talk about performing optimisations in compilation stages. Second, the processor transformations only describe changing the order of actions, but they do not explain any action eliminations. Finally, the processor memory models are tied to instructions at least in the sense that

$$\begin{array}{c}
 r := x; \text{ if } (r \neq 0) \text{ y} := 1 \text{ else } \text{y} := 1 \\
 \downarrow \\
 \text{y} := 1; r := x
 \end{array}$$

Figure 6.1: Reordering two instructions to one.

they describe dependencies using data and control flow dependencies of their instructions. This is unacceptable for compilers, because they use different program representations through different compilation stages. Moreover, they can remove or introduce dependencies between instructions, or even join or split instruction instances. Our framework addresses all these issues: we prove compositionality of our transformations, we allow action eliminations, and our actions are not linked to instructions throughout the transformation in any way.

To illustrate the last point, consider the reordering transformation from Figure 6.1. In hardware memory models, the instruction $\text{y} := 1$ clearly depends on the read of x , so it cannot be reordered with the read. However, our definition of reordering allows such a transformation. In fact, the program

$$r := x; \text{ if } (r \neq 0) \text{ y} := 1 \text{ else } \text{y} := 1$$

has the same traceset as the program $r := x; \text{y} := 1$, which can be obviously reordered to $\text{y} := 1; r := x$.

6.2 Related Work

6.2.1 Compilers

The existing research on compiler optimisations for shared-memory concurrency concentrates mainly on maintaining sequential consistency for all programs, starting with the foundational work of Shasha and Snir (1988). In their paper, they take a sequentially consistent execution of a straight line program and describe a set of reordering constraints that preserve sequential consistency. Based on this analysis, they suggest placing memory barriers to prevent reordering that might violate sequential consistency. Using this, Sura et al. (2005) developed whole program analyses that determine allowable reorderings in multi-threaded program while preserving behaviours of all programs. The emphasis of that work is different from ours: their work does not analyse existing optimisations for sequential

programs. Instead, they design a restricted compiler that guarantees (an illusion of) sequential consistency for all programs. In contrast, we show that many existing program transformations maintain an illusion of sequential consistency, if the programs are correctly synchronised. We are not aware of any such work in the context of compilers, although this problem has been well studied in processor memory model literature. We will compare our work to hardware optimisations in Section 6.2.4.

Microsoft.NET MM. We feel that we should comment on the .NET memory model and compare it to the JMM, but the situation there seems to be confusing. The ECMA standard of .NET (ECMA, 2006b) contains a very weak informal memory model that prohibits any optimisations on volatile locations, but it allows roach motel reordering and any optimisations on non-volatile locations that are valid for sequential programs (Morrison, 2005). Such a specification does not satisfy the DRF guarantee. To see this, consider the program

```
Thread 1: r1 := x; if (r1 == 1) y := 1; print r1
Thread 2: r2 := y; if (r2 == 1) x := 1
```

where the default values for `x` and `y` are 0. This program cannot print 1 in any interleaving and it does not contain data races, because no interleaving of the program performs a write to `x` or `y`. However, an optimising compiler might speculate on the value of `y` in the second thread and rewrite to

```
Thread 1: r1 := x; if (r1 == 1) y := 1; print r1
Thread 2: r3 := x;
         x := 1;
         r2 := y;
         if (r2 != 1) x := r3
```

This program can clearly print 1 by executing the first two statements of thread 2 followed by all the statements of thread 1. This breaks the illusion of sequential consistency even for data race free programs. We believe that such a weak memory model is not acceptable for most programmers. Interestingly, our experiments with the Microsoft.NET virtual machine suggest that the current implementation does not take advantage of the weak memory model as it does not perform even the simplest optimisations of shared-memory accesses, such as the elimination of redundant and irrelevant reads.

6.2.2 Other Models of Concurrency

Although shared memory concurrency with monitors is a prevalent model of parallel computing for multiprocessors today, there are other models and some of them avoid the problems described in this theses.

Transactional memory. Transactional memory (Shavit and Touitou, 1995; Herlihy and Moss, 1993) simplifies the model with locks and monitors by letting the programmer specify regions of code that should appear to execute atomically. To achieve high performance, many transactional memory implementations relax the semantics for statements outside atomic regions so that these statements can see values from restarted transaction or other artifacts of transactional executions (Moore and Grossman, 2008; Abadi et al., 2008). The semantics of such weak implementations of transactions seem to be quite similar to the semantics of weak memory models. To our knowledge, there is no study of the effects of program transformations on transactional programs and it would be interesting to use our techniques in that context.

