View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions

Jean Pichon-Pharabod Peter Sewell

University of Cambridge, United Kingdom

first.last@cl.cam.ac.uk

Abstract

Despite much research on concurrent programming languages, especially for Java and C/C++, we still do not have a satisfactory definition of their semantics, one that admits all common optimisations without also admitting undesired behaviour. Especially problematic are the "thin-air" examples involving high-performance concurrent accesses, such as C/C++11 relaxed atomics. The C/C++11 model is in a per-candidate-execution style, and previous work has identified a tension between that and the fact that compiler optimisations do not operate over single candidate executions in isolation; rather, they operate over syntactic representations that represent all executions.

In this paper we propose a novel approach that circumvents this difficulty. We define a concurrency semantics for a core calculus, including relaxed-atomic and non-atomic accesses, and locks, that admits a wide range of optimisation while still forbidding the classic thin-air examples. It also addresses other problems relating to undefined behaviour.

The basic idea is to use an event-structure representation of the current state of each thread, capturing all of its potential executions, and to permit interleaving of execution and transformation steps over that to reflect optimisation (possibly dynamic) of the code. These are combined with a non-multi-copy-atomic storage subsystem, to reflect common hardware behaviour.

The semantics is defined in a mechanised and executable form, and designed to be implementable above current relaxed hardware and strong enough to support the programming idioms that C/C++11 does for this fragment. It offers a potential way forward for concurrent programming language semantics, beyond the current C/C++11 and Java models.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Concurrency; Relaxed memory models; C/C++

1. Introduction

Batty et al. [6] note that:

> Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the concurrency semantics of any general-purpose high-level language that includes high-performance shared-memory concurrency primitives. This is a major open problem for programming language semantics.

The basic problem is how we can define semantics for such languages (including C, C++, and Java) that provides useful guarantees for programmers while permitting optimisation. Classically, one might imagine having a fixed language semantics and adopting only optimisations that are sound with respect to it. The situation here is inverted: mainstream compiler and hardware implementations have accumulated a range of optimisations, that are sound with respect to conventional sequential execution models but which have observable effects in concurrent contexts, and it seems impractical to significantly limit the commonly accepted optimisations. Instead, we have to identify an envelope around them that is tight enough to support concurrent programming.

Neither of the main previous attempts succeeds in this. The Java Memory Model [17] is unsound with respect to standard compiler optimisations [11, 25]. The C/C++11 model [1, 3, 7, 8] is arguably the current state of the art, and gives the "right" behaviour in many cases, but it permits too much behaviour in others. At the heart of the problem are the "thin-air" examples for concurrent high-performance accesses, recalled in §2.1, in which values appear out of nowhere [6, 9, 17, 25].

Those thin-air executions are not thought to occur in practice, with any combination of current compiler and hardware optimisations, but excluding them without also excluding important optimisations has not been previously been achieved.

A further concern for C and C++, also highlighted by Batty et al. [6], is the interaction between undefined behaviour and relaxed memory, which we recall in §2.2.

Contribution We propose a new approach to the definition of concurrent shared-memory programming language semantics that addresses both of these problems. We develop it in the simplest possible setting: a core calculus featuring relaxed and non-atomic accesses and locks. Our semantics forbids the classic thin-air examples and gives the desired behaviour on the relevant Java causality test cases [21]. It is designed to be weak enough to form an envelope around all reasonable behaviour induced by hardware and compiler optimisations, so that implementing relaxed accesses above the ARM or IBM Power architectures (the most relaxed current mainstream hardware) does not require any memory barriers or other synchronisation, while being strong enough to support

C/C++11-style programming. It moreover handles undefined behaviour in concurrent contexts.

After recalling the problems in more detail (§2), we begin with an informal introduction to our semantics (§3), before explaining it more precisely (§4 and §5). Our definitions are mechanised in Lem [19] and available online (`http://www.cl.cam.ac.uk/~pes20/popl16-thinair`), and we use Lem to generate executable OCaml code to make a tool allowing one to explore the semantics on small examples. We discuss how the semantics prevents out-of-thin-air executions (§6), how it deals with undefined behaviour (§7), the reasons why it should be efficiently implementable above ARM/Power (§8), and its relationship to C/C++11 (§9). Finally we discuss related work and conclude (§10,11).

2. Recalling the Problems

2.1 The Thin-Air Problem

As acknowledged by the C++ standard [7, 23.9p9], in trying to define an envelope around all reasonable optimisations, the C/C++ memory model also admits undesirable executions where values seem to appear out of thin air:

> [Note: The requirements do allow `r1 == r2 == 42` in the following example, with x and y initially zero:
>
> ```
> // Thread 1:
> r1 = x.load(memory_order_relaxed);
> if (r1 == 42) y.store(r1, memory_order_relaxed);
> // Thread 2:
> r2 = y.load(memory_order_relaxed);
> if (r2 == 42) x.store(42, memory_order_relaxed);
> ```
>
> However, implementations should not allow such behavior.
> – end note]

Similar issues arose in the first Java Memory Model design, in a slightly different form [17]. C11 and C++11 have nonatomic and relaxed accesses, both of which are intended to be implementable without memory barriers or other synchronisation; concurrent relaxed access to the same location is permitted while concurrent nonatomic access gives wholly undefined behaviour (to let optimisers assume the latter does not occur). Java normal (non-volatile) accesses are similarly intended to be implementable with just the underlying hardware plain loads and stores; programmers are not supposed to make non-synchronised concurrent use of these, but, to provide safety guarantees even in the presence of arbitrary code, the semantics must forbid the forging of pointers.

There is no precise definition of what thin-air behaviour is—if there were, it could simply be forbidden by fiat, and the problem would be solved. Rather, there are a few known litmus tests (like the one above) where certain outcomes are undesirable and do not appear in practice (as the result of hardware and compiler optimisations). The problem is to draw a fine line between those undesirable outcomes and other very similar litmus tests which important optimisations do exhibit and which therefore must be admitted.

Per-candidate-execution semantics does not suffice A common approach to relaxed-memory semantics, and that followed by C/C++11, is to define what is called an *axiomatic* memory model. One defines a notion of candidate execution, each consisting of a set of memory actions and basic relations over them (such as program order, reads-from, coherence, etc.), and a consistency predicate that picks out the candidate executions that are allowed by the semantics (e.g. including a check that some happens-before relation, derived from the basic relations, is acyclic). The semantics of a program is taken to be the set of all consistent executions that are compatible with some control-flow unfolding of the program, or, in some semantics, that set modulo the existence of data races.

However, as observed by Batty et al. [6], there are programs which share a particular candidate execution where that execution should be allowed for one but not for the other. Consider the following (the second is just the example above in a more concise syntax):

Example 1

`r1 = load`$_{rlx}$`(x);`
`if (r1 == 42)`
`  store`$_{rlx}$`(y,r1)`
	`r2 = load`$_{rlx}$`(y);`
	`if (r2 == 42)`
	`  store`$_{rlx}$`(x,42)`
	`else`
	`  store`$_{rlx}$`(x,42)`

Example 2

`r1 = load`$_{rlx}$`(x);`
`if (r1 == 42)`
`  store`$_{rlx}$`(y,r1)`
	`r2 = load`$_{rlx}$`(y);`
	`if (r2 == 42)`
	`  store`$_{rlx}$`(x,42)`

As there is a write to x in both branches of the second thread of Example 1, the conditional can be collapsed by compiler optimisation to yield

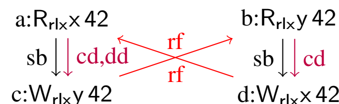
Example 3

`r1 = load`$_{rlx}$`(x);`
`if (r1 == 42)`
`  store`$_{rlx}$`(y,r1)`
	`r2 = load`$_{rlx}$`(y);`
	`store`$_{rlx}$`(x,42)`

in which both threads can read 42, because the read and the write of the second thread can be reordered (by compiler or hardware), either thread-locally or when propagating between the threads. This happens in practice and so should be allowed by the semantics.

