# Lecture Notes on Weak Memory Models – Part I

Viktor Vafeiadis

June 16, 2014

## 1 Axiomatic Memory Models

In this section, we introduce axiomatic definitions of memory models.

### 1.1 Basic Setup

**Program Executions** The meaning of programs is defined as a set of consistent executions. An execution, $X$, is a tuple, containing (at least) the following components:

- A set of events, $A \subseteq \mathbb{N}$
- A labelling function $lab : A \to \mathsf{Label}$ where labels are given by the following grammar:

  $$\mathsf{Label} ::= \mathsf{Skip} \mid \mathsf{R}_{typ}(\ell, v) \mid \mathsf{W}_{typ}(\ell, v)$$

  where $\ell \in \mathsf{Loc}, v \in \mathsf{Val}$ and $typ$ is the access type (only relevant for memory models such as C11 that support multiple access types—ignore it for the time being).
- A relation over events, $po \subseteq A \times A$, denoting the program order.

The above three components are determined by the program itself. To these components, each memory model typically adds a few components justifying why the execution is possible. Common additional components are:

- A reads-from map, $rf : A \rightharpoonup A$, mapping read events to the write events that they read.
- A memory order, $mo \subseteq A \times A$, relating write events to the same location.
- A sequentially consistent order, $sc \subseteq A \times A$, relating all events that are of "sequential consistent" type.

**Mapping programs to executions** We define a denotation function that maps program expressions to a set of executions with the corresponding return values:

$$[\![\_]\!] : \mathsf{Command} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Execution})$$

This is defined inductively over the syntax of program expressions:

$$\begin{aligned}
[\![v]\!] &\stackrel{\text{def}}{=} \{v, (\{a\}, \{a \mapsto \mathsf{Skip}\}, \emptyset) \mid a \in \mathbb{N}\} \\
[\![x.\mathtt{load}(typ)]\!] &\stackrel{\text{def}}{=} \{v, (\{a\}, \{a \mapsto \mathsf{R}_{typ}(x, v)\}, \emptyset) \mid a \in \mathbb{N} \wedge v \in \mathsf{Val}\} \\
[\![x.\mathtt{store}(v, typ)]\!] &\stackrel{\text{def}}{=} \{v, (\{a\}, \{a \mapsto \mathsf{W}_{typ}(x, v)\}, \emptyset) \mid a \in \mathbb{N}\} \\
[\![e; e']\!] &\stackrel{\text{def}}{=} \{v', (A \uplus A', lab \cup lab', po \cup po' \cup (A \times A')) \mid \\
&\qquad (v, (A, lab, po)) \in [\![e]\!] \wedge (v', (A', lab', po')) \in [\![e']\!]\} \\
[\![\textbf{let } t = e \textbf{ in } e']\!] &\stackrel{\text{def}}{=} \{v', (A \uplus A', lab \cup lab', po \cup po' \cup (A \times A')) \mid \\
&\qquad (v, (A, lab, po)) \in [\![e]\!] \wedge (v', (A', lab', po')) \in [\![e'[v/x]]\!]\} \\
[\![e \parallel e']\!] &\stackrel{\text{def}}{=} \{v, (A \uplus A', lab \cup lab', po \cup po') \mid \\
&\qquad (v, (A, lab, po)) \in [\![e]\!] \wedge (_, (A', lab', po')) \in [\![e']\!]\} \\
[\![e \oplus e']\!] &\stackrel{\text{def}}{=} [\![e]\!] \cup [\![e']\!]
\end{aligned}$$

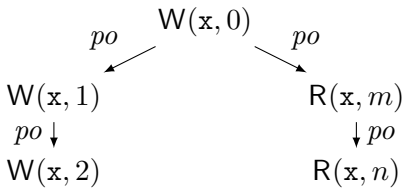
(For simplicity, we do not consider looping constructs because these could generate infinite executions.)

