The Java Memory Model and Program Transformations

Jaroslav Ševčík

University of Edinburgh

Supported by ITI Techmedia

Overview

- Introduction to the Java Memory Model (JMM)
- Overview of transformation legality in the JMM.
- Notes on implementing the JMM.
- Definition of the JMM:
  - overview of the formal definition,
  - operational view of the JMM,
  - examples.
- Flaws in the JMM:
  - several standard optimisations not legal. . .
  - . . . including some that are implemented in HotSpot.

Java Memory Model

The Java Memory Model (JMM)

- is a contract between hardware, compiler and developers.
- describes legal behaviours in a multi-threaded Java code with respect to the shared memory.
- implies:
  - Promises for programmers to enable reusable reasoning about programs.
  - Security guarantees.
  - Legal optimisations for compiler/JVM implementors.

Brief History of JMM

The Java Memory Model (Manson, Pugh and Adve, POPL 2005) was introduced after the original memory model was found to be "fatally flawed" (Pugh, 2000). The main flaws were:

- Many optimisations illegal,
- Final fields could be observed to change,
- Unclear semantics of finalisation.

The JMM aims to fix these problems with 3 different fixes.

The core of the JMM only deals with the first problem. This tutorial is mostly about the core.

Brief History of JMM

The new JMM:

- part of the Java Language Specification,
- accompanied by a POPL paper with two theorems:
  - Data race free programs have only sequentially consistent behaviours (Theorem 3 of the POPL paper, DRF guarantee). This allows using standard reasoning for DRF programs.
  - Reordering of independent statements is legal. (Theorem 1.) This was falsified by Cenciarelli et al. (2007). Can be partially fixed.
- claims several properties informally:
  - Out-of-thin-air behaviours are prevented (security).
  - Adding synchronisation is a legal transformation.

Transformation Validity

Instead of analysing large transformations, we will look at several peephole transformations and classify their validity in the JMM.

Program transformation is valid in the JMM if it cannot introduce a new observable behaviour.

More precisely, for every JMM-execution of the transformed program there must be a JMM-execution of the original program with the same behaviour. Note that validity is compositional.

In the JMM, observable behaviour of an execution are the I/O actions in the execution together with the termination behaviour.

Transformation Overview

1. Trace preserving transformation, e.g.,

$$\texttt{if(r==1) x:=r; else x:=1} \rightarrow \texttt{x:=1}.$$

2. Irrelevant read introduction/elimination:

$$\texttt{r:=x;}C \rightleftarrows C,$$

   where $\texttt{r}$ is not free in C.

3. Redundant read after read elimination:

$$\texttt{r1:=x;r2:=x} \rightarrow \texttt{r1:=x;r2:=r1}.$$

4. Redundant read after write elimination:

$$\texttt{x:=r1;r2:=x} \rightarrow \texttt{x:=r1;r2:=r1}.$$

Transformation Overview

5. Redundant write after read elimination:

$$\texttt{r:=x;x:=r} \longrightarrow \texttt{r:=x}.$$

6. Redundant write before write elimination:

$$\texttt{x:=r1;x:=r2} \longrightarrow \texttt{x:=r2}.$$

7. Normal memory access reordering:

$$\texttt{r1:=x;r2:=y} \longrightarrow \texttt{r2:=y;r1:=x},$$
$$\texttt{x:=r1;y:=r2} \longrightarrow \texttt{y:=r2;x:=r1},$$
$$\texttt{r1:=x;y:=r2} \rightleftarrows \texttt{y:=r2;r1:=x}.$$

Transformation Overview

8. Roach motel reordering, i.e., moving a memory access inside a synchronised section. This means reordering a memory access with a later acquire or with an earlier release.

$$\texttt{memop;acq} \rightarrow \texttt{acq;memop},$$

$$\texttt{rel;memop} \rightarrow \texttt{memop;rel},$$

where

- `memop` is `x:=r1` or `r1:=x`,
- `acq` is `lock m` or `r2:=v`,
- `rel` is `unlock m` or `v:=r2`.

Transformation Validity Overview

Transformation	SC	JMM	JMM-Alt
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	×	×	×
External action reordering	×	×	×

Note: The SC column only applies to adjacent statements.

Notes on implementing the JMM

The situation with the JMM is not settled:

Some standard optimisations, including CSE, are not valid, but compilers still perform them (Sun HotSpot). One can even observe behaviours forbidden by the JMM.

It is not likely that the JVMs will sacrifice these optimisations. The JMM will have to be changed. You might be shooting at a moving target!

In addition, Java 7 will introduce explicit memory fences in the JDK. These do not have a clear meaning in the JMM.

Notes on implementing the JMM

Valid compiler optimisation:

- If your optimisation does not change any sequence of shared memory accesses, it is legal. This includes:
  - Loop unrolling,
  - Final/static method inlining,
  - Redundant conditional elimination (e.g., if both branches are the same).
- Removing reads based on previous writes is legal
  - . . . even if the read and the write not adjacent.
- Removing overwritten writes is legal.
- Reordering of independent reads/writes almost legal:
  - Loop rearrangements, code motion.

Notes on implementing the JMM

Compiler optimisations that should be avoided:

- Do not reuse older reads/writes across synchronisation.
- Avoid optimisations that introduce writes with new values or to memory locations that would not be otherwise written.
  - These are often illegal even in the DRF guarantee model.
  - Introducing a write with the same value and to the same location is safe (but probably not profitable).
- Avoid introducing reads (even if their value is thrown away).

Notes on implementing the JMM

Optimisations to avoid:

- Avoid reusing values of previous reads (including CSE).
- Do not reorder with I/O operations
- Do not reorder with synchronisation
  - In fact, treating synchronisation and I/O as opaque is a good idea.

If you want to be completely safe, do not reorder memory accesses.

Notes on implementing the JMM

Legality of hardware optimisations is a slightly different issue because we must relate two different models–the processor model and the JMM:

- For each execution of the processor model there must be a JMM-execution with the same behaviour.

The difficulty of showing validity depends on the model:

- simple for write buffering model (Sun TSO, x86) or location consistency because these models are simple.
- straightforward for Intel Itanium because there is a total order that can be used to construct the JMM execution.
- difficult for Power and ARM MMs because they are not well-understood.

Implementing final fields

JVMs must guarantee visibility of data reachable from final fields for the threads that did not see an escaped reference to the containing instance from the constructor.