On the other hand, for Example 2, this outcome does not appear in practice, and is highly undesirable, as it breaks elementary programming and reasoning principles [5, 9, 27].

In the C/C++11 semantics, the two share the candidate execution below (here "sb" stands for "sequenced-before", that is, program order, "cd" stands for (syntactic) control dependency, "dd" stands for (syntactic) data dependency, and "rf" indicates where the reads read from).



Dependencies As the example above shows, syntactic data and control dependencies do not provide enough information to draw the distinction between desirable and undesirable outcomes. In retrospect, this is not surprising: programmers expect to be able to interchange control flow and dependencies without changing program outcome, and compilers do that freely, sometimes removing syntactic dependencies.

Merging Moreover, programming languages allow memory actions to be merged. For example, for the non-atomic version of the below, compilers can and do merge the reads of y. The same optimisation for relaxed atomics, turning it into the second thread of Example 1, has been proposed [2]:

Example 4

```
r2 = load_rlx(y);
if (r2 == 42) {
  r3 = load_rlx(y);
  store_rlx(x,r3)
} else {
  store_rlx(x,42)
}
```

2.2 The Concurrent Undefined Behaviour Problem

If a language features undefined behaviour, as C and C++ do, then its memory model needs to be able to express it, to determine whether it is triggered. As previously pointed out [6, §7], there is a mismatch in the C/C++ standard: the thread-local semantics, which is described operationally, assumes that there is some form of execution order that makes it possible to tell whether a point of the program has been reached, and therefore undefined behaviour is triggered (though if undefined behaviour *is* triggered, it makes the whole program undefined, not merely the execution from that point). However, the candidate executions of the axiomatic memory model do not have such a notion, and because they are candidate *complete* executions, rather than being built incrementally, it is far from obvious how it could be included. We illustrate the problem and explain how our memory model accounts for it in Section 7.

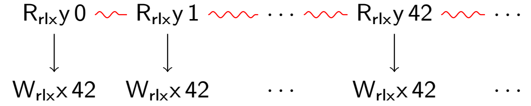
3. Our Approach, Informally

3.1 Thread State

As we have seen, candidate executions, taken individually, fail to capture the difference between Example 1 and Example 2. Looking at the compiler optimisation that collapses the conditional on the second thread of Example 1, we can see that it is important for the compiler to know that the write of `x = 42` occurs in both control-flow paths of the conditional, or, more accurately, that it occurs irrespective of the value of the preceding read of `y`. This is not a per-candidate-execution property. However, a set of candidate executions almosts contain the relevant information, that is, whether certain actions have a semantic dependency upon others. The missing ingredient is a way to indicate how the different candidate executions relate to each other, and the points at which they diverge into incompatible actions.

This structure, of a set of "events" equipped with some causal order (C/C++11 "sequence-before") and a conflict relationship, obeying certain sanity conditions, is called an event structure [20]. Event structures were introduced as 1979 as a foundational framework for "true concurrency" (that is, non-interleaving concurrency), with conflict-free subsets of the event structure corresponding to global states of a concurrent program. We use event structures in a different way, to describe "true" concurrency, in the sense of that found in mainstream relaxed-shared-memory programming languages. Here the current state and potential future executions of each individual thread will be represented by an event structure, the thread can take transitions that mutate that, and the whole system has a state and transitions that are the composition of those for each thread with a storage subsystem.

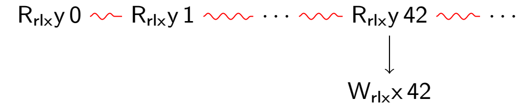
The event structure of the second thread of Example 3, below, has one read event for each value that the read of `y` might read, and all these read events are in conflict with each other (represented by the wavy red line), as only one of them can happen. Below each of these read events, there is the subsequent write event of 42 to `y` (program order is represented by black arrows):



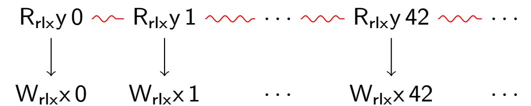
Intuitively, it is the presence of the $W_{rlx} x\, 42$ in *all* the branches of this[1] that means it can be reordered before the read of `y`, as it is semantically independent of the value read.

Note that by moving to an event structure representation we abstract over the details of how the control flow is expressed, and the event structure of the second thread of Example 1 is the same as that above.

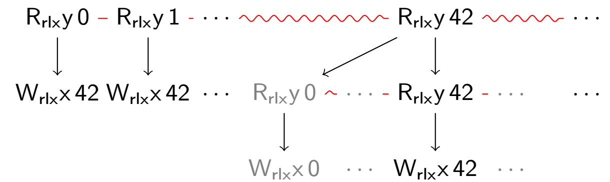
Looking at the event structure of the second thread of Example 2 (below), the difference with Example 1, namely that the write occurs only when 42 is read, is immediate:



The difference with the second thread of Example 1 (below), namely that the value to write semantically depends on the value read, is also immediate.



The similarity with Example 4 below is also clear: we have to coalesce the two reads of 42, and discard the greyed out events, to make them the same:



By using event structures to represent the states of the threads, we make the relevant differences manifest in a way the dynamic semantics can easily exploit.

3.2 Dynamics

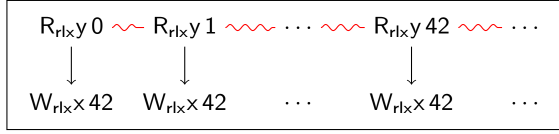
The goal of the memory model is to define an envelope around all "reasonable" optimisations, taking those done by current compilers to be a de facto lower bound on what should be deemed reasonable.

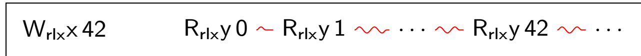
But compiler optimisations as they are normally considered are highly complex syntactic transformations, typically over some intermediate representations, relying on various static analysis properties, and changing over time, and there are hundreds of passes in current compilers. A semantics should not (and cannot) describe all that explicitly. Instead, we define an envelope based on steps that account for the elementary changes of memory actions that syntactic optimisation passes induce, broadly following Ševčík [29]: reordering (in fact, *de*ordering) and merging pairs of read and write memory actions.

Deordering Optimisations that involve code motion rely on the fact that although the source program has a syntactic order, this order is in these cases semantically irrelevant. To account for this,

[1] More precisely: in the descendants of each member of the set of incompatible read events.

we include deordering steps. For example, as there is a write of 42 in each "branch" of the read in the first diagram above, there is no dependency between the read and the write. The event structure makes this immediate, while it can be obscured by control flow in the syntax. Because the write does not depend on the read, and the two are at different locations, the order between the two can be removed to yield a new event structure, in a thread-local transition as below:

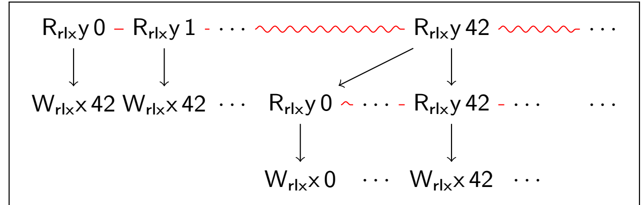


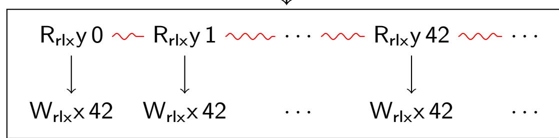


The reasoning underpinning this deordering is a semantic counterpart of what a compiler has to approximate by abstract interpretation on the syntax.