Note that the semantics of loads generates an arbitrary value that is being read.

**Example** As a simple example, consider the program:

```
                x.store(0);
x.store(1);     ||   x.load();
x.store(2)      ||   x.load()
```

Its executions are of the form:




for every possible $m, n \in \mathsf{Val}$. Obviously not every such execution is a valid run of the program. It is the memory model that will constrain the set of 'valid' executions by providing a number of constraints (axioms).

**Consistency axioms** A memory model $M$ defines a number of additional components that are added to executions together with a collection of conditions (axioms) which should be satisfied. An execution satisfying all the memory axioms of the model is said to be *consistent*.

## 1.2 The reads-from map

A standard component that all memory models add to execution is the reads-from map, $rf : A \rightharpoonup A$, mapping read events to the write events that they read.

We require that $rf$ satisfies the following two standard axioms:

**ConsistentRF** The reads-from map relates reads and writes with the same locations and values.

$$\forall a, b.\ rf(a) = b \implies \exists \ell, v.\ \mathsf{isread}_{\ell,v}(a) \wedge \mathsf{iswrite}_{\ell,v}(b)$$

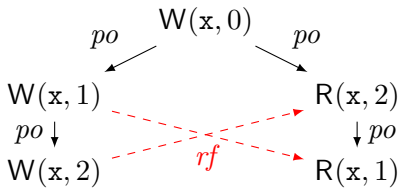
**ConsistentRFdom** The reads-from map is defined for all reads.

$$\forall a.\ \mathsf{isread}(a) \implies \exists b.\ rf(a) = b$$

(This axiom is simplified. In reality, one requires a $rf$ edge to exist only for reads that are known to happen after the relevant location has been initialized. Here, we assume that the executions start by initialisation writes to each location used in the execution.)

Using these axioms, we can immediately rule out some of the strange executions from our earlier example program. As we have to match each read to a write (because of ConsistentRFdom), and since the locations and values of matched read-writes have to agree (from ConsistentRF), we can only get the executions where $m, n \in \{0, 1, 2\}$.

We do, however, still have too many executions. For instance, the thread on the right could first read 2 and then 1, as depicted below.




(Note that although $rf$ is a function from reads to writes, when depicting it as a relation, we point the arrow from the write to the read, because the write is logically before the read in time.)

Such executions will be ruled out by our next model.

## 1.3 Coherence

Coherence adds another component to the executions, the memory order, $mo \subseteq A \times A$, relating write events to the same location.

We say that an execution is consistent with respect the coherence memory model if it satisfies the following axioms:

**ConsistentRF & ConsistentRFdom** as before.

**ConsistentMOord** $mo$ is an irreflexive partial order, i.e.,

$$(\nexists a.\ mo(a,a)) \wedge (\forall a,b,c.\ mo(a,b) \wedge mo(b,c) \implies mo(a,c)$$

**ConsistentMOdom** The memory order relates only writes to the same location:

$$\forall a,b.\ mo(a,b) \implies \exists \ell.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)$$

**ConsistentMOtotal** The memory order is total on writes to the same location:

$$\forall \ell, a, b.\mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a,b) \vee mo(b,a) \vee a = b$$

**CoherenceWW** The memory order cannot contradict the program order:

$$\begin{array}{l} a : \mathsf{W}(\mathtt{x},1) \\ mo \nwarrow \quad \searrow po \\ b : \mathsf{W}(\mathtt{x},2) \end{array} \qquad \nexists a,b.\ po(a,b) \wedge mo(b,a)$$

**CoherenceRW1** We cannot read from the future:

$$\begin{array}{l} a : \mathsf{R}(\mathtt{x},1) \\ rf \nwarrow \quad \searrow po \\ b : \mathsf{W}(\mathtt{x},1) \end{array} \qquad \nexists a,b.\ po(a,b) \wedge rf(a) = b$$