The JMM only specifies what “visibility”, “reachable” and “escape” mean in terms of the JMM vocabulary.

Implementing final fields

JVMs must guarantee visibility of data reachable from final fields for the threads that did not see an escaped reference to the containing instance from the constructor.

The JMM only specifies what “visibility”, “reachable” and “escape” mean in terms of the JMM vocabulary.

In practice, implementation must ensure that

- final field values become visible to other threads before the reference to the constructed object.
- final fields are read after reading of the reference of their containing object.

The former sometimes needs memory fence/prevents optimisations, the latter often automatic (dependencies).

Basic Definitions (Action)

Java does not have a global store or global time. These are approximated by actions, orders and a visibility function.

An action $\langle t, k, v, u \rangle$ is described by:

1. thread t performing the action,
2. kind k of the action:
   - volatile read or write, non-volatile read or write, lock, unlock, external, synthetic actions (first and last action of thread etc.),
   - volatile reads, writes, locks and unlocks are synchronization actions,
3. runtime variable or monitor v associated with the action,
4. unique identifier u.

Basic Definitions (Execution)

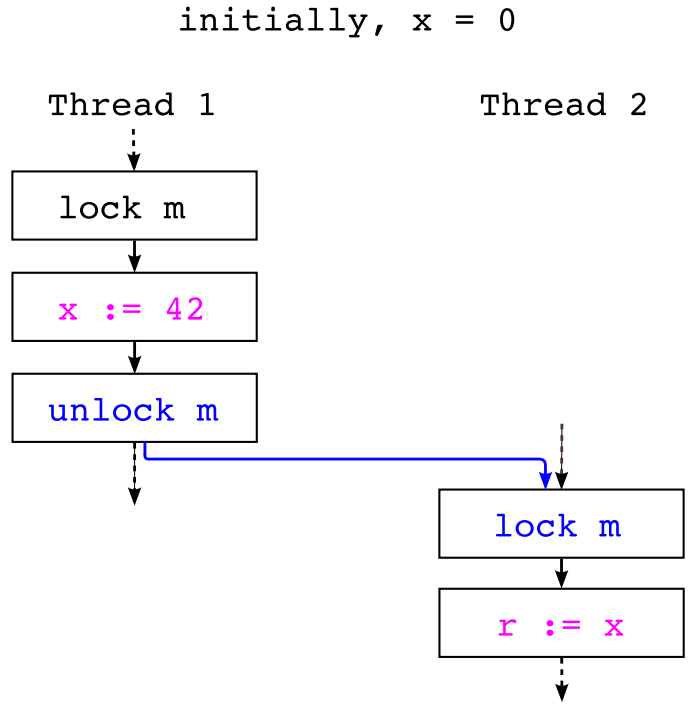
Program order $\leq_{po}$ is a union of total orders on actions of each thread.

Synchronisation order $\leq_{so}$ is a total order on synchronisation actions.

Happens-before order: $a \leq_{hb} b$ if either:

1. $a \leq_{so} b$ and a, b is a release-acquire pair:
   - a is an unlock, b is a lock on the same monitor, or
   - a is a volatile write to v, b is a volatile read from v,
2. $a \leq_{po} b$, or
3. there is c such that $a \leq_{hb} c$ and $c \leq_{hb} b$ (i.e., $\leq_{hb}$ is transitive).

Happens-before Example I



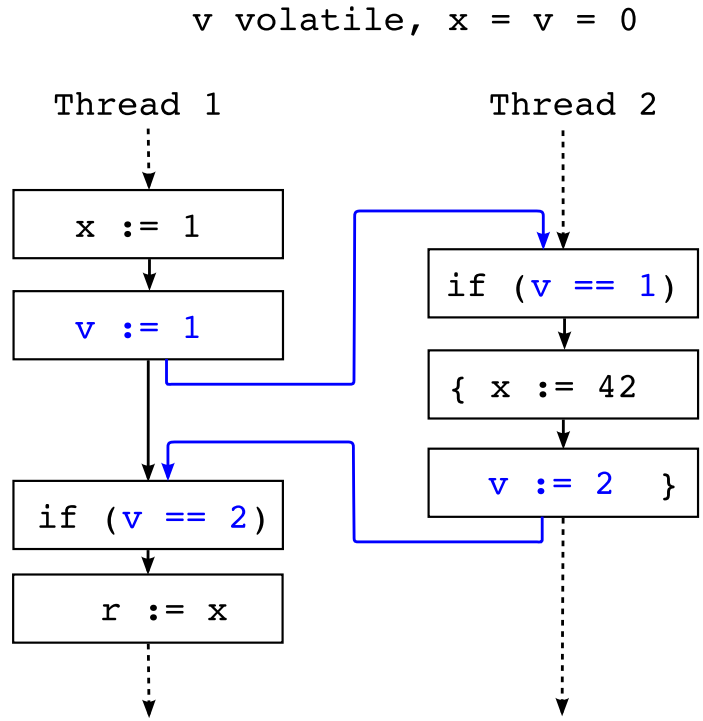
Assuming that `unlock(m)` $\leq_{so}$ `lock(m)`, we have

$$\texttt{x:=42} \leq_{hb} \texttt{r:=x},$$

because `x:=42` $\leq_{po}$ `unlock(m)` $\leq_{so}$ `lock(m)` $\leq_{po}$ `r:=x`, and `unlock(m)` and `lock(m)` are release-acquire pair.

Happens-before Example II

In general, reads cannot see writes that happen after them or are overwritten.



If `r:=x` gets executed, then it must see the write `x:=42`. The other writes to `x` are overwritten.

Execution Formally

Execution $\langle P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb} \rangle$ consists of:

1. program P
2. set of actions A
3. program order $\leq_{po}$ – union of total orders on actions of each thread
4. synchronization order $\leq_{so}$ – total order over all synchronization actions
5. write-seen function W
6. value-written function V
7. synchronizes-with order $<_{sw}$
8. happens-before order $\leq_{hb}$

Well-formed executions

Execution is well-formed if:

- each read of x sees a write to x, i.e. $r.v = W(r).v$,
- $\{x \in A : x \leq_{so} y\}$ is finite for each $y \in A$,
- $\leq_{so}$ is consistent with $\leq_{po}$,
- $\leq_{so}$ is consistent with mutual exclusion of locks,
- the execution is intra-thread consistent,
- volatile reads are consistent with $\leq_{so}$, i.e. for all volatile reads r we have $\neg(r \leq_{so} W(r))$ and there is no w s.t. $w.v = r.v \wedge W(r) <_{so} w \leq_{so} r$ (volatile reads see the most recent write in $\leq_{so}$),
- all reads are consistent with $\leq_{hb}$ (reads see a most recent write in $\leq_{hb}$).