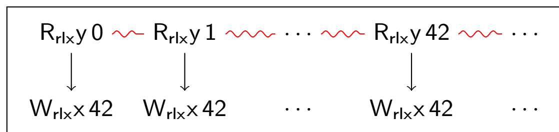
Merging Optimisations that involve removing memory instructions, like common subexpression elimination, rely on determining that the effect of one instruction can be subsumed by that of another. For example, in Example 4, the effect of the second read is subsumed by that of the first, so the second can be eliminated and replaced by `r3 = r2`.

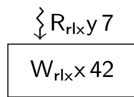
To account for this, the memory model includes merging steps in which a single event is merged into subsuming events. For example, the event structure of the second thread of Example 4 can be turned into that of the second thread of Example 1, by merging the second $\mathsf{R}_{\mathsf{rlx}}\,\mathsf{y}\,42$ into the first:





Execution Steps The event structure for a thread can be mutated by optimisation steps, but it can also take execution steps, to actually perform memory actions, for any of its events that are not program-order-after any other events (the possible read values will be constrained further by the overall state, as we describe in §5.3, but for now we speak of the thread in isolation). For example, in the initial state of the second thread of Example 1 or Example 3, any of the reads can be executed, e.g. with transitions like that below:





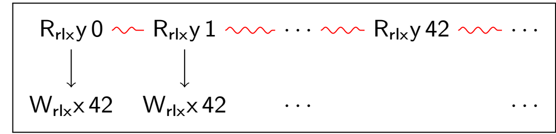
Value-range speculation steps Optimisations are complemented by analyses which extend their applicability. For example, if analysis shows that in the program below, the read of `x` always reads a value different from 42, then the conditional can be collapsed to just $\mathtt{store}_{\mathtt{rlx}}\mathtt{(y,42)}$.

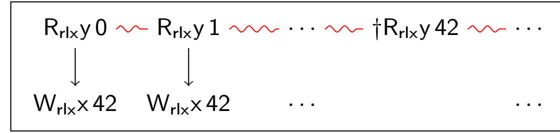
```
r1 = load_rlx(x);      ||
if (r1 != 42)          ||   ...
  store_rlx(y,42)      ||
```

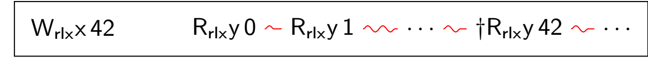
This allows more behaviour, as the write to `y` can then execute out of order with the read. However, the collapse of the conditional is only valid if that extra behaviour does not invalidate the result of the analysis. For example, if the rest of the program were the second thread of Example 2, this speculation would be incorrect.

To account for the effect of these analyses, the memory model includes value-range speculation steps, in which a read of a certain value is speculated to be impossible, and marked by a †; the other steps of the memory model ignore reads that have been speculated to be impossible. This speculation can be done only if the program then cannot actually execute the read of that value (this depends on the behaviour of the program as a whole, as defined below, not just the thread-local transition system that we have focussed on so far).

For the example program above, speculating that the read cannot read 42 allows deordering the write.







This mechanism allows the memory model to account for the extra optimisations enabled by inter-thread analyses, like alias analysis. But it does come at a combinatorial price, creating a large number of potential executions even for small programs; it would be desirable to limit it if possible.

Interleaving of optimisation and execution Some optimisations only become possible when particular values become known or constrained: if a read event (recalling that this is a dynamic occurrence of a read, not a source-language syntactic occurrence) is guaranteed not to read certain values, the corresponding branches of the program do not need to be considered, and this can enable more deordering or merging steps.

This means that we cannot separate an initial optimisation phase from program execution, and instead we allow arbitrary interleaving of execution and optimisation steps. This also accomodates implementations that actually do that in practice, for example JIT compilation steps that are aware of the current value environment.

We also cannot eagerly normalise threads to "fully optimised" forms. For example, Example 4 can have two radically different executions: either the two reads are merged, which allows the write can be reordered and executed before the read, or the two reads can read different values.

Relaxed atomics and non-atomics Relaxed atomics and non-atomic accesses are deordered in the same way. However, non-

atomic accesses are treated more aggressively by mergings than relaxed atomic accesses. For example, if the accesses in the program below were atomic, it would be unsound with respect to the expected behaviour of locks to remove the first write. Indeed, the write of 1 could be hiding a previous write, and removing it would make that previous write visible. However, as there is a non-atomic access to the same location outside of the critical section ended by `unlock l`, no race-free program can observe the write of 1 happening. Therefore, this transformation is sound for non-atomic accesses.

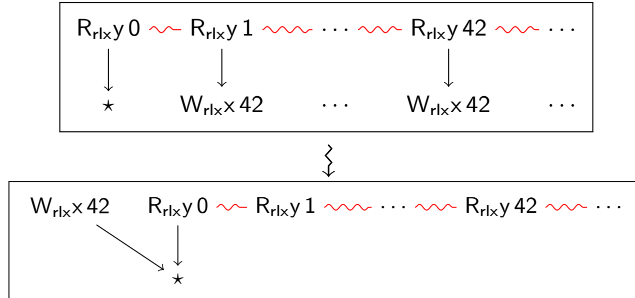
```
store_na(x,1);
unlock l;
store_na(x,2)
```

Some steps are accounted for implicitly Not all the computation and optimisation steps that runtimes and compilers do need to be accounted for explicitly. Some do not change the event structure of a thread at all, e.g. replacing $2 + 2$ by 4. Others may radically change its memory accesses, e.g. simplifying a thread-local computation, but, if this involves only variables that (during this part of the computation) are private to the thread, it should not affect whether other optimisations across that computation are permitted.

Undefined behaviour Undefined behaviour represents partiality of the language specification. When a program point exhibits undefined behaviour (which we represent with a $\star$-labelled event), the semantics of the language does not have to consider what happens at that point. For example, when deordering the write in

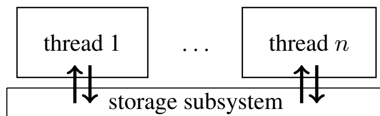
```
r1 = load_rlx(x);
if (r1 == 0)
  undef
else
  store_rlx(y,42)
```

the branch that contains undefined behaviour does not have to be considered (in the same way that a compiler can assume that the program being compiled does not exhibit undefined behaviour):



When a $\star$-labelled event is executed, it means that the undefined behaviour can be triggered, so the whole program has undefined behaviour.

Storage subsystem Communication between threads is mediated by a storage subsystem (as below) that takes care of propagation effects. The storage subsystem interacts with threads by synchronising on read, write, lock, and unlock actions.



The storage subsystem is based on the operational Power storage subsystem of Sarkar et al. [23], providing a non-multi-copy-atomic memory that guarantees coherence but little more. Threads can send writes to it, which it then propagates to each thread individually (this is non-multi-copy-atomicity). Writes at different locations can overtake each other, but writes at the same location remain in order. When a thread requests a read from the storage subsystem, the read is satisfied by the writes that have propagated to the thread but are not hidden by other writes. This is the point where read values are constrained in our memory model. Moreover, the storage subsystem maintains "coherence", a total order over writes at the same location that is respected by reads.