**CoherenceRW2** We cannot read from the very future:

$$\begin{array}{ccc} a : \mathsf{R}(\mathtt{x},2) & \xleftarrow{rf} & \\ \downarrow po & & \\ b : \mathsf{W}(\mathtt{x},1) & \xrightarrow[mo]{} & c : \mathsf{W}(\mathtt{x},2) \end{array} \qquad \nexists a,b,c.\ po(a,b) \wedge mo(b,c) \wedge rf(a) = c$$

**CoherenceWR** We cannot read from an overwritten write:

$$\begin{array}{ccc} a : \mathsf{W}(\mathtt{x},1) & \xrightarrow{mo} & b : \mathsf{W}(\mathtt{x},2) \\ & \searrow^{rf} & \downarrow po \\ & & c : \mathsf{R}(\mathtt{x},1) \end{array} \qquad \nexists a,b,c.\ mo(a,b) \wedge po(b,c) \wedge rf(c) = a$$

**CoherenceRR** Reads from the same location cannot see the write in conflicting order:

$$\begin{array}{ccc} b : \mathsf{W}(\mathtt{x},2) & \xrightarrow{rf} & c : \mathsf{R}(\mathtt{x},2) \\ mo \uparrow & & \downarrow po \\ a : \mathsf{W}(\mathtt{x},1) & \xrightarrow[rf]{} & d : \mathsf{R}(\mathtt{x},1) \end{array} \qquad \begin{array}{l} \nexists a,b,c,d.\ mo(a,b) \wedge po(c,d) \\ \qquad \wedge rf(c) = b \wedge rf(d) = a \end{array}$$

**An alternative presentation of the CoherenceWW axiom** Instead of the (CoherenceWW) axiom, which says that the memory order, $mo$, cannot contradict the program order, $po$, we could instead require $mo$ to include $po$ restricted to writes to the same location. That is,

$$\forall \ell, a, b. po(a,b) \wedge \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a,b)$$

This presentation is equivalent because the program order is total over writes to the same location and the program order is irreflexive.

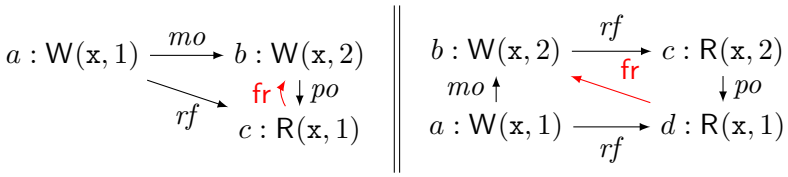
**A more concise presentation of the coherence axioms** While the coherence axioms rule out executions that one would naturally want to rule out, there are too many of them. We can come up with a much more concise axiomatisation by noticing that the CoherenceWW/RW1/RW2 axioms rule out certain cycles in $po \cup mo \cup rf$. In contrast, CoherenceWR/RR axioms do not directly rule out any cycles, but rather disallow reading from overwritten values.

Next, we introduce the auxiliary *from-read* (or *conflict*) relation, defined as follows:

$$\mathsf{fr}(a,b) \stackrel{\mathsf{def}}{=} \exists c.\ rf(a) = c \wedge mo(c,b)$$

We say that $a$ and $b$ are in conflict whenever $a$ reads from an $mo$-earlier write than $b$.

Now, if we draw the draw the fr-edges for the diagrams CoherenceWR/RR diagrams, we notice that there is a cycle in $po \cup \mathsf{fr} \cup rf$.




The alternative presentation of the coherence axioms relies on this observation and requires instead of the previous five coherence axioms the following one:

$$\forall \ell \in \mathsf{Loc}.\ \mathsf{acyclic}(po|_\ell \cup \{(a,b) \mid rf(b) = a\} \cup mo \cup \mathsf{fr})$$

where $X|_\ell \stackrel{\mathsf{def}}{=} \{(a,b) \mid X(a,b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) = \ell\}$ and $\mathsf{acyclic}(X) \stackrel{\mathsf{def}}{=} \nexists a.\ X^+(a,a)$.