Legal Execution I

An execution $E = \langle P, A, \leq_{po}, \leq_{so}, W, V, <_{sw}, \leq_{hb} \rangle$ satisfies the causality requirement if there is a sequence of sets of actions $\{C_i\}$ satisfying

- $C_0 = \emptyset$
- $C_i \subset C_{i+1}$
- $A = \bigcup C_i$

and a sequence of well formed executions $E_i = \langle P, A_i, \leq_{po_i}, \leq_{so_i}, W_i, V_i, <_{sw_i}, \leq_{hb_i} \rangle$ such that the following holds:

Legal Execution II

1. $C_i \subseteq A_i$,
2. $\leq_{hb_i} |_{C_i} = \leq_{hb} |_{C_i}$,
3. $\leq_{so_i} |_{C_i} = \leq_{so} |_{C_i}$,
4. $V_i|_{C_i} = V|_{C_i}$,
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$,
6. for any read $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$,
7. for any read $r \in C_i - C_{i-1}$ we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$,
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 $j \geq i$ ($<_{ssw_i}$ is transitive reduction of $<_{sw_i}$ without edges from $\leq_{po_i}$)

Legal Execution II

. . . can be weakened to

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 any read $r \in A_i - C_{i-1}$ we have $W_i(r) \leq_{hb_i} r$,
6. for any read $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$.

without invalidating the DRF guarantee.

Sequential Consistency (SC)

We say that an execution is sequentially consistent if there is a total order on actions consistent with the happens-before order such that each read sees the most recent write in that order.

In other words, sequential consistency simulates interleaved semantics.

Note:
Sequential consistency and well-formedness imply legality.

Data Race Free Program

Two accesses to the same non-volatile variable, of which at least one is write, are a data race if they are not ordered by happens-before.

Program P is data race free (DRF) if no sequentially consistent execution of P contains a data race.

This is equivalent to the DRF definition in terms of interleavings and adjacent actions.

Committing Sequence

- Start with a “well-behaved” execution—all reads see writes from the same thread or through synchronisation,
  - i.e., reads see writes that happen-before them.
- The JMM commits one or more read-write data races.
- Then it restarts the execution, but it must keep the commitment:
  - It must perform all the committed actions.
  - The reads must see the value that they were committed with.
  - Happens-before relationships of actions in the commitment must be preserved.
- The JMM may commit more actions and restart again.

Well-behaved Executions

In a well-behaved execution, all reads see writes that happen-before them.

For DRF programs, execution is well-behaved iff SC.

Otherwise, the program

x = 0	
`lock m`	`x:=2`
`x:=1`	`lock m`
`unlock m`	`r1:=x`
	`r2:=x`
	`unlock m`

can result in `r1` $\neq$ `r2` in a well-behaved execution, which is not possible in SC.

Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2

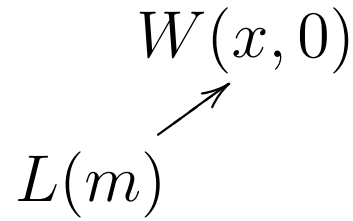
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2

$W(x, 0)$

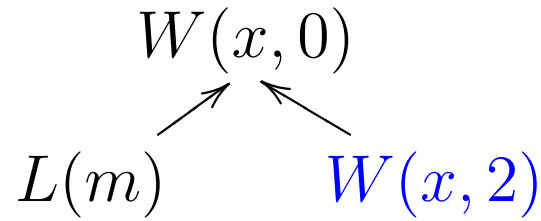
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



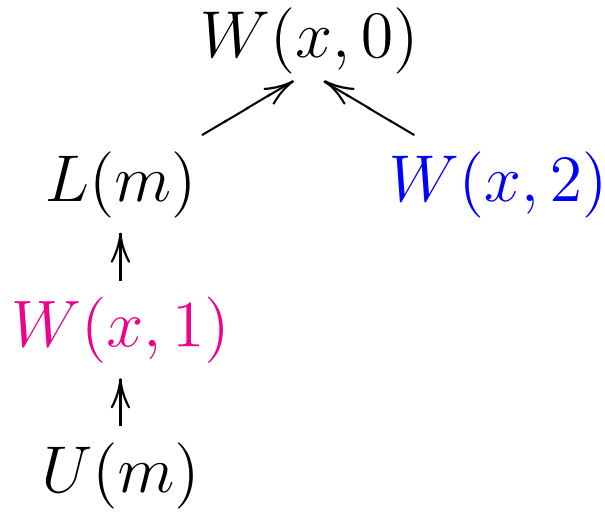
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



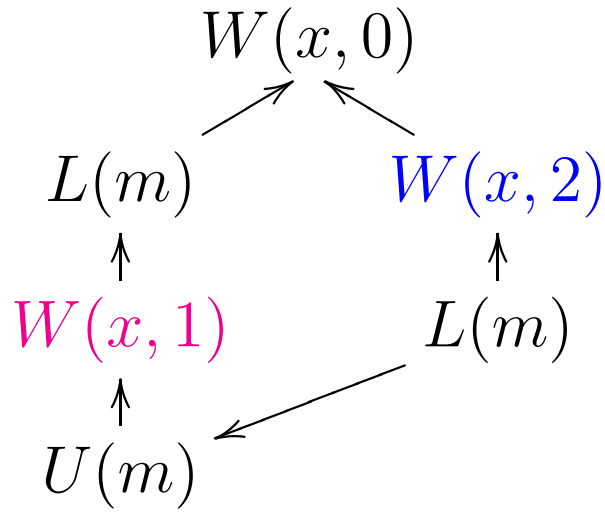
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