Steps of the semantics Summarising the steps of the semantics, given in more detail in §5, we have:

- thread execution, synchronising with the storage subsystem to do a read, write, lock, or unlock
- thread-local deordering:
  - of non-reads
  - of reads
- thread-local merging:
  - forwards: Read after Read, Read after Write, Write after Read, and Write after Write
  - backwards: Overwritten Write
- value-range speculation

4. Formal Setup

4.1 Language

We consider a minimal calculus featuring relaxed-atomic and non-atomic reads and writes, locks, and a thread-local undefined behaviour trigger.

```
p  ::=                          program
     | ss || ... || ss              parallel threads
s  ::=                          statement
     | r = load_mo(x)               read
     | store_mo(x,e)                write
     | if (r == v) ss else ss       conditional
     | r = e                        register assignment
     | lock l                       lock
     | unlock l                     unlock
     | undef                        undefined behaviour
ss ::=                          statements
     | s ; .. ; s                   sequential composition
     | { ss }                       block
mo ::=                          memory orders
     | na                           non-atomic
     | rlx                          relaxed
e  ::=                          pure expressions
     | v                            constant
     | r                            register
     | e + e                        sum
```

Variables `x`, `y`, ..., are shared memory locations, and actions on them induce memory actions, whereas `r1`, `r2`, ..., are thread-local register variables which have no effect on memory. This distinction is convenient for examples, allowing some computation without excessive numbers of events.

Undefined behaviour In C/C++11, in addition to memory errors, undefined behaviour can be induced by some operations, for example division by zero. In our calculus, for clarity, we separate thread-local undefined behaviour from other operations, with an `undef` statement that signals undefined behaviour.

Location typing As in the formalised C/C++ memory model, for simplicity we assume a location typing into atomic and non-atomic locations.

Finite domains and loops We restrict variables and registers to finite domains, both to match normal implementations and because, with infinite domains, deordering might be possible only after an infinite number of elementary steps. Lifting this limitation could be delicate: we want "horizontal" infinity (applying a transformation in the infinitely many branches of a read), but not "vertical" infinity (unreachable code at the end of an infinite loop becoming reachable). We also omit loops. This is not for any fundamental reason, but rather to keep the semantics straightforwardly executable.

4.2 Memory Actions

The memory actions of our language, used to label events in the threadwise event structures and for synchronisation with the storage subsystem, are as expected for the language, with the addition of "dead reads", marked with a '†', as explained in Section 5.8:

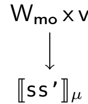
a ::=
| $\mathsf{R_{mo}\,x\,v}$ read
| $\mathsf{W_{mo}\,x\,v}$ write
| L l lock
| U l unlock
| $\star$ undefined behaviour
| $\dagger\mathsf{R_{mo}\,x\,v}$ dead read

4.3 Construction of the Initial Event Structure

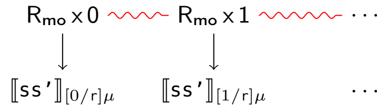
The construction of the initial event structure of a thread is a straightforward compositional function (the natural adaptation of the calculation of the set of C/C++11 pre-executions of a thread to the event structure setting).

Definition Given a current state μ of the register variables (initially 0), the event structure of the memory actions of a list of statements `ss`, $[\![\texttt{ss}]\!]_\mu$, is defined by induction on `ss`:

- case []: the empty event structure;
- case $\texttt{store}_{\texttt{mo}}\texttt{(x,e); ss'}$: an event labelled $\mathsf{W_{mo}\,x\,v}$, where e evaluates to v in μ, before all the events of $[\![\texttt{ss'}]\!]_\mu$:



- case $\texttt{r = load}_{\texttt{mo}}\texttt{(x); ss'}$: for each v in the domain of x, an event labelled $\mathsf{R_{mo}\,x\,v}$, before all the events of $[\![\texttt{ss'}]\!]_{[\mathsf{v}/\mathsf{r}]\mu}$, and all these read events in conflict:



- case `(if (r == v) ss1 else ss2); ss'`: if $\mu(\mathsf{r}) = \mathsf{v}$, then $[\![\texttt{ss1} +\!+ \texttt{ss'}]\!]_\mu$; otherwise, $[\![\texttt{ss2} +\!+ \texttt{ss'}]\!]_\mu$;
- case `r = e; ss'`: $[\![\texttt{ss'}]\!]_{[\mathsf{v}/\mathsf{r}]\mu}$, where e evaluates to v in μ;
- case lock: similar to write;
- case unlock: similar to write;
- case `undef; ss'`: an event labelled $\star$.

5. The Semantics in More Detail

The memory model is formalised in the executable fragment of Lem. It can thus be exported to OCaml to build a tool which allows to explore small examples, though the value speculation rule means that running it exhaustively is often not possible.

In this section we describe the rules in conventional non-mechanised mathematics, manually transcribed from the Lem definition.

5.1 State

Thread state The state of a thread is a confusion-free prime labelled event structure [20], that is

- An underlying set E, the events.
- A labelling function λ from E to an alphabet, here the set of all memory actions.
- A partial order $\leq$ on E that represents "program order", possibly after some transformations. To express that threads have finite histories, $\forall e \in E, \{e' \in E \mid e' \leq e\}$ is finite.
- An irreflexive, symmetric, binary "immediate conflict" relation $\sim$ between events. At most one of a set of events in immediate conflict will be executed.
  - Immediate conflict is "almost transitive": $\forall e, e', e'' \in E, e \sim e' \wedge e' \sim e'' \implies e \sim e'' \vee e = e''$.
  - Immediate conflict is the first point of divergence: $\forall e_1, e_2 \in E, e_1 \sim e_2 \implies \{e \in E \mid e \leq e_1 \wedge e \neq e_1\} = \{e \in E \mid e \leq e_2 \wedge e \neq e_2\}$.
  - Immediate conflict respects program order: $\forall e_1, e_2, e_3 \in E, e_1 \sim e_2 \wedge e_2 \leq e_3 \implies \neg(e_1 \leq e_3)$.

Moreover, all events in immediate conflict are labelled with reads of different values at the same location.

Definition Let $\#$ be the smallest superset of $\sim$ such that $\forall e, e', e'' \in E, e \# e' \wedge e' \leq e'' \implies e \# e''$.

Drawing convention We do not show transitively induced $\leq$ or $\sim$ edges. From now on, we will elide some of the branches of the event structures in the pictures.

Program state The state of a program is a tuple containing

- a finite map from thread ids to thread states;
- a storage subsystem state;
- a trace of the memory actions, and the associated thread ids, for race tracking.

5.2 Outcome

The outcome of the memory model is one of:

- a set of traces
- undefined behaviour:
  - caused by a race
  - caused by thread-local undefined behaviour

If one of the execution paths triggers undefined behaviour, the whole program has undefined behaviour.

5.3 Storage Subsystem

The storage subsystem is a non-multiple-copy-atomic storage subsystem (writes propagate independently to different threads) that preserves "coherence" (C/C++11 "modification order"), based on the Power storage subsystem of Sarkar et al. [23]. While this might seem arbitrary, this storage subsystem model is relatively simple, and is broadly the "minimal" non-multiple-copy-atomic storage subsystem that enforces coherence. Moreover, we want to expose as much of the strength of Power as possible, as C/C++11 did, and Power is the weakest target to which C/C++11 is supposed to be mapped without barriers for relaxed atomics.