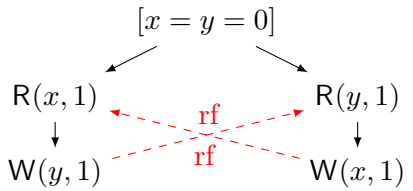
It is clear from the pictures that this new axiom implies the five previous coherence axioms (CoherenceWW/RW1/RW2/WR/RR). One can also show that it is actually equivalent to them.

**Exercise 1.** *Show that the previous set of coherence axioms (WW, RW1, RW2, WR, RR) imply the new acyclicity axiom.*

**Dependency cycle example** Coherence rules out the weird behaviours of our earlier example (e.g., the one that was depicted in §1.2 is ruled out by CoherenceRR), but it does still allow some strange behaviours, as illustrated by the execution below:

```
                    x = y = 0;
if (x.load() == 1)  ||   if (y.load() == 1)
   y.store(1)       ||      x.load()
```

This program has two consistent executions, (1) one where the loads read the initial value 0 and hence the stores are not taken, and (2) another where the loads read the value 1 and therefore the stores happen and produce the ones justifying the loads. This latter execution is depicted below.




Allowed by coherence.

Note that there is a cycle in $po \cup rf$ in this example, but this cycle is not forbidden by the coherence axioms because it involves two different locations.

## 1.4 Release-acquire

Release-acquire consistency is a stronger model than coherence that restricts behaviours such as the one above.

In this model, we use the same executions and axioms as in coherence, and introduce the following two auxiliary definitions:

$$\mathsf{sw}(a, b) \stackrel{\mathsf{def}}{=} rf(b) = a \wedge \mathsf{isrelease}(a) \wedge \mathsf{isacquire}(b)$$
$$\mathsf{hb} \stackrel{\mathsf{def}}{=} (po \cup \mathsf{sw})^+$$

First, we say that two events synchronize if and only if the first is a release write, and the second is an acquire read that reads from the release write. (In the basic release-acquire model, all reads are 'acquire' reads, and all writes are 'release' writes. In more advanced versions of the model, there are also memory accesses that are neither releases nor acquires.)

Second, we say that an event $a$ happens before an event $b$, denoted as $\mathsf{hb}(a, b)$, if there is a path of program order ($po$) and synchronisation-with edges ($\mathsf{sw}$) from $a$ to $b$.

Then, we adapt the coherence axioms to use the happens before order, $\mathsf{hb}$, instead of the program order, $po$. This gives us the following axioms:

**ConsistentRF & ConsistentRFdom** as before.

**ConsistentMOord, ConsistentMOdom, ConsistentMOtotal** as before.

**Coherence** The per-location coherence axiom (here, presented in its concise formulation):

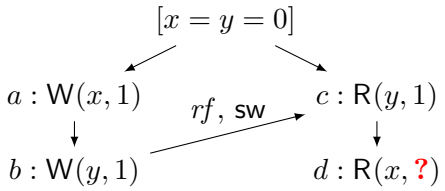
$$\mathsf{acyclic}(\mathsf{hb}|_\ell \cup rf \cup mo \cup \mathsf{fr})$$

Note that the strengthened coherence axiom rules out more executions than the original axiom with just $po|_\ell$. For instance, the execution with the dependency cycle depicted at the end of Section 1.3 is disallowed because the $rf$-edges induce sw synchronization edges, leading to hb-cycle.

**Message passing example** Release-acquire is fairly easy for programming because it allows the following common sychronization pattern known as *message passing* (MP). The program is the following:

```
                x = y = 0;
x.store(1);   ||   if (y.load() == 1)
y.store(1)    ||      x.load()
```

There are two threads, a producer and a consumer. The producer writes to $x$ and then sends a signal that it has finished by writing to $y$. The consumer sees the signal and then proceeds to access $x$. With release-acquire, we can know that the read of $x$ (if it happens) returns the value written by the producer and not the original value.

