The Problem of Programming Language Concurrency Semantics

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell

University of Cambridge

Abstract. Despite decades of research, we do not have a satisfactory concurrency semantics for any general-purpose programming language that aims to support concurrent systems code. The Java Memory Model has been shown to be unsound with respect to standard compiler optimisations, while the C/C++11 model is too weak, admitting undesirable *thin-air executions*.

Our goal in this paper is to articulate this major open problem as clearly as is currently possible, showing how it arises from the combination of multiprocessor relaxed-memory behaviour and the desire to accommodate current compiler optimisations. We make several novel contributions that each shed some light on the problem, constraining the possible solutions and identifying new difficulties.

First we give a positive result, proving in HOL4 that the existing axiomatic model for C/C++11 guarantees sequentially consistent semantics for simple race-free programs that do not use low-level atomics (DRF-SC, one of the core design goals). We then describe the thin-air problem and show that it cannot be solved, without restricting current compiler optimisations, using any per-candidate-execution condition in the style of the C/C++11 model. Thin-air executions were thought to be confined to programs using relaxed atomics, but we further show that they recur when one attempts to integrate the concurrency model with more of C, mixing atomic and nonatomic accesses, and that also breaks the DRF-SC result. We then describe a semantics based on an explicit operational construction of out-of-order execution, giving the desired behaviour for thin-air examples but exposing further difficulties with accommodating existing compiler optimisations. Finally, we show that there are major difficulties integrating concurrency semantics with the C/C++ notion of undefined behaviour.

We hope thereby to stimulate and enable research on this key issue.

1 Introduction

Context Shared-memory concurrent machines are now ubiquitous, but, despite decades of research, we still do not have a satisfactory concurrency semantics for any general-purpose programming language that aims to support concurrent systems code. The basic tension is between implementability and usability: to be efficiently implementable, such a semantics must admit the relaxed-memory

behaviours that are permitted by multiprocessor architectures, and those that are introduced by compiler optimisations, but it must also provide sufficiently strong guarantees for concurrent algorithms to work correctly. It is important also for the semantics to be mathematically rigorous, as informal reasoning is particularly error-prone here, it should be as intuitive as possible, it should support testing of implementations and of concurrent algorithms, and it should support compositional reasoning.

There have been two major attempts to develop concurrency semantics for such languages, for Java and C/C++. For Java, the original language specification [20] was shown by Pugh [31] to be flawed in both directions: too strong to be implementable and too weak for some concurrent programming idioms. A new specification [25] was developed in JSR-133, and incorporated into Java 5.0, but that too has been shown to be unsound with respect to standard compiler optimisations, by Cenciarelli et al. [16] and Ševčík and Aspinall [34]. This remains unresolved.

For C and C++, an effort as part of the C++0X standardisation process led to a specification incorporated into the C++11 and C11 standards [9, 2]. The basic design was outlined by Boehm and Adve [13], and Batty et al. [8] developed a formal semantics in the latter stages of the standardisation process, identifying various flaws in the draft standard and feeding back into the ratified standards and later defect reports. C/C++11 concurrency has been supported by GCC and Clang since versions 4.9 and 3.2 respectively, and the model by Batty et al. has been used for many purposes, including correctness proofs for compilation schemes to x86, by Batty et al. [8], and to IBM Power, by Batty et al. [7] and Sarkar et al. [32]; compiler testing via a theory of sound optimisations, by Morisset et al. [29]; model checking, by Norris and Demsky [30]; compositional library abstraction, by Batty et al. [6]; and program logics, by Vafeiadis and Narayan [39] and by Turon et al. [37]. Elements of the model have also been incorporated into OpenCL 2.0. The C/C++11 concurrency model is the best-developed currently in existence, but it also suffers from major problems. The model is known to admit undesirable "thin-air" executions which actual implementations are not thought to exhibit, and it has become clear that these make informal reasoning, formal compositional reasoning, and compiler optimisation very difficult [14, 6, 39, 38]. This too is unresolved.

Without a semantics, programmers currently have to program against their folklore understanding of what the Java and C/C++ implementations provide, and research on verification, compilation, or testing for such languages is on shaky foundations.