Message passing. In pure message passing concurrency, there is no shared memory and the only means of communication is sending messages to other processes. Such approach avoids the problems with memory consistency, but it imposes communication overhead on accessing shared data. The typical representative of message passing concurrent language is Erlang (Armstrong et al., 1996), which uses the Mnesia database management system to store shared data (Mattsson et al., 1998). Although Mnesia can access data transactionally, it also provides unsafe access to the database, which is implemented on top of a native library using standard shared memory. The semantics of the unsafe access is not defined precisely, and we suspect that a precise specification would run into the same problems as the weak transactional memory described in Abadi et al. (2008).

6.2.3 Semantics of Concurrency

Trace semantics is a traditional way to describe concurrent systems. The traditional trace theory describes algebraic properties of traces of abstract events equipped with a congruence induced by equations of the form $ab = ba$ (Mazurkiewicz, 1986). While this resembles our framework for reordering interleavings, there are fundamental differences. First of all, our reordering is not a congruence; for ex-

ample, roach motel reordering only allows reordering in one direction. Even if we ignore roach motel reordering, it is not clear how to represent intra-thread dependencies, such as control and data dependencies.

Brookes (1993) defines a fully abstract trace semantics for a concurrent imperative language. He applies the semantics to proving validity of simple transformations in any context. This contrasts to our work, where we consider only the data race free context and prove a much larger class of transformations to be safe. Instead of traces of events, he uses traces of global states. However, our reordering transformations make any reasoning about global state difficult and such trace semantics does not seem to be applicable to reordering. Brookes (2007) uses a semantics close to ours to prove soundness of concurrent separation logic. Although this work is not primarily concerned with program transformations, recent research suggests that the information from separation logic annotations could be used to prove correctness of optimisations (Hobor et al., 2008), because the annotations determine an interference-free memory footprint for computations inside critical sections. A compiler can then perform any sequentially valid transformation as long as the transformed program still accesses only the footprint.

The theory of memory models by Saraswat et al. (2007) builds on a similar idea. The building blocks of the theory are partial functions from partial stores to partial stores. These partial functions are denotations of a synchronisation-free piece of code. Parallel shared-memory computation is represented as a directed acyclic graph, where its nodes are the denotations and the edges are synchronisation or program order links. They define several classes of transformations on these graphs, such as joining and splitting the nodes, and show that the transformations satisfy the DRF guarantee. The theory uses a restricted language that does not contain looping constructs and general branches. They do not show how to map syntactic transformations to their denotational transformations.

A different theoretical approach employs event structures to formalise concurrency. Event structure theory (Winskel, 1986) is an algebraic framework for describing concurrent computations as possible configurations of abstract events. Event structures have been applied to describe the Java Memory Model using a combination of axiomatic, denotational and operational techniques (Cenciarelli et al., 1999, 2007). However, this work does not attempt to establish validity of any transformations and it seems that some trivial program transformations,

such as the elimination of redundant read after read, are not generally valid in such a framework.

6.2.4 Hardware Memory Models

The computer architecture community has studied correctness of processor optimisations for parallel programs for a long time. The excellent survey by Adve and Gharachorloo (1996) summarises the main developments of hardware memory models.

Our safety criterion, the DRF guarantee, was invented independently by Adve and Hill (1990) and Gharachorloo et al. (1990) in the hardware context. Building on the work of Shasha and Snir (1988), they show that some reordering of instructions, such as reordering of independent statements and roach motel reordering, satisfies the DRF guarantee. To describe data race freedom, they introduced the concept of release-acquire pairs and named the memory model *release consistency*. In their later work, they relax the model even more to allow for different orders of visibility on different processors, and prove the DRF guarantee in this setting (Adve and Aggarwal, 1993; Adve and Hill, 1992). Although the basic approach is very similar to ours, there are some notable differences: in particular, release consistency does not explain eliminations, the actions in hardware are tied to instructions, and it is not a compositional memory model, as discussed in Section 6.1.