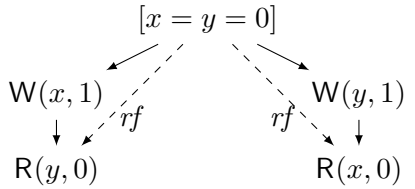



Because of coherence, $d$ can only read from $a$. Therefore, ? = 1.

**Store buffering example** It is worth pointing out that release-acquire is weaker than sequential consistency (i.e., the usual interleaving semantics), as illustrated by the *store buffering* (SB) example below. The program is the following:

```
                x = y = 0;
x.store(1);   ||   y.store(1);
y.load()      ||   x.load()
```

Under release-acquire, it is possible for both loads to return 0, as shown in the following execution:



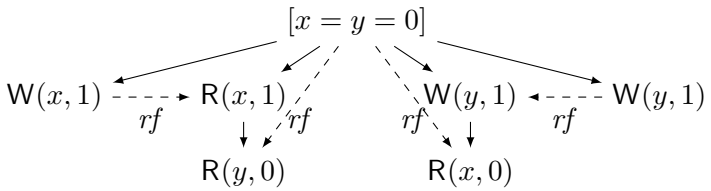

Allowed by Rel-Acq (and TSO)

This relaxed behaviour, however, cannot occur under any interleaving of the two threads. The reason is that in any interleaving, one of the threads has to start before the other, and therefore its write will happen before the corresponding read of the other thread.

**IRIW example** Release-acquire also allows two threads to observe independent writes happen in different orders, as illustrated by the following *independent-reads independent-writes* (IRIW) example. The program is:

```
                        x = y = 0;
x.store(1)  ||  x.load()==1  ||  y.load()==1  ||  y.store(1)
            ||  y.load()==0  ||  x.load()==0  ||
```

Note that this program is essentially the store buffering example, where we have pushed the stores to separate threads and introduced a $rf$ edge to connect them. The following execution is possible under release-acquire:




## 1.5 Sequential consistency (SC)

Sequential consistency is the strongest model expressible in this framework and insists that there is a total ordering on all events of an execution, which corresponds to the interleaving of threads.

Formally, in sequential consistency, we add two components to the executions:

- The reads-from map, $rf : A \rightharpoonup A$, mapping read events to the write events that they read.
- The sequentially consistent order, $sc \subseteq A \times A$.

We have the following axioms:

**ConsistentRF, ConsistentRFdom** as before.

**ConsistentSC** The $sc$ relation is an irreflexive total order on $A$.

**ConsistentSCpo** $sc$ contains the program order, $po$:

$$\forall a, b.\ po(a, b) \implies sc(a, b)$$

**ReadSC** Reads read from the immediately preceding write to same location in the global $sc$ order.

$$\forall a, b.\ rf(a) = b \implies sc(b, a) \wedge (\nexists c.\ sc(b, c) \wedge sc(c, a) \wedge \mathsf{iswrite}_{\mathsf{loc}(a)}(c))$$

Note that these axioms rule out the relaxed behaviours for both the SB and the IRIW examples. In this model, we do not need the memory order and the coherence axioms.

**Exercise 2.** *Show that coherence is strictly weaker than sequential consistency. That is: (1) give an execution that is consistent under Coherence, but not under SC; and (2) prove that every consistent execution under SC is also consistent under Coherence.*
Hint: *For the second part, show that* $\mathsf{Consistent}_{\mathrm{SC}}(A, lab, po, rf, sc)$ *implies* $\mathsf{Consistent}_{\mathrm{Coherence}}(A, lab, po, rf, \bigcup_\ell \{(a, b) \in sc \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\})$.