Contributions Our goal in this paper is to highlight and articulate this major open problem as clearly as is currently possible, explaining the difficulties with the design of concurrency semantics for shared-memory programming languages in general and for C/C++-like languages (and Java-like, albeit in less depth) languages in particular. We make several novel contributions that each shed some light on the problem, constraining the possible solutions and identifying

new challenges. We begin (§2) by recalling some basic design constraints and choices, to make this paper as self-contained as possible.

Our first new contribution is a positive result: we describe a machine-checked proof, in HOL4 [21], that (for programs without loops or recursion) the model of Batty et al. satisfies one of the core design goals for C/C++11 concurrency: programs that do not use the low-level atomics of the language, and that are race-free in a sequentially consistent (SC) semantics, only exhibit sequentially consistent behaviour (§3). This *DRF-SC* property gives a relatively simple semantics for programmers using that fragment of the language.

We then consider *thin-air* reads (§4). This is a long-standing open problem in the design of the semantics for C/C++11 relaxed atomics: accesses for which races are permitted but where one does not wish to pay the cost of any barriers or other hardware instructions beyond normal reads and writes. The question is how one can define an envelope that permits current compiler optimisations and hardware behaviour, while excluding particular example executions that it is agreed should be forbidden: those with self-satisfying conditional cycles or values appearing out of thin air (this is also closely related to the difficulties with Java). Here we give an instructive negative result: the C/C++11 model is expressed in terms of candidate executions, defining which candidate executions are consistent, but we show that thin-air executions cannot be forbidden in a per-execution style by any adaptation of the C/C++11 consistency predicate that uses the same notion of candidate execution.

In §5 we identify a new problem that arises when one tries to integrate C/C++11 concurrency with semantics for more of the C language. Thin-air executions have previously been thought to be a problem only for programs using the relaxed atomics (intended only for expert use) of C/C++11, but that turns out not to be the case. The model of Batty et al. presupposes an up-front distinction between atomic and non-atomic locations, but that is not present in C, where (for example) one should be able to reuse `malloc`'d regions to store atomics and then nonatomics, or use `char` pointers to read the representation bytes of an atomic. We show that the thin-air problem essentially recurs in this setting, even in the absence of relaxed atomics, and that also breaks the DRF-SC result.

Moving away from per-candidate-execution semantics, we explore an out-of-order operational semantics construction (§6); this gives the desired behaviour for the thin-air examples of §4 but exposes further difficulties with accommodating existing compiler optimisations.

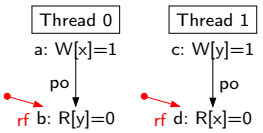
Finally we identify additional new difficulties that arise when integrating concurrency semantics with the C/C++ notion of undefined behaviour (§7). We conclude briefly in §8.

Our HOL4 proof script and the associated Lem definitions are available at `www.cl.cam.ac.uk/~pes20/esop2015-supplementary-material`. We introduce aspects of the C/C++11 model as required, but it is not possible to recap the whole model here; for a full description we refer to [8].

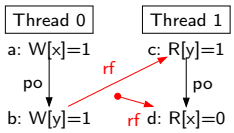
2 Background: an Introduction to the Design Space

Sequential Consistency The most obvious shared-memory concurrency semantics is *sequential consistency (SC)*, in which, as articulated by Lamport [23], any execution has a total order over all memory writes and reads, with each read reading from the most recent write to the same location. This is attractively simple from a theoretical point of view, and it has been the underlying assumption for much research on shared-memory concurrency verification. But it does not capture the concurrency behaviour of typical current systems: multiprocessors exhibit non-SC behaviour, compilers perform optimisations that violate SC, and for C/C++-like languages the language-level memory accesses cannot reasonably be implemented as atomic machine-level accesses. We briefly summarise each of these points in turn.

Non-SC Multiprocessor Behaviour The behaviour of Intel/AMD x86, IBM Power, and ARM multiprocessors has been clarified by a series of recent papers [35, 33, 32, 26, 4]. For x86, normal memory accesses have a Total Store Ordering (TSO) semantics, similar to SPARC TSO [1] — as if there were a FIFO write buffer (with a readback path) for each hardware thread, above a single memory. This allows the SB behaviour on the left below, but little other relaxed behaviour (in these execution diagrams x and y are shared locations, initially 0, po denotes program order, and rf denotes the reads-from relation). Power and ARM are much more relaxed, with programmer-visible out-of-order and speculative execution. For example, the MP behaviour on the right below is allowed, as the writes to different locations might be committed out-of-order, the writes might propagate out-of-order to other threads, and the reads might be satisified out of order.