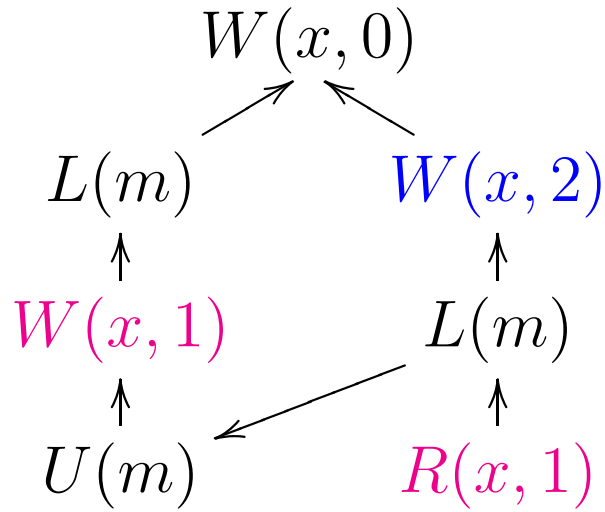
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



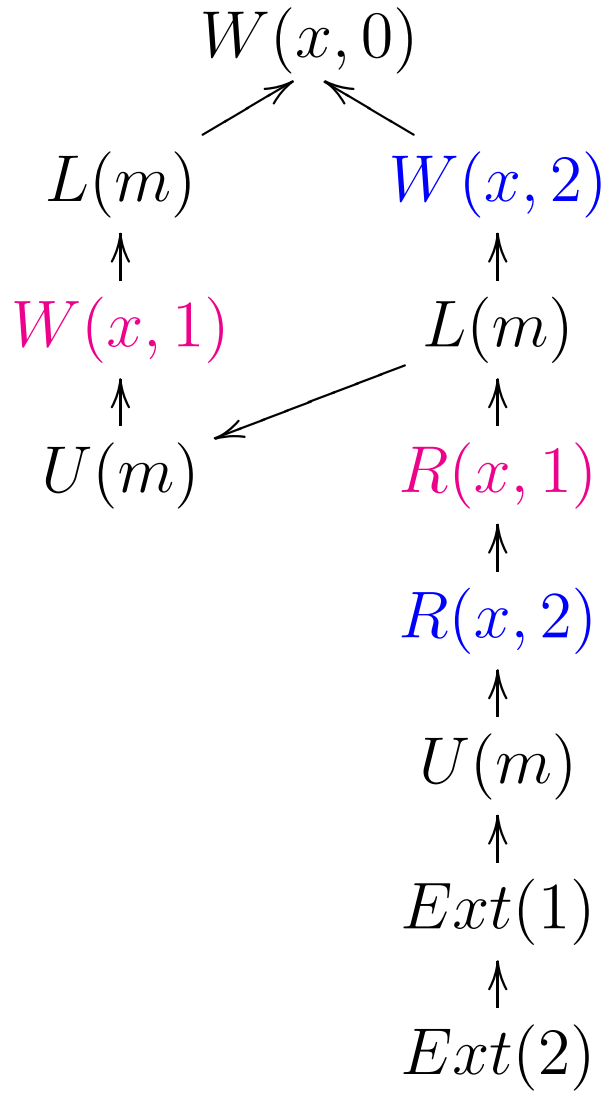
Well-behaved execution example

x = 0	
lock m	x:=2
x:=1	lock m
unlock m	r1:=x
	r2:=x
	unlock m
	print r1
	print r2



Well-behaved execution example

x = 0	
`lock m`	`x:=2`
`x:=1`	`lock m`
`unlock m`	`r1:=x`
	`r2:=x`
	`unlock m`
	`print r1`
	`print r2`



Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1

We want to justify the result $r1 = r2 = 1$.

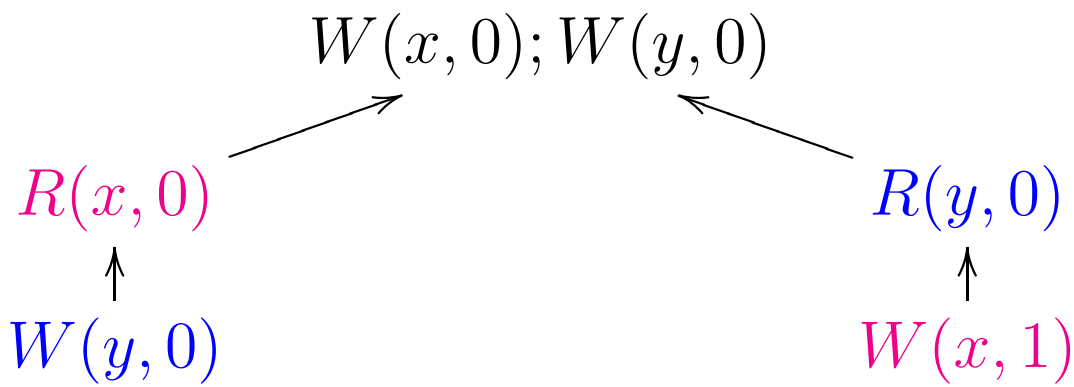
We will have to find a well-behaved execution, where each read sees a write that happens before it.

Then we commit a data race from this execution, and restart. After restarting I must use the committed races, and preserve their ordering by happens-before. The reads that are not committed will see writes that happen-before them.

Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1

The only well-behaved execution:

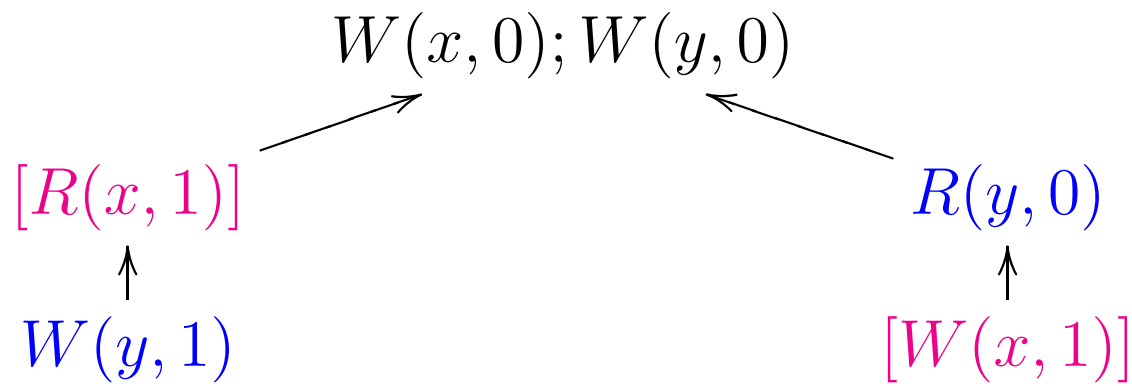


This has two data races: $\langle R(x,0), W(x,1)\rangle$ and $\langle R(y,0), W(y,0)\rangle$.

Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1

Let us commit $\langle R(x, \mathbf{1}), W(x, 1)\rangle$. Now we must use the race, leaving just one possibility for execution:

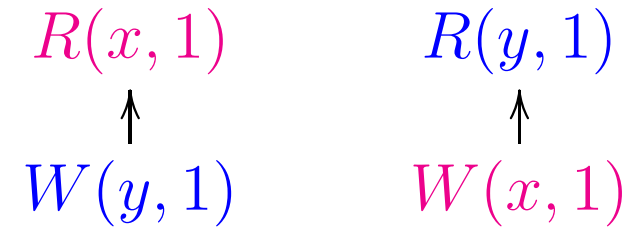


The only available race is $\langle W(y, 1), R(y, 0)\rangle$.

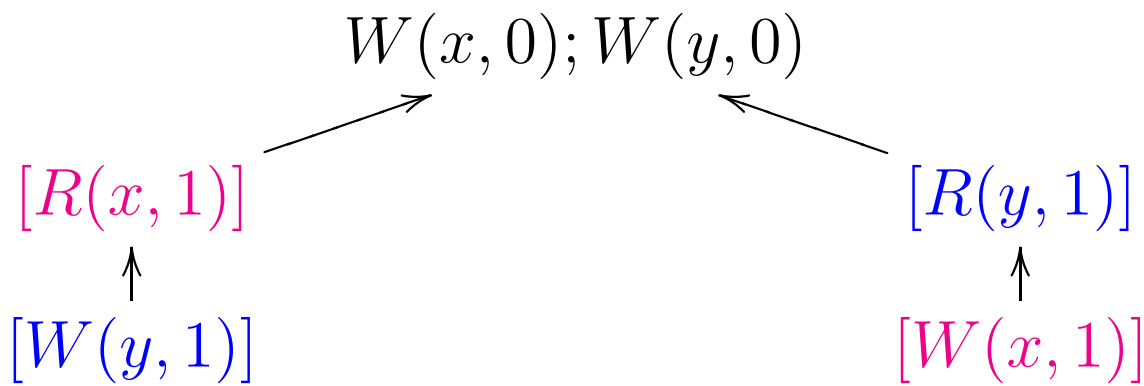
Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1

Now our commit set is:

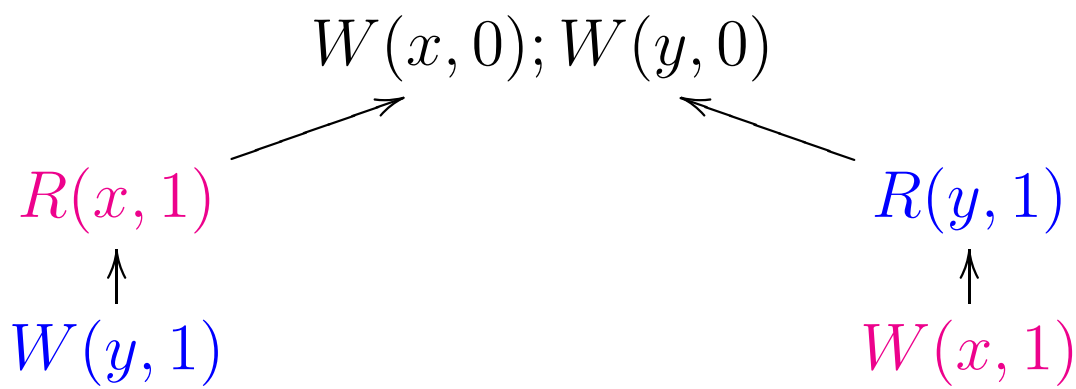


. . . and the only execution:



Justification Example

x = y = 0	
r1 := x	r2 := y
y := r1	x := 1



means that we can get $\texttt{r1} = \texttt{r2} = 1$.

Bug I—reordering

Let's take the program (Cenciarelli et al., 2007)

x = y = z = 0

r1:=z	r2:=x
if (r1==1) {x:=1; y:=1}	r3:=y
else {y:=1; x:=1}	if (r2==r3==1)
	z:=1

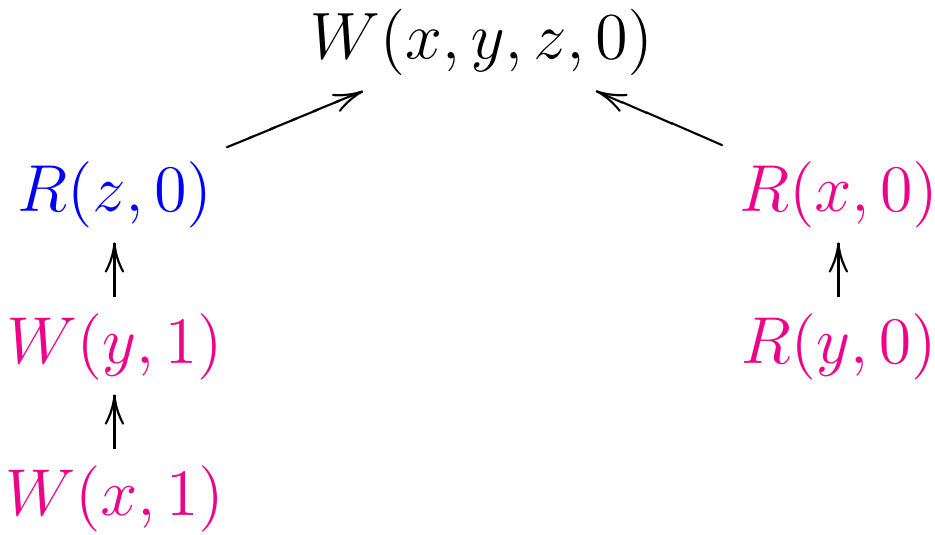
Can we get $\mathtt{r1} = \mathtt{r2} = \mathtt{r3} = 1$?

No! When we commit the races on x and y, $W(y) \leq_{hb} W(x)$. However, when we commit the read of 1 from z, we cannot keep the ordering of the writes.