In practice, current processors use a slightly different model. Instead of implementing the high-level synchronisation primitives, such as locks and volatile locations, they describe allowed reorderings and provide low-level memory barrier instructions that prevent certain kinds of reordering. For example, Sun's Total Store Order only allows write buffering, i.e., writes to memory can be delayed until the next memory barrier or the next write to the same location on the same processor (Sparc International, 2000). The memory model of an Intel 64 processor prohibits similar reorderings as the TSO model, but it allows each processor to see a different order of writes on different locations (Intel, 2007; Sarkar et al., 2009). Intel Itanium implements a more complex memory model that is based on release consistency (Intel, 2002). To enforce ordering on volatile accesses and monitor actions in compiled programs, compilers must insert memory barriers, locked instruction modifiers (Intel 64) and release/acquire instruction modifiers

(Intel Itanium). For details, see Lea (2008).

It is interesting that some computer architecture researchers suggest that hardware should guarantee sequential consistency (Hill, 1998) arguing that recent research on speculation techniques can maintain the illusion of interleaved semantics (Gniady et al., 1999).

We believe that our framework can explain some stronger hardware memory models, such as TSO, because moving writes later is always safe transformation, provided that the write does not pass a release action or a memory access to the same location. It is an interesting research question whether our transformations can cover the more relaxed memory models.

6.3 Future Work and Open Questions

Our investigation still leaves many interesting questions open. We have only looked at thread-local elimination and reordering transformations. In Subsection 6.3.1 we discuss how other classes of transformations, such as read introductions or synchronisation transformations, fit in our framework. In Subsection 6.3.2, we suggest some refinements of the trace semantics to handle dynamic thread creation and object allocation. These features would be necessary to apply our framework to a real language, such as Java. In Subsection 6.3.3, we propose to investigate program transformations with respect to termination properties. In Subsection 6.3.4, we pose interesting research questions regarding formal techniques.

6.3.1 Transformations

Although we believe that our technique is expressive enough to explain most common optimisations in current compilers and virtual machines, there are some safe transformations that our framework does not explain. An overview of some of these transformations follows.

6.3.1.1 Introductions

Our transformations do not allow any kind of introduction of memory accesses. Although it might seem that no optimisation should introduce new memory operations, there are interesting cases where this might be performed in both hardware

and software.

Read Introduction. Perhaps the most interesting introduction is an introduction of an irrelevant read. A simple example of such transformation might be:

$$r1 := 0; \rightarrow r1 := x; r1 := 0;$$

Obviously, no sane compiler performs such an ‘optimisation’. However, irrelevant read introduction can emerge as an artifact of hoisting a read from a while-loop when its body is not executed because its guard evaluates to *false*. For example, consider the program

```
r1 := 1;
...
r3 := 0;
while(r1 == 0) {
    r2 := x;
    r3 = r3 + r2;
}
```

Optimising compilers might avoid the potentially repeated reading of *x* by hoisting the read before the loop:

```
r1 := 1;
...
r3 := 0;
r2 := x;
while(r1 == 0) {
    r3 = r3 + r2;
}
```

If *r1* is not 0 at the beginning of the loop, this optimisation introduces a new read of *x*. Our framework’s inability to explain this transformation might seem to be an important flaw. However, a quick test of the GCC and JVM compilers reveals that they unroll the loop once before hoisting the read. The generated assembly code for the program above is equivalent to

```
r1 := 1;
```

```

...
if (r1 == 0) {
    r2 := x;
    r3 = r2;
    while(r1 == 0) {
        r3 = r3 + r2;
    }
}

```

Note that this program does not introduce any new reads. In terms of semantic transformations, the program is a semantic elimination of the original program.

Observe that the read introduction does not change programs' behaviours, because removing the introduced reads from the execution of the transformed program yields an execution of the original program with the same behaviour. As a result, the read introduction transformation cannot introduce new behaviours by itself, so it trivially satisfies the DRF guarantee. Similarly, the transformation cannot introduce origins of values.

Unfortunately, read introduction can introduce data races. This means that our proofs of safety of elimination and reordering no longer apply if they are preceded by any irrelevant read introduction. It is an interesting open question whether the read introduction is a compositionally safe transformation in the following sense:

Definition 6.1. A traceset T' is a read introduction of traceset T if for any trace t' from T' there is a wildcard trace t^* such that t' is an instance of t^* and the trace t^* with all wildcard reads removed is in T , i.e., $[a \leftarrow t. \forall l. a \neq \text{Rd}(l, *)] \in T$.