## 1.6 Total store order (TSO)

The axiomatisation of TSO is similar to coherence, in that we have the $rf$ map and the memory order $mo$, but here the memory order can relate more events that just writes to the same location.

We say that an execution is consistent with respect to the TSO memory model, $\mathsf{Consistent}_{\mathrm{TSO}}(A, lab, po, rf, mo)$, if it satisfies the following axioms:

**ConsistentRF, ConsistentRFdom** as before.

**OrderMO** $mo$ is an irreflexive partial order (i.e., irreflexive & transitive)

**TotalMO** $mo$ is a total order on writes to the same location.

$$\forall \ell, a, b.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a, b) \vee mo(b, a) \vee a = b$$

**OrderWW** Writes in program order are also ordered by $mo$

$$\forall a, b.\ po(a, b) \wedge \mathsf{iswrite}(a) \wedge \mathsf{iswrite}(b) \implies mo(a, b)$$

**OrderRx** Every event after a read in program order is also after it in memory order:

$$\forall a, b.\ po(a, b) \wedge \mathsf{isread}(a) \implies mo(a, b)$$

**ReadTSO** Reads read from a $po \cup mo$ maximal write preceding them.

$$\forall a, b.\ rf(a) = b \implies \mathsf{mopo}(b, a) \wedge \nexists c.\ \mathsf{iswrite}_{\mathsf{loc}(a)}(c) \wedge \mathsf{mopo}(c, a) \wedge mo(b, c)$$

where $\mathsf{mopo} \stackrel{\mathsf{def}}{=} mo \cup po$.

# 2 The C11 memory model

The C11 memory model was introduced by the 2011 revisions of the C and C++ standards, and was formalized by [Batty et al., 2011].

It is a more complex model than the ones that we have seen so far because it provides several kinds of memory accesses:

- non-atomic,
- relaxed atomic,
- acquire atomic reads,
- release atomic writes, and
- sequentially consistent (SC) atomic

each providing different consistency guarantees. On the one end of the spectrum, races on non-atomic accesses result in completely undefined behaviour (they are treated as programming errors); on the other end, SC-atomic accesses are globally synchronized. The guarantees provided by relaxed, acquire, and release accesses lie somewhere in between: different threads can observe them happening in different orders.

More concretely, relaxed atomic accesses ensure coherence, release writes and acquire reads follow the release-acquire memory model, while SC atomics ensure sequential consistency.

The reason for having all these kinds of accesses is that they map differently to the various common architectures, and have very different implementation costs. Non-atomic and relaxed atomic accesses are generally rather cheap as they correspond to plain machine loads and stores, and may be reordered by the compiler and/or by an out-of-order execution unit. At the other end of the spectrum, SC accesses are very expensive because their implementation involves a full memory barrier (`mfence` on x86, `sync` on Power).

The cost of acquire and release accesses depends a lot on the architecture. On x86, they are compiled down to plain reads and writes and are therefore cheap. On PowerPC and ARM, the cost is somewhat higher as they induce a memory barrier, but of a weaker kind than full memory barriers (`lwsync` or `isync`).

The model also provides some more advanced features:

- read-modify-write instructions (such as CAS);
- acquire, release, and SC fences; and
- 'consume' atomic reads.

that we will not cover here.

In presenting the C11 memory model, we will also simplify some of its axioms. C11 executions are tuples of the form $(A, lab, po, rf, mo, sc)$ containing all the components we have seen before, namely:

- The set of events, $A \subseteq \mathbb{N}$
- The labelling function, $lab : A \to \mathsf{Label}$, with labels given by the following grammar:

$$\mathsf{Label} ::= \mathsf{Skip} \mid \mathsf{R}_{typ}(\ell, v) \mid \mathsf{W}_{typ}(\ell, v)$$

  where $\ell \in \mathsf{Loc}, v \in \mathsf{Val}$ and $typ \in \{\mathrm{na}, \mathrm{rlx}, \mathrm{acq}, \mathrm{rel}, \mathrm{sc}\}$.
- The program order, $po \subseteq A \times A$. (This is called *sequenced before* in C11 terminology.)
- The reads-from map, $rf : A \rightharpoonup A$, mapping read events to the write events that they read.
- The memory order, $mo \subseteq A \times A$, relating write events to the same location.
- The sequentially consistent order, $sc \subseteq A \times A$, relating all events that are of "sequential consistent" type.