Locks Lock and unlock steps are modelled using a read immediately followed by a successful write conditional, and a write immediately followed by a lwsync, respectively, as per the standard Power lock implementation [14] as analysed by Sarkar et al. [24, §5.1]. Because the memory model works directly on the storage subsystem state, there is no need to surround the lock with a loop: the test of whether the right value can be read can be done in the memory model, and the lock action is performed only if the right value is available.

Races The storage subsystem keeps a trace of the executed actions to be able to detect races during execution steps.

5.4 Execution

A thread and the storage subsystem can synchronise to execute a memory action if it is ready to be executed by not being program-order-after anything. The storage subsystem constrains reads by only synchronising on values that can be read at the location of the read, and locks by only synchronising if the location is currently unlocked.

Definition b is a strict descendant of a, $a < b$, when $a \leq b \wedge a \neq b$.

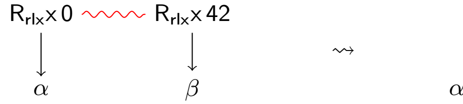
Definition The sub-event structure from e (excluded) of an event structure $(E, \leq, \sim, \lambda)$ is $(E', \leq \cap E'^2, \sim \cap E'^2, \{(x, y) \in \lambda \mid x \in E'\})$, where $E' = \{e' \in E \mid e < e'\}$.

Execution step An event e can be executed if:

- e is minimal with respect to $\leq$;
- the storage subsystem accepts the action labelling e (see Section 5.3).

Action: restrict E to its sub-event structure from e.

For example,



if the storage subsystem accepts the request to read 0 for x in that state (where α and β are just placeholders to make the transformation clearer).

Special events Executing an event labelled $\star$ or †R$_{mo}$ x v has special consequences, see §5.7 and §5.8.

Data races If the last two transitions are execution transitions, where at least one of them is a write, they are both reads or writes, and they are at a non-atomic location, then there is a data race, and the program has undefined behaviour.

5.5 Deordering

We want our semantics to abstract over syntactic ordering and dependencies, and respect only semantic order and dependencies. To do so, we allow threads to remove order when this order does not matter, i.e., where the same action occurs in all the branches. The fact that the same action occurs in all the branches is represented by a set of events B below a set of incompatible events above, A, or a single event above, a.

Definition b is a child of a, $a <_1 b$, when $a < b \wedge (\neg \exists c. a \leq c \wedge c \leq b \wedge c \neq a \wedge c \neq b)$.

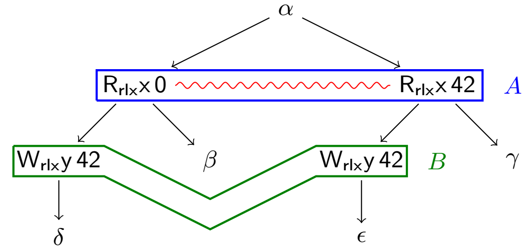
Non-read deordering step A thread can take a non-read deordering step of B with respect to A if:

- each event a_i of a maximal set A of events in immediate conflict has exactly one child b_i in B with the same label (or see §7 and §5.8);
- that label is not a read;
- the memory actions of a_i and b_i are not at the same location;
- the memory action of b_i is not an unlock;
- the memory action of a_i is not a lock;
- all events of B are children of events of A.

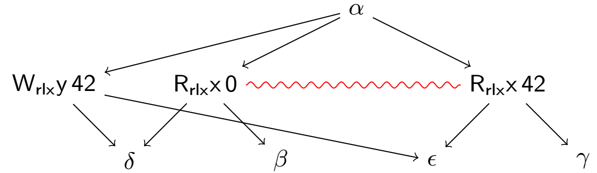
Action:

- remove all but of one the b_i, b;
- remove order from A to b;
- add order from b to the descendants of the b_i;
- restrict $\leq$, $\sim$, and λ.

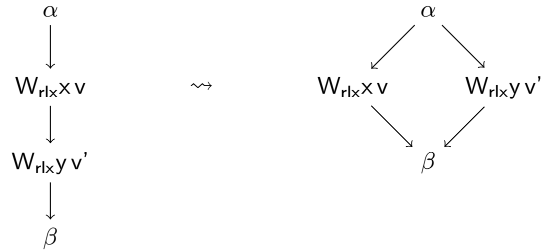
For example,



can transition to



Deordering also applies with respect to non-reads (where A is a singleton), for example

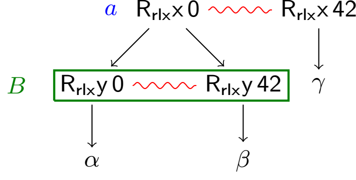


Read deordering step A thread can take a read deordering step of a set of events B with respect to a if:

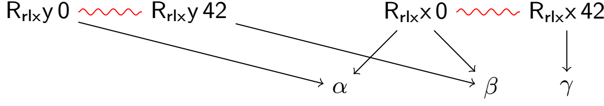
- B is a maximal set of reads in immediate conflict;
- a is a single event;
- a is not a lock;
- a and events of B are not labelled with actions at the same location.

Action: remove order from a to B.

For example,



can transition to



Roach motel deordering The rules for deordering allow "roach motel" deordering, that is, extending critical sections by moving locks up and unlocks down.

5.6 Merging

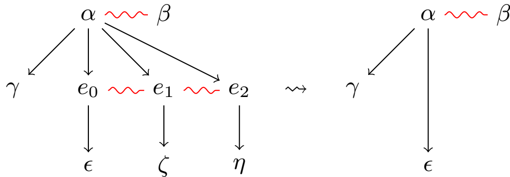
Merging steps allow to merge an event into one or several others, when the effect of the latter can subsume the effect of the former. This is expressed on the event structure by doing a "partial commitment" to the subsumed event, where its alternatives are removed to express that they cannot be executed any more (as in Example 4 in §3.1).

The exclusion of an event e, $\neg e$, is "what e excludes by itself", that is, events in immediate conflict with e, and their descendants:

Definition If $(E, \leq, \sim, \lambda)$ is a confusion-free event structure, and e an event of E, then the *exclusion* of e is $\neg e = \{e' \in E \mid \exists e'' \in E, e \sim e'' \wedge e'' \leq e'\}$

Definition If $(E, \leq, \sim, \lambda)$ is a confusion-free event structure, and e is an event of E, then the relativisation of E with respect to e is $\mathit{relativise}(e, (E, \leq, \#, \lambda)) = (E', \leq \cap E'^2, \sim \cap E'^2, \{(x, y) \in \lambda \mid x \in E'\})$, where $E' = \{e' \in E \mid e' \notin \neg e \wedge e \neq e'\}$.

For example,



An event can be merged forward into a previous event when they are at the same location and have the same value. This merging can take place at a distance, as long as there is not "too much" interposing synchronisation: for an atomic location, an interposing lock restricts the execution of the merged event; for a non-atomic location, an interposing unlock followed by a lock allows for actions from other threads to be executed in between the two events. Moreover, merging a Write into a Read is problematic [28, §7.1], so it is only allowed for non-atomic locations.

Forward merging A thread can take a forward merging step of e_1 into e_0 if:

- e_0 and e_1 are two events of E of value v at location x;
- $e_0 < e_1$;
- on the path between e_0 and e_1
  - there are no actions at location x;
  - if x has an atomic type, there is no lock, and it is not the case that e_0 is a write and e_1 a read;
  - otherwise, there is no unlock followed by a lock.