Bug I—reordering

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

Start with a well-behaved execution:

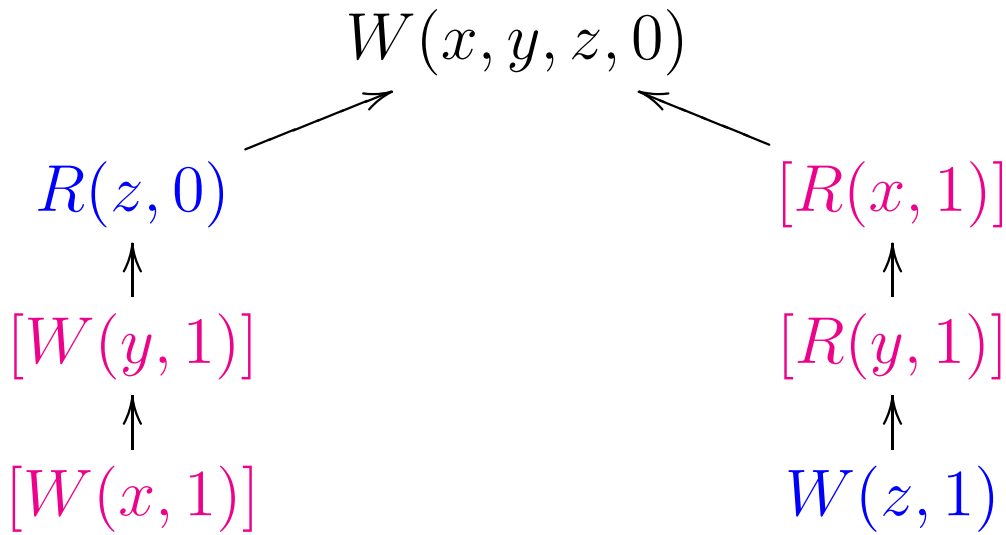


Bug I—reordering

x = y = z = 0

r1:=z	r2:=x
if (r1==1) {x:=1; y:=1}	r3:=y
else {y:=1; x:=1}	if (r2==r3==1)
	z:=1

After committing races on x and y:

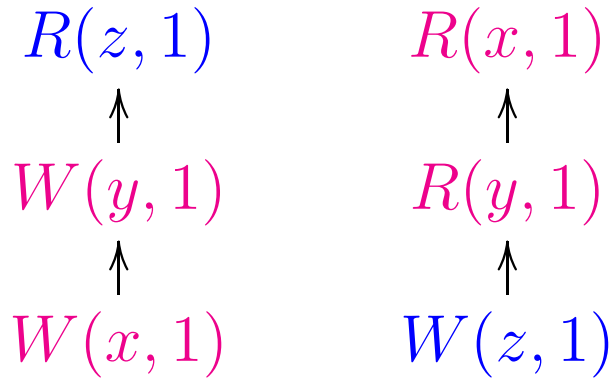


Bug I—reordering

x = y = z = 0

r1:=z	r2:=x
if (r1==1) {x:=1; y:=1}	r3:=y
else {y:=1; x:=1}	if (r2==r3==1)
	z:=1

After the commits we get the commit set



which is impossible to honor.

Bug I—reordering

x = y = z = 0

`r1:=z`	`r2:=x`
`if (r1==1) {x:=1; y:=1}`	`r3:=y`
`else {x:=1; y:=1}`	`if (r2==r3==1)`
	`z:=1`

However, the result is possible if I swap the assignments to x and y in one of the branches. Bug in the memory model, reordering of independent normal memory accesses should not introduce a new behaviour!

Can be fixed by relaxing the requirement on the preservation of the structure of commitments.

Bug II – Read After Read Elim

Reusing values from previous reads is illegal in the JMM in general:

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

cannot result in `r2 = 1`.

Bug II – Read After Read Elim

Reusing values from previous reads is illegal in the JMM in general:

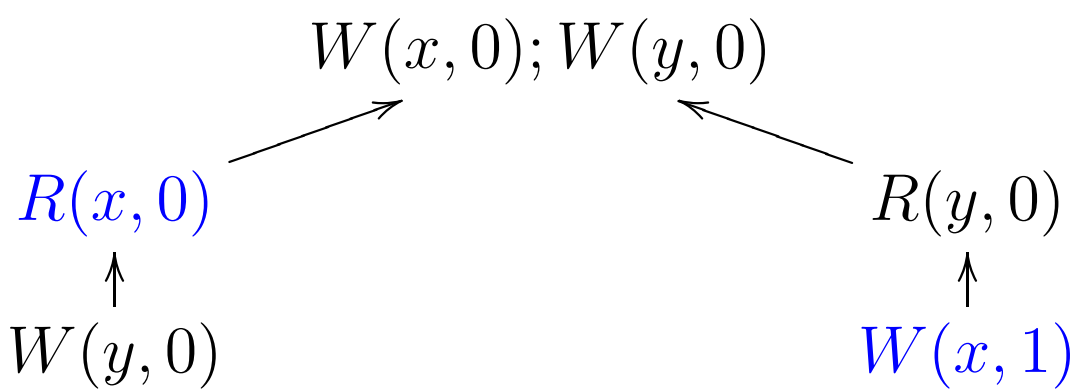
x = y = 0	
`r1 := x`	`r2 := y`
`y := r1`	`if (r2 == 1)`
	`{r3 := r2`
	`x := r3}`
	`else x := 1`

But after replacing reusing the value of y, `r2` can be `1`!

Bug II – Read After Read Elim

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

Start with:

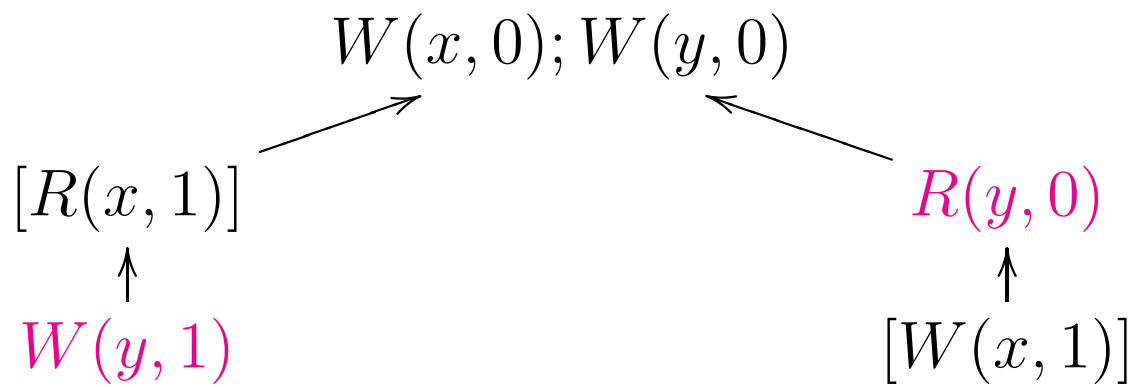


and then commit the data race on `x`.