Conjecture 1. Let $\{T_i\}_{i=0}^n$ be a sequence of tracesets such that T_0 is data race free and T_{i+1} is an elimination, a reordering or a read introduction of T_i for any $i \in \{0, \dots, n-1\}$. Then any execution of T_n has the same behaviour as some execution of T_0 .

Write Introduction. The only valid write introduction seems to be an introduction of an overwritten write. For example, we can safely perform the following transformation of a code fragment

$$x := 1 \quad \rightarrow \quad x := 1; x := 1$$

Observe that the transformation preserves data race freedom, because we can always remove a write from a ‘racy’ execution of the transformed program to obtain a ‘racy’ execution of the original program. The transformation also cannot introduce a new behaviour, because no read can see the introduced write in an execution without data races.

We suspect that this transformation may be useful for explaining hardware memory models that duplicate writes for each processor; for example, see the Intel Itanium memory model (Intel, 2002).

6.3.1.2 Transforming Synchronisation

Our definitions of elimination and reordering do not allow any changes to synchronisation actions. However, some compilers, e.g., Sun’s Hotspot JVM, can remove synchronisation on objects that are accessed only from one thread. We have omitted this transformation, because its assumptions are not thread-local. It is not hard to see that the transformation is safe: given an execution of the program with removed thread-local synchronisation, we can always insert the synchronisation back. Because the synchronisation is thread-local, there cannot be any conflict with other actions and the resulting interleaving must be an execution. Moreover, any data race in the execution of the transformed program remains a data race in the execution of the original program.

It is an interesting question whether there are any other useful transformations of synchronisation. For example, it seems that eliminating one of two adjacent reads of a volatile location with the same value is a safe transformation.

6.3.2 Richer Trace Semantics

Our simple language assumes only statically created threads and no memory allocation. We believe that it would not be hard to add dynamic thread spawning and allocation to the trace semantics. However, the Java language also allows definition of immutable classes. Safe transformations should guarantee objects from these classes remain immutable even in presence of data races.

6.3.2.1 Object Creation and Immutability

Our trace semantics does not describe languages with object allocation, because there is no action for allocations and no rule for freshness of allocated references.

We believe that these could be easily added without any further changes to the framework.

The more difficult task is to add a JMM-like support for immutable objects. The Java Memory Model devotes a sizable proportion of the specification to the guarantees of immutability. For programmers, the JMM provides a way to create objects that will appear immutable even when there are data races. For compilers, the fields of immutable objects can be cached and any accesses to such fields can be freely reordered with other actions. To differentiate between the immutable object's initialisation and the rest of its immutable lifetime, the JMM introduces a special freeze action and guarantees that if a thread sees a reference to the immutable object that was published after the freeze action then all accesses to locations reachable from that location will return values at least as fresh as the values at the time of the freeze action. The main difficulty is that there is no global time and no global store in the JMM, so the memory model must define the meaning of the phrases “publish after”, “reachable”, “at the same time”, etc. It would be interesting to see whether we could capture these ideas from the perspective of semantic transformations, and then derive the guarantees from the transformations.

6.3.2.2 Dynamic Thread Creation

Dynamic thread creation can be added in a similar way to the JMM, i.e., by introducing a thread spawn action, and defining the thread spawn action and the corresponding thread start action to be a release-acquire pair. Similarly, we can introduce a thread finish and thread join actions and make them a release-acquire pair. It seems this refinement would require only small changes in the proofs of safety to accommodate these new actions and the updated synchronises-with relation.

6.3.3 Termination Properties

Our notion of safety guarantees that nothing unexpected happens. However, one should expect more from well-behaved transformations. For example, a well-behaved transformations should not transform a program that always terminates to a program that never terminates or vice versa.

To illustrate the problem more concretely, consider a transformation that

transforms any program to a traceset that contains just one empty trace. By our definition of safety, this is a safe transformation. We believe that such transformations should not be allowed. Ideally, for any execution of the transformed program there should be an execution of the original program with the same external behaviour and termination behaviour, where the termination behaviour can be either terminating or non-terminating or deadlocked. We suspect that the reasoning about termination behaviours requires extending the framework to possibly infinite traces. This might be quite difficult, because we rely on finiteness in many places.

6.3.4 Formal Techniques

Due to the complicated and often counter-intuitive structure of the definitions and proofs, the semantics of weak memory models and transformations is an ideal target for proof mechanisation. In our earlier work (Aspinall and Ševčík, 2007), we have formally checked the DRF guarantee for the Java Memory Model in the Isabelle/HOL theorem prover (Paulson, 1988) with the intent to continue with proving safety of transformations in the JMM. During the preparation for the formalisation we have found the counterexamples described in Chapter 5 of this thesis.