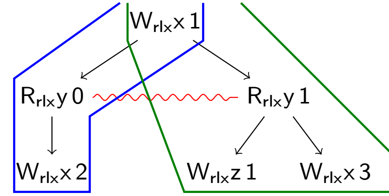
Action: replace E by $\mathit{relativise}(r_1, E)$.
These steps are referred to by the type of action of e_1 and e_0: Read after Read, Read after Write, Write after Read, or Write after Write.

For example, the second read of 42 in the initial event structure of Example 4 (in Section 3.1) can be merged into the first, transforming it into the second thread of Example 3.

Backward merging While the previous step is about an action being subsumed by a previous action, a write can be subsumed by a subsequent write that exists in all executions. This presence of an action in all executions is captured by the notion of a configuration [20].

Definition A configuration C of a confusion-free event structure $(E, \leq, \sim, \lambda)$ is a "downwards-closed" $(\forall e \in C, \forall e' \in E, e' \leq e \implies e' \in C)$ (downwards is upwards in our pictures), conflict-free $(\forall e, e' \in C, e \not\sim e')$ subset of E.

In the example below, the write of 1 to x can be merged into the writes of 2 and 3 to x, because there is an overwriting write in both configurations (in blue and red):



Overwritten write A thread can take an overwritten-write merging step of w into W if:

- $w \notin W$;
- $\{w\} \cup W$ is a set of write events of E at location x;
- each configuration of the sub-event structure rooted at w contains exactly one write in W;
- all writes in W are in a configuration of the sub-event structure rooted at w;
- on the path from w to any write in W;
  - there is no action at location x;
  - if x has an atomic type, there is no unlock;
  - otherwise, there is no unlock followed by a lock.

Action: replace E by $\mathit{relativise}(w, E)$.

5.7 Undefined Behaviour

$\star$ ***as a deordering joker*** If a branch triggers undefined behaviour, then, in a sense, anything could happen in that branch. To reflect this, the condition for the non-read deordering step is relaxed to allow some branches to be labelled with $\star$ instead of the shared label:

> each event a_i of a maximal set A of events in immediate conflict
>
> - has exactly one child b_i in B with the same label,
> - or has a child labelled $\star$ (not in B)

$\star$ ***single child*** Given the above, there is no need to keep the siblings of a $\star$-labelled event, so when deordering a $\star$-labelled event, its (new) siblings and their descendants are removed.

Triggering undefined behaviour If an event labelled $\star$ is executed, it means that the potential for undefined behaviour can be realised, so the program has undefined behaviour.

5.8 Value-Range Speculation

As a dead read is ignored by the steps of the memory model, there is no need to keep the sub-event-structure from the read.

Value-range speculation step A thread can speculate that e is impossible if:

- e is labelled with a non-dead read;
- e is never executed in any execution starting from the resulting state.

Action:

- replace the label of e by the corresponding dead read label;
- remove the sub-event structure from e from the event structure.

Dead reads as deordering jokers Dead reads are not supposed to be executed, so what takes place after them should not matter. To reflect this, the condition for the non-read deordering step with respect to a set of reads in minimal conflict is relaxed so that dead reads are not considered in A by adding the following clause:

- or is a dead read

Merging Similarly, for overwritten write merging, configurations that contain a dead read are ignored.

6. No Classic Out-of-Thin-Air

The point of our semantics is to avoid out-of-thin-air behaviour. However, this cannot be proved, as there is no a priori general definition of what thin-air behaviour is. Instead, we check our semantics gives the desired behaviour on the classic examples.

Example 1 From the initial state, the second thread of Example 1 can deorder the write of 42 to x with respect to the read of y, as it occurs in all branches, and then execute it. The write can then propagate to the first thread, and be read. The write of 42 to x can now be executed, propagated to the second thread, and read.

Example 2 On the other hand, the second thread of Example 2 cannot deorder the write, because it only occurs in one branch of the read. The only possible steps for both reads are to read 0, and to speculate that certain values other than 0 (which can obviously be read) cannot be read. However, as the 0 branch does not contain any write, taking speculation steps will not enable more deordering, so the reads can only read 0.

Java Causality Test Cases We also hand-checked the behaviour of our semantics on the 20 causality test cases collected by Pugh during the JSR-133 Java Memory Model development [21]. Leaving aside the two tests that involve loops and `volatile`, the outcomes are the same as those desired for the JMM, except for causality test 16. This test exposes the (intentional) lack of coherence for non-volatile accesses in the Java memory model, in contrast to C/++11 relaxed atomics, which satisfy coherence. These tests exercise the power of value-range speculation.

No-thin-air for arithmetic-free programs A further sanity check is the "out-of-thin-air guarantee" of Ševčík [29]: in a program without arithmetic, the only values that are read and written are the initial values and values that appear syntactically in the program. Intuitively, this is true in our model because if deordering can deorder a write of a value with the first read that reads it, then that value must syntactically appear.

7. Undefined Behaviour

The undefined-behaviour example of [6, §7] can be expressed in our setting as

```
r1 = load_rlx(x);        ||  r2 = load_rlx(y);
if (r1 == 0)             ||  store_rlx(x,r2)
  r3 = load_rlx(a)       ||
else                     ||
  undef                  ||
store_rlx(y,1)           ||
```

with shared memory variables x and y, initially 0, and a, representing the array in the original example.

In this example, the order of the conditional and the write to y should not matter, because they concern different locations. The undefined behaviour should be triggered, because the write of 1 to y can be executed, making it possible for the second thread to read it, and write 1 to x, which the first thread can read, triggering the undefined behaviour. On the other hand, if the write to y is removed, the presence of a source of undefined behaviour in the program should not be enough for it to trigger itself.

Starting from the initial event structure

R_{rlx}x 0 ~ R_{rlx}x 1
R_{rlx}a 0 ~ R_{rlx}a 1 ⋆
W_{rlx}y 1 W_{rlx}y 1

it can be deordered to

R_{rlx}x 0 ~ R_{rlx}x 1
W_{rlx}y 1 R_{rlx}a 0 ~ R_{rlx}a 1 ⋆

which can itself be deordered to

W_{rlx}y 1 R_{rlx}x 0 ~ R_{rlx}x 1
R_{rlx}a 0 ~ R_{rlx}a 1 ⋆

as the ⋆ acts as a joker for deordering.

The write of 1 to y can then be executed (and propagated) for the second thread to read it, and write 1 to x, which can then be read, which allows executing ⋆.

The order of the conditional and the write in the source does not matter, as desired. If we remove the write to y, the undefined behaviour is not triggered any more, as desired.

8. Implementability on Power/ARM

One of the design goals of the memory model is to be cheaply implementable on Power and ARM, by mapping relaxed and nonatomic reads and writes to plain assembly loads and stores, without memory barriers or any other synchronisation.

We give an informal argument why this holds here, leaving detailed proof for future work. We focus on Power to be concrete, but the reasoning carries over to ARM.

Dependencies Power allows arbitrary thread-local reordering of accesses to different locations (without dependencies or barriers), which is what makes the basic two-thread two-location two-event litmus tests (MP, SB, LB, R, S, 2+2W) of Sarkar et al. [23] all permitted there. Our semantics allows all those reorderings, using the first and second rules of Section 5.5 for R,W→W and R,W→R respectively.