Next, C11 defines synchronizes-with and happens before as in the release-acquire model:

$$\begin{aligned}
\mathsf{isrelease}(a) &\stackrel{\mathsf{def}}{=} \exists \ell, v.\ lab(a) \in \{\mathsf{W}_{\mathsf{rel}}(\ell, v), \mathsf{W}_{\mathsf{sc}}(\ell, v)\} \\
\mathsf{isacquire}(a) &\stackrel{\mathsf{def}}{=} \exists \ell, v.\ lab(a) \in \{\mathsf{R}_{\mathsf{acq}}(\ell, v), \mathsf{R}_{\mathsf{sc}}(\ell, v)\} \\
\mathsf{sw}(a, b) &\stackrel{\mathsf{def}}{=} rf(b) = a \wedge \mathsf{isrelease}(a) \wedge \mathsf{isacquire}(b) \\
\mathsf{hb} &\stackrel{\mathsf{def}}{=} (po \cup \mathsf{sw})^+
\end{aligned}$$

Just note that SC accesses are also treated as release and acquire ones.

The axioms of C11 are as follows:

**ConsistentRF** (as before) The reads-from map relates reads and writes with the same locations and values.

$$\forall a, b.\ rf(a) = b \implies \exists \ell, v.\ \mathsf{isread}_{\ell,v}(a) \wedge \mathsf{iswrite}_{\ell,v}(b)$$

**ConsistentRFdom** (updated) The reads-from map is defined for all *initialized* reads. We say that a read is initialized if there exists a write to the same location that happens before it.

$$\forall a, b, \ell.\ \mathsf{isread}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \wedge \mathsf{hb}(b, a) \implies \exists c.\ rf(a) = c$$

(In our exaples, we will generally assume that all reads are initialized, in which case the axiom can be simplified to the one in Section 1.2.)

**ConsistentMOord** (as before) $mo$ is an irreflexive partial order, i.e.,

$$(\nexists a.\ mo(a, a)) \wedge (\forall a, b, c.\ mo(a, b) \wedge mo(b, c) \implies mo(a, c)$$

**ConsistentMOdom** (as before) The memory order relates only writes to the same location:

$$\forall a, b.\ mo(a, b) \implies \exists \ell.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)$$

**ConsistentMOtotal** (as before) The memory order is total on writes to the same location:

$$\forall \ell, a, b.\mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a, b) \vee mo(b, a) \vee a = b$$

**ConsistentMOhb** (new, but redundant) The memory order includes happens before, when restricted to writes to the same location.

$$\forall \ell, a, b.\mathsf{hb}(a, b) \wedge \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b) \implies mo(a, b)$$

As we have mentioned in Section 1.3, this axiom is equivalent to the CoherenceWW one, and is therefore redundant.

**ConsistentSCord** $sc$ is an irreflexive partial order, i.e.,

$$(\nexists a.\ sc(a, a)) \wedge (\forall a, b, c.\ sc(a, b) \wedge sc(b, c) \implies sc(a, c)$$

**ConsistentSCdom** The SC order relates only SC events:

$$\forall a, b.\ sc(a, b) \implies \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)$$

**ConsistentSCtotal** The SC order is total on SC events:

$$\forall a, b.\ \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b) \implies sc(a, b) \vee sc(b, a) \vee a = b$$

**ConsistentSCmo** $sc$ contains the memory order, $mo$, restricted to SC events:

$$\forall a, b.\ mo(a, b) \wedge \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b) \implies sc(a, b)$$