To avoid such problems in our theory, we would like to check the results from Chapters 3 and 4 in an interactive theorem prover and build a certified optimising compiler for our simple concurrent language. This would be an important improvement on the current state-of-the-art certified compilation: existing mechanisations cover only sequential programs (Leroy, 2006) or annotated concurrent programs (Hobor et al., 2008).

Another relevant formal technique is model checking programs while taking into account possible transformations allowed in a weak memory model. Such work has been done for hardware optimisations (Burckhardt et al., 2006; Yang et al., 2004, 2005), where it is a useful tool for placing memory barriers that prevent reordering. Although interesting theoretically, we believe that model checking programming language memory models is probably not as important, because programs should be data race free and by the DRF guarantee, model checkers can continue assuming the interleaved semantics if they establish the data race freedom.

Bibliography

- Abadi, M., Birrell, A., Harris, T., and Isard, M. (2008). Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA. ACM.
- Ada95 (1995). *Ada 95 Reference Manual. The Language. The Standard Libraries*. International Organization for Standardization. ANSI/ISO/IEC-8652:1995.
- Adve, S. V. and Aggarwal, J. K. (1993). A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624.
- Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76.
- Adve, S. V. and Hill, M. D. (1990). Weak ordering — a new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA '90)*, pages 2–14.
- Adve, S. V. and Hill, M. D. (1992). Sufficient conditions for implementing the Data-Race-Free-1 memory model. Technical Report 1107, University of Wisconsin, Madison.
- Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1996). *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition.
- Aspinall, D. and Ševčík, J. (2007). Formalising Java's data race free guarantee. In Schneider, K. and Brandt, J., editors, *TPHOLs*, volume 4732 of *LNCS*, pages 22–37. Springer.

- Bacon, D. F., Strom, R. E., and Tarafdar, A. (2000). Guava: A dialect of Java without data races. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 382–400, Minneapolis, Minnesota.
- Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA. ACM.
- Boyapati, C. and Rinard, M. (2001). A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, New York, NY, USA. ACM Press.
- Boyland, J. (2008). An operational semantics including “volatile” for safe concurrency. Technical Report ICIS-R08013, Radboud University Nijmegen. Informal Proceedings of FTfJP'2008.
- Brauer, W., Reisig, W., and Rozenberg, G., editors (1987). *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*. Springer.
- Brookes, S. (2007). A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270.
- Brookes, S. D. (1993). Full abstraction for a shared variable parallel language. In *LICS*, pages 98–109. IEEE Computer Society.
- Burckhardt, S., Alur, R., and Martin, M. M. K. (2006). Bounded model checking of concurrent data types on relaxed memory models: A case study. In Ball, T. and Jones, R. B., editors, *CAV*, volume 4144 of *LNCS*, pages 489–502. Springer.
- Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. (1999). An event-based structural operational semantics of multi-threaded Java. In Alves-Foss, J., editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer.
- Cenciarelli, P., Knapp, A., and Sibilio, E. (2007). The java memory model: Operationally, denotationally, axiomatically. In Nicola, R. D., editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 331–346. Springer.

- Click, C. (1995). Global code motion/global value numbering. *SIGPLAN Not.*, 30(6):246–257.
- ECMA (2006a). *Standard ECMA-334 - C# Language Specification*. ECMA, 4 edition.
- ECMA (2006b). *Standard ECMA-335 - Common Language Infrastructure (CLI)*. ECMA, 4 edition.
- Elmas, T., Qadeer, S., and Tasiran, S. (2007). Goldilocks: a race and transaction-aware Java runtime. *SIGPLAN Not.*, 42(6):245–255.
- Flanagan, C. and Freund, S. N. (2000). Type-based race detection for Java. In *PLDI*, pages 219–232, New York, NY, USA. ACM Press.
- Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P. B., Gupta, A., and Hennessey, J. L. (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, pages 15–26.
- Gniady, C., Falsafi, B., and Vijaykumar, T. N. (1999). Is SC + ILP = RC? *SIGARCH Comput. Archit. News*, 27(2):162–171.
- Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005a). *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional.
- Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005b). *Java(TM) Language Specification, The (3rd Edition) (Java Series)*, chapter Threads and Locks, pages 557–573. Addison-Wesley Professional.
- Gosling, J., Joy, B., and Steele, G. L. (1996). *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300.
- Herlihy, M. and Shavit, N. (2008). *The art of multiprocessor programming*. Morgan Kaufman.
- Hill, M. D. (1998). Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34.