Our semantics has to ensure that semantic dependencies (data, address, and control) from a read to a write are respected, otherwise

the thin-air examples become permitted. The syntactic dependencies that Power respects over-approximate these. For example, take the first thread of the following:

Example 5

```
r1 = load_rlx(x);  ||  r2 = load_rlx(y);
store_rlx(x,r1)    ||  store_rlx(x,r2)
```

This will be mapped, by the compilation scheme for relaxed atomics, to a Power load and store instruction

```
ld   r1,0(rx)  // load doubleword from [rx] to r1
std  r1,0(ry)  // store doubleword in r1 to [ry]
```

where `rx` and `ry` are registers holding the addresses of `x` and `y`. In the Power semantics, that store cannot become visible to other threads until its address and data are known (ultimately, because the architecture does not permit value speculation), validating the fact that our semantics does not allow deordering of the write w.r.t. the read.

In contrast, Power does not respect control dependencies from a read to a read, as the hardware can and does speculate conditional branches. For example,

```
r1 = load_rlx(x);
if (r1 == 1)
  r2 = load_rlx(y)
```

will be mapped to

```
  ld r1,0(rx) // load doubleword from [rx] to r1
  cmpdi r1,1  // compare r1 with 1
  bne .label  // branch if nonequal, to label
  ld r2,0(ry) // load doubleword from [ry] to r2
.label
```

In Power the second load can be satisfied before the first, and this is accommodated in our semantics by permitting the second read (`r2=load_rlx(y)`) to be deordered with respect to the first.

Non-multi-copy-atomicity Power also has non-multi-copy-atomic write propagation, allowing a write to propagating to other threads one-by-one. Our semantics accommodates that with a storage subsystem model that does exactly the same.

Write forwarding Power allows reads to read from program-order-before writes thread-locally, without going through the storage subsystem, as the PPOCA litmus test illustrates [23]. This is just an instance of Read after Write, and is covered by merging in our semantics.

Registers An important source of weakness in Power has to do with the fact that operations on registers do not enforce program order, because of register shadowing, as illustrated by litmus tests MP+dmb/sync+rs and LB+rs [23]. In our semantics, registers are resolved to concrete values when building the event structure, so the program order of register does not impose any order on execution.

Locks The crucial property of locks is that they enforce order between actions before an unlock, and actions after the next lock. In our memory model, this is enforced by preventing reordering with respect to locks, and deordering of unlocks with respect to other actions. In the standard Power lock implementation [14], as analysed by Sarkar et al. [24, §], (1) the `lwsync` preceding the write signalling the unlocking enforces the order between the actions programs-before the unlock and the unlock itself; (2) the branching on the result of the store-conditional, followed by an `isync` in the implementation of `lock`, enforces the order between the lock itself and the actions program-order-after it; (3) the load-reserve/store-conditional pair enforces the order between the unlock and the lock.

9. Relation to C/C++11

The fragment of the C/C++11 memory model restricted to relaxed-atomic and non-atomic reads and writes and locks is widely believed to form an envelope around all reasonable optimisations. Therefore, it would be desirable to have all the behaviours of our memory model, on this fragment, be admitted by C/C++11. The converse is not to be expected, of course, as the point of our model is to exclude the thin-air executions that C/C++11 allows.

While very different in style, our memory model can be related quite closely to C/C++11 candidate executions[2]. A C/C++11 candidate execution is composed of two parts:

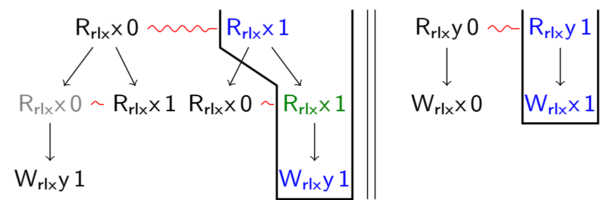
- A pre-execution, which is a set of memory actions of the program annotated with just the syntactic relation between the memory actions: (program order 'sb', and syntactic dependencies 'cd' and 'dd'). When the thread-local semantics builds the set of pre-executions of a program, it considers all the values the reads can read.
- An execution witness, which is an attempt at justifying the pre-execution by saying where each read could have read from (with 'rf' edges), in what order locks are taken and released (with 'lo' edges), and by imposing an order on writes at the same location (with 'mo' edges), etc.

A trace of our memory model induces, for each thread, a unique configuration in its initial event structure. The executed reads, and the reads merged into them, are enough to characterise this configuration. Now, the union of these thread-local configurations is a configuration in the disjoint union of the initial event structures of the threads. This whole-program configuration corresponds to a pre-execution, up to the 'cd' and 'dd' edges, and the initial writes which are implicit in our memory model.

For example, if we consider the program below

```
r1 = load_rlx(x);  ||  r3 = load_rlx(y);
r2 = load_rlx(x);  ||  store_rlx(x,r3)
if (r1 == r2)      ||
  store_rlx(y,1)   ||
```

and its execution where the second read is merged into the first (one step per branch), where the write is deordered, executed, and read, then the corresponding configuration is:



where actions that are executed are in blue, actions that are merged into actions that are executed, and are hence implicitly executed, are in green, actions that are not executed are in black, and actions that are merged into actions that are not executed are in grey.

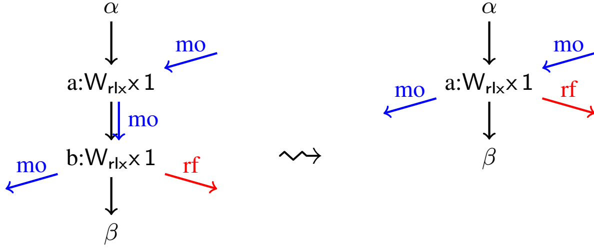
Execution witness A trace also induces an execution witness:

- When receiving a read request, the storage subsystem determines which writes can satisfy it; this induces 'rf' edges.
- When propagating writes, the storage subsystem determines a coherence order on writes at the same location; this induces 'mo' edges.

[2] Because of the limitations w.r.t. undefined behaviour of C/C++11, in this section, we only consider programs without thread-local sources of undefined behaviour.

- When executing the load-linked/store-conditional pairs underpinning locks, the storage subsystem determines an oder between unlock and lock actions; this induces 'lo' edges.

This execution witness is partial, because it only concerns memory actions that are issued to the storage subsystem. However, merged memory actions can be inserted back into the execution witness. For example, if a Write after Write merging turns the event structure fragment (in black) on the left to the one on the right, then the partial execution witness (in colour) on the right can be completed into the one on the left:

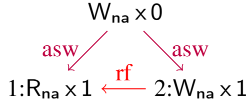


The question now is whether this candidate execution is consistent.

Race reconstruction If the program is racy, the above reconstruction does not always produce a consistent candidate execution. For example, the program below (where x is initially 0) contains a race:

$$\texttt{r1 = load}_{\texttt{na}}\texttt{(x)} \parallel \texttt{store}_{\texttt{na}}\texttt{(x,1)}$$

However, the execution witnessing this race in the C/C++11 memory model is the one where the read reads 0 from the initial state. The execution in which the race actually happens, and for which our model flags the race, is the one where the read reads 1. However, this execution is not consistent according to C/C++11, because the read and the write are not related by the C/C++11 happens-before relation.



This is arguably a design flaw of C/C++11 [4, 6, 27]: the presence of races is determined on consistent candidate executions, but the consistency predicate filters out the actual execution exhibiting the race.