**ConsistentSChb** $sc$ contains the happens before order, $\mathsf{hb}$, restricted to SC events:

$$\forall a, b.\ sc(a,b) \wedge \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b) \implies sc(a,b)$$

**ReadSC** Sequential consistent reads can read from either from the immediately preceding SC write to same location in the global $sc$ order, or from a non-SC write.

$$\forall a, b.\ rf(a) = b \wedge \mathsf{isSeqCst}(a) \implies \mathsf{isc}(b,a) \vee \neg\mathsf{isSeqCst}(b)$$

where

$$\mathsf{isc}(a,b) \stackrel{\mathsf{def}}{=} sc(a,b) \wedge \nexists c.\ sc(a,c) \wedge sc(c,b) \wedge \mathsf{iswrite}_{\mathsf{loc}(b)}(c)$$

(The standard also says that in case the SC read reads from a non-SC write, that write should not happen before the immediately preceding SC write to the same location of the read. This condition is, however, problematic. We should either say that it does not happen before any preceding SC write to the same location, or just drop the requirement completely, as done here.)

**ReadNA** (new) Non-atomic reads can only read from happens-before-earlier writes:

$$\forall a, b.\ rf(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a,b)$$

(This axiom is also problematic, because it is the only one where $\mathsf{hb}$ appears in a positive position (e.g., not under a negation or at the LHS of an implication). This gives the counter-intuitive property that by adding more synchronization (that is, increasing $\mathsf{hb}$), we can possibly get more consistent behaviours.)

**CoherenceWW/RW1/RW2/WR/RR** as before.

**IrreflexiveHB** Happens before is irreflexive: $\nexists x.\ \mathsf{hb}(x,x)$.

As we have seen in Section 1.3, we can replace the coherence axioms and the IrreflexiveHB axiom with the following equivalent one:

$$\mathsf{acyclic}(\mathsf{hb}|_\ell \cup \{(a,b) \mid rf(b) = a\} \cup mo \cup \mathsf{fr}) \qquad \text{(Coherence)}$$

where

$$\begin{aligned} \mathsf{fr}(a,b) &\stackrel{\mathsf{def}}{=} \exists c.\ rf(a) = c \wedge mo(c,b) \\ \mathsf{hb}|_\ell &\stackrel{\mathsf{def}}{=} \{(a,b) \mid \mathsf{hb}(a,b) \wedge \mathsf{loc}(a) = \mathsf{loc}(b) = \ell\}\,. \end{aligned}$$

**Data races** We say that two memory access are *conflicting* if they both access the same location and at least one of them is a write. Two distinct accesses are *concurrent* if they are not ordered by the happens before order. A *data race* occurs whenever there are two concurrent conflicting accesses, at least one of which is non-atomic. That is,

$$\mathsf{race}(a,b) \stackrel{\mathsf{def}}{=} \begin{pmatrix} a \neq b \wedge \neg\mathsf{hb}(a,b) \wedge \neg\mathsf{hb}(b,a) \\ \wedge(\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \\ \wedge(\mathsf{iswrite}(a) \vee \mathsf{iswrite}(b)) \\ \wedge\exists \ell.\ \mathsf{loc}(a) = \mathsf{loc}(b) = \ell \end{pmatrix}$$

An execution is *data race free* if it does not have any races: i.e., $\nexists a, b.\ \mathsf{race}(a,b)$.

According to C11, the semantics of a program is calculated by considering the set of all its consistent executions. If all the consistent executions are race free, then the semantics of the program is exactly that set of consistent executions. If, however, there is a consistent racy execution of the program, then the program is deemed incorrect and has "undefined semantics." Undefinedness can be modelled as the set of all possible executions, whether consistent or not, whether corresponding to the program text or not. This style of defining memory models, where programs with races are given undefined semantics, is known as DRF (standing for data race free).

## References

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.