Bug II – Read After Read Elim

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

After committing the race on `x`:

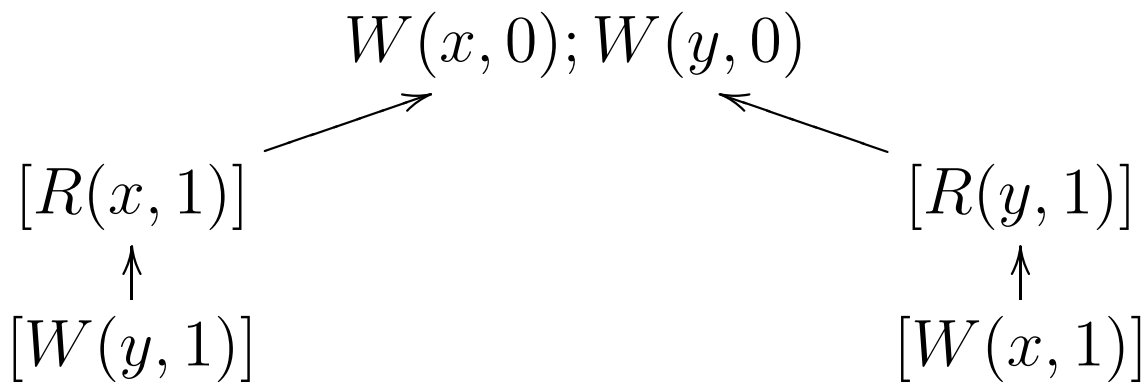


commit the data race on `y`.

Bug II – Read After Read Elim

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

Finally we obtain the result:

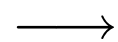


where $\texttt{r2} = 1$!

Bug II – HotSpot JVM

Sun's HotSpot JVM actually performs such an optimisation:

x = y = 0	
r1=x	r2=y
y=r1	**x=(r2==1)?y:1**
	print r2



x = y = 0	
r1=x	**x=1**
y=r1	**r2=y**
	print r2

The original program cannot print "1" by the JLS.

But the optimised program can print "1" even on a sequentially consistent processor!

Bug III – write-after-read elimination

Note that the program

x = 0	
`lock m`	`x:=2`
`x:=1`	`lock m`
`unlock m`	`r1:=x`
	`x:=r1`
	`r2:=x`
	`unlock m`

does not have a well-behaved execution where $\texttt{r1} \neq \texttt{r2}$ because reads must see a most recent write in $\leq_{hb}$.

So it is illegal to remove the write!

Bug IV – Adding Synchronisation

One of the design goals was that increasing synchronisation should not introduce new behaviour.

By increasing synchronisation we mean:

- Moving normal accesses into synchronised blocks (roach motel).
- Making variables volatile.

However, none of these transformations are legal in the JMM in general!

Bug IV – Adding Synchronisation

initially `x = y = z = 0`

`lock m`	`lock m`	`r1:=x`	`r3:=y`
`x:=2`	`x:=1`	`lock m`	`z:=r3`
`unlock m`	`unlock m`	`r2:=z`	
		`if(r1==2)`	
		`  y:=1`	
		`else`	
		`  y:=r2`	
		`unlock m`	

Behaviour `r1 = r2 = r3 = 1` not possible.