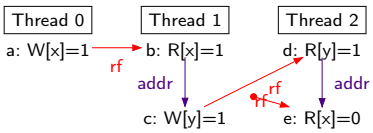


Test SB: Allowed on x86, Power, and ARM



Test MP: Allowed on Power and ARM

Moreover, Power and ARM are not multi-copy atomic: writes to different locations can propagate to multiple other threads in different orders, as in the WRC+addrs example below (pulling the a write of MP to a third thread). The address dependencies prevent local reordering, but the fact that Thread 0's write of x propagates to Thread 1 before its write of y can be committed does not guarantee that the write of x has propagated to Thread 2 before the write of y is propagated to Thread 2.



Test WRC+addrs: Allowed on Power and ARM

One can recover SC in each architecture, but at nontrivial cost: without sophisticated analysis, for x86 one needs an MFENCE barrier between shared stores and loads, while for Power, Sarkar et al. [7] prove that one needs a heavyweight sync barrier between each pair of shared memory accesses.

SC-violating Compiler Optimisations Just as hardware optimisations can result in non-SC behaviour, compiler optimisations can too. The simplest example here is Common Subexpression Elimination (CSE): if two subexpressions are identical, e.g. perhaps just reads of the same location, typical compilers will sometimes retain the value of the first in a register for use instead of the second, effectively hoisting the second read above whatever memory accesses to other locations are in between. This is one of the ways in which the Java Memory Model is unsound with respect to (e.g.) the HotSpot implementation: the implementation does that, but the semantics (unintentionally) disallows it [16, 34]. We return to other compiler optimisations in §4 and §6.

Atomicity Problems Finally, as highlighted by Boehm [10], there is an atomicity mismatch between the language-level memory operations of C/C++-like languages and those that can be implemented reasonably in a concurrent setting. For example, C lets one access a bitfield or a byte within a larger struct, but that might have to be compiled into machine operations that also read or write some of the adjacent memory.

All this means that SC is not viable for current languages, compiler implementations, and hardware (though some authors argue that SC could be achieved at reasonable cost with modified compilers and hardware, e.g. [27, 36]). It is also highly debatable whether SC is desirable: for example, McKenney argues that it does not match the intuitive programming models of those who implement high-performance concurrent algorithms, and notes that the *"Linux kernel makes heavy use of weak ordering"* [28].

TSO as a Language Semantics As the hardware models are now tolerably well-understood, one can imagine lifting them to the programming language level, limiting compiler optimisations to those that are sound w.r.t. the hardware model. The CompCertTSO verified compiler of Ševčík et al. [41] does this for a C-like language (without bitfield accesses), and Demange et al. propose their BMM model for Java-like languages [17]. Both use TSO, which makes for simple implementation on x86 processors but would require expensive fences or sophisticated analysis on Power or ARM machines. This can be reasonable in particular circumstances, especially as x86 is very common, but it is not viable for a general-purpose language intended to support portable high-performance concurrent code.

DRF-SC or Catch Fire The compiler optimisation and atomicity problems with SC described above are only an issue for programs in which multiple threads might be accessing the same location concurrently. Exploiting this fact, Adve and

Hill [3] and Gharachorloo et al. [19] proposed language-level models in which programs that are free of such *data races* (in any SC execution) are guaranteed to have only SC behaviours (DRF-SC), while other programs have completely undefined behaviour. This model is simple to explain and to implement, and it allows a wide range of compiler optimisations (c.f. Ševčík [40] and Morisset et al. [29]). It has two disadvantages: first, giving wholely undefined behaviour to racy programs, while perhaps acceptable for C/C++-like languages (which already have undefined behaviour for other reasons, many of which are not statically decidable), is not acceptable for Java-like languages, which aim to provide memory safety guarantees for arbitrary well-typed code (that led to the complexities of the JSR-133 Java Memory Model [25]). It also begs the question of how one can debug code, and indeed whether there are any large programs that are actually race-free. Second, it requires heavier synchronisation than one wants in some concurrent algorithms.