- Hobor, A., Appel, A. W., and Nardelli, F. Z. (2008). Oracle semantics for concurrent separation logic. In Drossopoulou, S., editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer.
- Intel (2002). A formal specification of Intel Itanium processor family memory ordering. Available from <http://www.intel.com/design/itanium/downloads/251429.htm>.
- Intel (2007). Intel 64 architecture memory ordering white paper. Available from <http://www.intel.com/products/processor/manuals/318147.pdf>.
- Kennedy, K. and Allen, J. R. (2002). *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall Professional Technical Reference.
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.
- Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691.
- Lea, D. (2008). The JSR-133 cookbook for compiler writers. <http://g.oswego.edu/dl/jmm/cookbook.html>.
- Ledgard, H. (1983). *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Leroy, X. (2006). Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press.
- Manson, J. (2004). *The Java memory model*. PhD thesis, University of Maryland, College Park.
- Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA. ACM Press.

- Mattsson, H., Nilsson, H., and Wikstrom, C. (1998). Mnesia - a distributed robust DBMS for telecommunications applications. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 152–163, London, UK. Springer-Verlag.
- Mazurkiewicz, A. W. (1986). Trace theory. In Brauer et al. (1987), pages 279–324.
- Moore, K. F. and Grossman, D. (2008). High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 51–62, New York, NY, USA. ACM.
- Morrison, V. (2005). Understand the impact of low-lock techniques in multi-threaded apps. *MSDN Magazine*.
- Murphy, B. R., Menon, V., Schneider, F. T., Shpeisman, T., and Adl-Tabatabai, A.-R. (2008). Fault-safe code motion for type-safe languages. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 144–154, New York, NY, USA. ACM.
- Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for Java. *SIGPLAN Not.*, 41(6):308–319.
- Paleczny, M., Vick, C., and Click, C. (2001). The Java Hotspot(TM) server compiler. In *USENIX Java(TM) Virtual Machine Research and Technology Symposium*.
- Paulson, L. C. (1988). Isabelle: The next seven hundred theorem provers. In Lusk, E. L. and Overbeek, R. A., editors, *CADE*, volume 310 of *Lecture Notes in Computer Science*, pages 772–773. Springer.
- Pugh, W. (2000). The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455.
- Pugh, W. and Manson, J. (2004). Java memory model causality test cases. <http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html>.
- Saraswat, V., Jagadeesan, R., Michael, M., and von Praun, C. (2007). A theory of memory models. In *ACM 2007 SIGPLAN Conference on Principles and Practice of Parallel Computing*. ACM.

- Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T., Myreen, M. O., and Alglave, J. (2009). The semantics of x86-cc multiprocessor machine code. In Shao, Z. and Pierce, B. C., editors, *POPL*, pages 379–391. ACM.
- Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411.
- Shasha, D. and Snir, M. (1988). Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312.
- Shavit, N. and Touitou, D. (1995). Software transactional memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213.
- Sparc International (2000). Sparc architecture manual, version 9. Available from <http://developers.sun.com/solaris/articles/sparcv9.html>.
- Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Sura, Z., Fang, X., Wong, C.-L., Midkiff, S. P., Lee, J., and Padua, D. (2005). Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, New York, NY, USA. ACM.
- Ševčík, J. (2008). The Sun Hotspot JVM does not conform with the Java memory model. Technical Report EDI-INF-RR-1252, School of Informatics, University of Edinburgh.
- Ševčík, J. and Aspinall, D. (2008). On validity of program transformations in the Java memory model. In Vitek, J., editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer.
- Winkel, G. (1986). Event structures. In Brauer et al. (1987), pages 325–392.
- Yang, Y., Gopalakrishnan, G., and Lindstrom, G. (2004). Memory-model-sensitive data race analysis. In Davies, J., Schulte, W., and Barnett, M., editors, *ICFEM*, volume 3308 of *LNCS*, pages 30–45. Springer.

Yang, Y., Gopalakrishnan, G., and Lindstrom, G. (2005). UMM: an operational memory model specification framework with integrated model checking capability: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487.