Bug IV – Adding Synchronisation

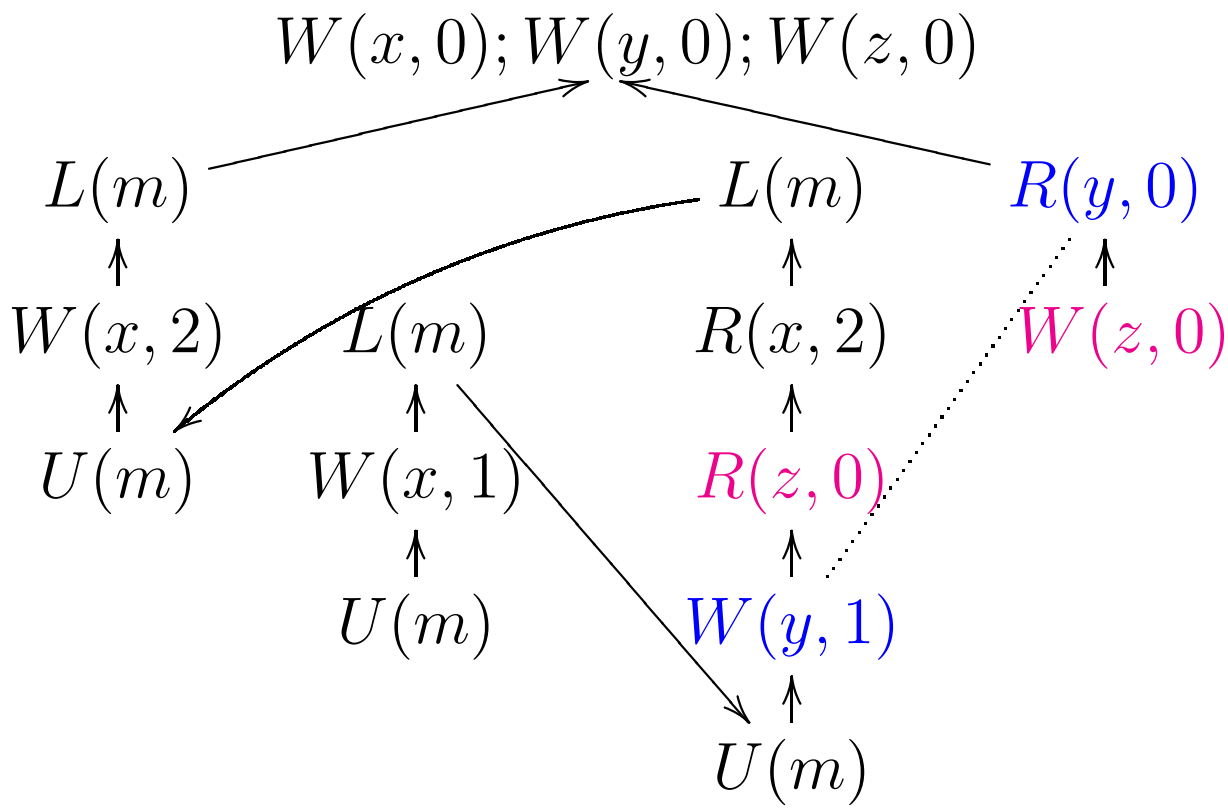
initially `x = y = z = 0`

`lock m`	`lock m`	`lock m`	`r3:=y`
`x:=2`	`x:=1`	`r1:=x`	`z:=r3`
`unlock m`	`unlock m`	`r2:=z`	
		`if(r1==2)`	
		`  y:=1`	
		`else`	
		`  y:=r2`	
		`unlock m`	

. . . but becomes possible after moving `r1:=x` inside the synchronised block. Let's start by committing the data race on `y`, and then on `z` with value 1.

Bug IV – Adding Synchronisation

Why is `r1 = r2 = r3 = 1` possible?



Committing the data race on `y` and then on `z` (with value 1).

Bug IV – Adding Synchronisation

initially `x = y = z = 0`

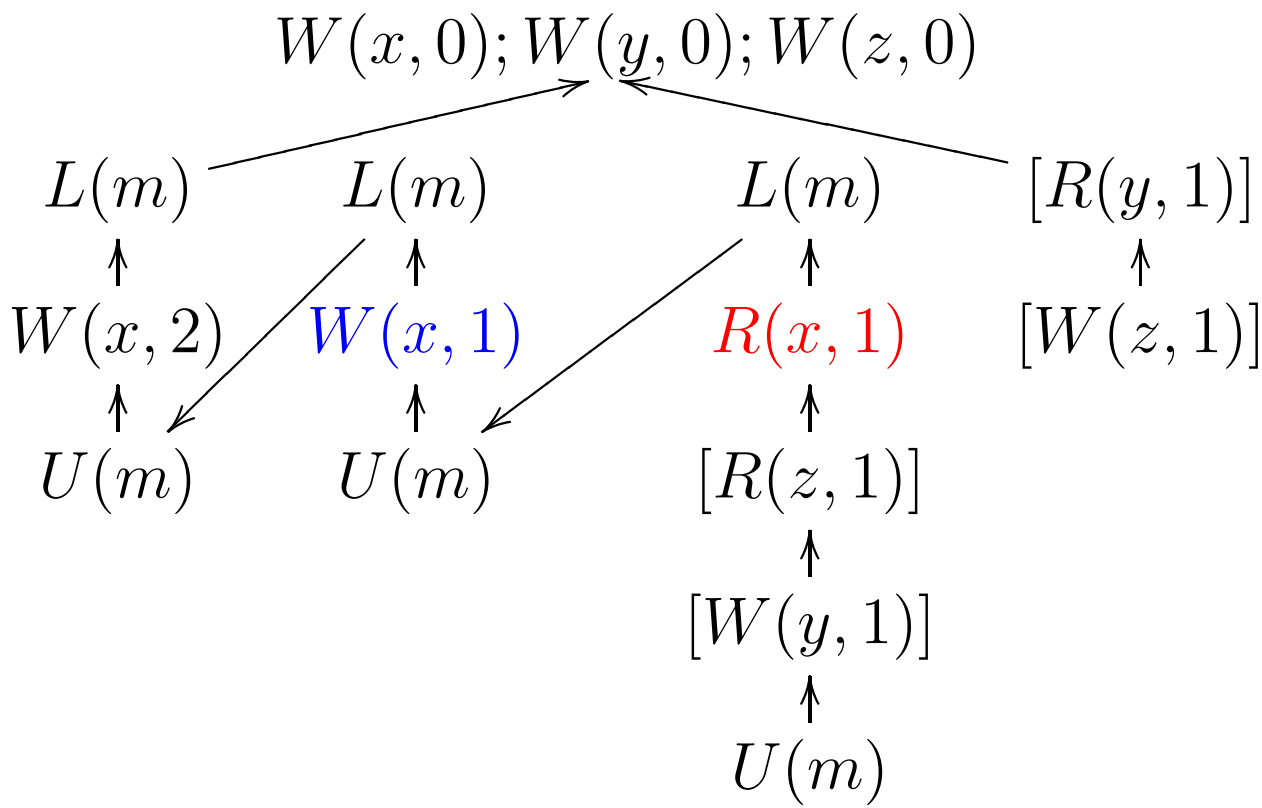
`lock m`	`lock m`	`lock m`	`r3:=y`
`x:=2`	`x:=1`	`r1:=x`	`z:=r3`
`unlock m`	`unlock m`	`r2:=z`	
		`if(r1==2)`	
		`  y:=1`	
		`else`	
		`  y:=r2`	
		`unlock m`	

Finally switch to the other branch of the `if` statement...

Note: this switch is impossible if `r1:=x` is before the lock, because `x` would have to be committed with `2`.

Bug IV – Adding Synchronisation

Why is `r1 = r2 = r3 = 1` possible?



. . . by changing the synchronisation order, so that the read of `x` sees the write of 1.

Proving Legality

Proving legality of compiler optimisation relatively straightforward:

- Take a justifying sequence of the transformed program...
- ...and massage it into a justifying sequence of the original program

Legality of hardware optimisations is straightforward if there is an order that we can use to commit the actions:

- For Sun TSO and Intel Itanium, the order is given directly by the processor specification.
- For Power and ARM: we first need to understand their spec.

Summary

The Java Memory Model:

- is the semantics of multi-threaded Java.
- satisfies most of its design goals...
- ...but not the most important one:
  - it does not allow several standard optimisations, and so it is not implemented by the reference JVM. (that does not mean that it is not implementable)
- many questions are still open.

We are looking for a new Java memory model (or a fix for the current one).

Thank you for your attention.

Questions?