Value-range speculation C/C++11 implicitly assumes any value-range speculation is valid. For example, the configuration that corresponds to the pre-execution of Example 2 that exhibits the thin-air behaviour would correspond to an execution of our memory model where reading values other than 42 has (incorrectly) been speculated to be impossible:

10. Related Work

Currently, no other programming-language memory model addresses the issue of high-performance concurrent accesses satisfactorily.

As we recalled in §2.1, the C/C++11 model [1, 3, 7, 8] does not attempt to exclude thin-air examples, merely noting that they are undesirable. C++14 is similar, with the intentionally vague language "*§29.3p9 Implementations should ensure that no 'out-of-thin-air' values are computed that circularly depend on their own computation*. What "circularly depend on" means is very unclear in a relaxed-memory setting, where some naively impossible cycles are permitted.

The Java memory model (JMM) of Manson et al. [17] excludes the classic thin-air examples by building chains of partial executions, where each execution is justified by the previous one, but can make different control-flow choices. The JMM was later shown not sound w.r.t. some common compiler optimisations, including common subexpression elimination [11, 25]: turning Example 4 into the second thread of Example 3 is not sound in the JMM.

Boehm and Demsky [9] propose a restriction of the C/C++11 memory model that forbids out-of-of-thin-air, but also forbids "load buffering" (including Example 3 reading 42). As that is architecturally allowed on Power and ARM (and actually observed on some ARM hardware), this would require introducing memory barriers when implementing relaxed accesses. It is currently unclear whether the cost of this would be prohibitive.

There is also work suggesting that stronger memory models, like sequential consistency, could be implementable with low overhead [18, 26]. However, it relies on specialised hardware, whereas we target existing hardware.

In Batty et al. [6, §6], we proposed a broadly similar approach to the one we develop here. This previous model is based on steps combining execution and deordering, and accounts for deordering-only examples, like turning Example 1 into Example 3. However, it does not account for more complex optimisations, like turning Example 4 into Example 3. Our approach in this paper features finer-grained steps, separating the different aspects of optimisations, and uses an event-structure rather than a labelled-transition-system representation of the semantics of each thread. This allows us to address the more complex optimisations that had been left for future work, and makes it easier to relate to how compilers operate.

Ševčík [29] describes a memory model aimed at showing the soundness of certain classes of optimisations with respect to DRF and the "out-of-thin-air guarantee". His memory model describes the semantics of each thread as a set of memory-access-event traces, closed under primitive semantic changes: reordering, elimination, and introduction of memory actions. Wildcard traces, containing wildcard read events $\mathsf{R}\,x\,*$, are used to express independence of a trace's validity on the value that the wildcard reads might read; this captures some of the intensional structure of each thread. (Our event structure semantics captures more of that structure, and one can see our previous work [6, §6] as intermediate between the two.)

The memory model framework of Saraswat et al. [22] tackles a language with limited control flow: conditionals can only surround expressions (including reads), not statements (so no writes). The framework is based on merging and splitting state transformers ("steps"). Our treatment of merging is inspired by that work, but tackles unrestricted conditionals by using event structures.

Jeffrey and Riely propose, in work in progress [16], a memory model that also makes use of event structures, but in a more classical way: a single event structure represents the whole program, and an execution (a configuration of the event structure) is justified by a chain of events in a "relaxed" configuration, which can include conflicting events. Their semantics (as it stands) allows some out-of-thin-air executions. We believe their semantics is more liberal than ours.

Jagadeesan et al. [15] define a memory model for Java, replacing the 2005 Java Memory Model with a structural operational semantics for a syntactic calculus that includes explicit value speculation steps, introducing hypothetical writes. To prevent thin-air behaviour, threads cannot see their own speculative writes, and the

justification for speculation needs to be justified by less speculation. The justification for speculating a value can come from a different execution of the program, so it is not clear what behaviour it permits.

Zhang and Feng [30] propose an operational memory model with a "replay" mechanism, where instructions can be re-executed several times. They forbid some, but not all, out-of-thin-air behaviour [13]. Moreover, their replay mechanism appears difficult to relate to compiler and hardware operational intuitions.

Petri and Boudol [10] also develop an operational semantics with explicit speculation steps to show the soundness of some speculation-based optimisations in a DRF memory model.

Demange et al. [12] propose a memory model for Java, but one oriented towards implementation above the relatively simple TSO model of x86 hardware, without the load-store reordering of the Power and ARM architecture that (when combined with compiler optimisations) makes the thin-air problem challenging.

Hardware memory models are typically thin-air-free, as hardware generally does not do value speculation, but they are not sound with respect to common compiler optimisations (collapse of conditionals, CSE, and so on).

11. Conclusion

In this paper, we reaffirmed the need of programming languages for a memory model that forms an envelope around all reasonable optimisations, yet does not exhibit out-of-thin-air behaviour, and outlined a proposed solution for a core calculus. The memory model we describe is an operational semantics where threads are represented by event structures, and transitions mix execution and transformation of the thread state. This memory model behaves as desired on the classical out-of-thin-air test cases. Moreover, unlike C/C++11, it deals with undefined behaviour.

Much work remains to turn it into a full-fledged proposal for a C/C++ or Java semantics. The main avenues are:

1. Feature parity with C/C++11: other memory orders (SC, release, acquire, and consume), other memory actions (read-modify-write, fences), and cross-memory-order optimisations.
2. Integration with a memory layout model, to deal with memory allocation, pointers, mixed-size accesses, etc.
3. Assurance of implementability on more platforms, preferably by mechanised proof w.r.t. well-established hardware models.
4. Assurance of usability, by developing methods of reasoning and discussion with programmers.
5. Assurance of soundness of optimisations, by analysis and testing of current compilers, correctness proofs of particular syntactic optimisations, and discussion with compiler developers to ensure that the model is comprehensible to them.

Although the development so far tackles only the core of the issue, not a full-blown programming language, it offers a potential way forward for concurrent programming language semantics.

Acknowledgments

We thank Mark Batty and Robin Morisset for discussions. This work was partly funded by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems*, EP/K008528/1.

References

[1] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[2] J. F. Bastien. N4455 No sane compiler would optimize atomics, Apr. 2015. available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4455.html`.

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

[4] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[5] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[6] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP*, 2015.

[7] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[8] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008. .

[9] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proc. MSPC*, 2014.

[10] G. Boudol and G. Petri. A theory of speculative computation. In *ESOP*, 2010.

[11] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, 2007.

[12] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *POPL*, 2013.

[13] X. Feng. Presentation at Dagstuhl seminar 15191, May 2015.

[14] IBM. Power ISA version 2.06, 2009.

[15] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. ESOP*, 2010.

[16] A. Jeffrey and J. Riely. Event structures and refinement for relaxed memory. Slides presented at the Memory Model meeting, Cambridge, Sept. 2014.

[17] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *POPL*, 2005.

[18] D. Marino, A. Singh, T. D. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an sc-preserving compiler. In *PLDI*, 2011.

[19] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *Proc. ICFP*, 2014.

[20] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In *Proceedings of Semantics of Concurrent Computation*, 1979. . URL `http://dx.doi.org/10.1007/BFb0022474`.

[21] W. Pugh. Causality test cases. available at `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`.

[22] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPOPP*, 2007.

[23] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186, June 2011.

[24] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *PLDI*, 2012.

[25] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[26] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *Proc. ISCA*, 2012.

[27] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[28] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, 2015.

[29] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[30] Y. Zhang and X. Feng. An operational approach to happens-before memory model. In *TASE*, pages 121–128, 2013.