The C/C++11 Model The C/C++11 model [13, 8, 2, 9] aims to support DRF-SC for simple programs (those using only locks and *SC atomics*), but also provides a range of *low-level atomics* that provide less synchronisation but without the cost of restoring full SC: *release/acquire* write/read pairs for message-passing synchronisation, *relaxed atomics* that should be implementable just with single machine-level loads and stores, and *release/consume* pairs to expose some dependency preservation guarantees of the hardware to make them available in the language. As we shall see, the semantics of all these remains problematic.

3 DRF-SC: sequential consistency for race-free programs

The design of the C/C++11 model could not simply adopt DRF-SC/catch-fire as its definition, due to the need to provide low-level atomics, but it aimed to provide a DRF-SC *property* (for programs that do not use those) as a consequence of its actual definition. We now report on a proof that, for the first time, establishes DRF-SC for the full C/C++11 concurrency model: for programs that do not use low-level atomics and that are race-free in an SC semantics (and subject to conditions detailed below), the full model permits only SC executions. The proof is mechanised in HOL4 and is included in the supplementary material (approx. 23k lines of proof script, including additional model equivalence results). For a more complete account of the proof, see Batty's thesis [5]. Recalling that the prose ISO standards for C++11 and C11 [9, 2] and the mathematical formalisation of the model by Batty et al. [8] correspond closely, this is effectively a mechanised proof of a key metatheoretic property of a mainstream language definition.

There have been two previous results along these lines, but both were preliminary: Boehm and Adve [13] give a hand proof for a preliminary model that omits many features, while Batty et al. [7, Thm. 5] give a hand proof based on an earlier version of their formal model that uses that model's notion of races for the SC semantics. This is a major simplification: the point of a DRF-SC

theorem is to let programmers in the DRF fragment reason solely in terms of an SC semantics, but that result required users to grapple with the full model complexity to understand whether their program contained races. In contrast, the result we present here uses the straightforward SC notion of race based on identifying two conflicting adjacent actions. The mechanisation of the current proof adds assurance, particularly desirable for a fundamental result about an industry-standard model of this intricacy.

To state DRF-SC, we first define a memory model for C/C++ executions, the *total model*, that is manifestly sequentially consistent. We start with a graph over memory accesses, called a *pre-execution* [8], that captures the syntactic structure of the program with a relation for parent-to-child thread ordering and another (*sequenced-before*) for program order. The total model and C/C++11 differ in the relations added to the pre-execution to form their candidate-executions: C/C++11 represents the dynamic behaviour of memory with many partial orders (modification order, lock order and SC order), whereas the total model has only a single total order over all memory accesses in the pre-execution. Reads must read from the immediately preceding write to the same location in the total order, and two accesses race if they access the same location, at least one is a write, they are not both atomic, and they are adjacent in the total order.

The theorem requires that the program ensures that atomic initialisation happens before all atomic accesses for each location. To simplify the proof, we also restrict its statement to programs that satisfy a strong finiteness condition: there must be a finite bound on the size of the pre-executions allowed by the threadwise semantics (this lets us use a simple form of induction). This means it does not apply to programs with recursion or loops. However, intuitively those are orthogonal to the concurrency semantics; we do not know of any reason why including them might affect the truth of the theorem.

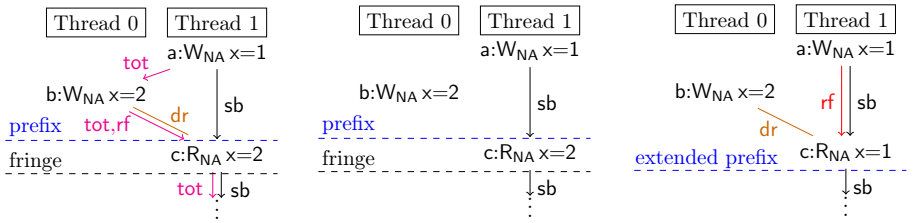
Theorem 1. *For programs whose pre-executions (i) use only mutex, non-atomic and SC-atomic accesses, (ii) have atomic initialisations ordered by sequenced-before and parent-to-child thread synchronisation before all atomic accesses at the same location, and (iii) are bounded in size by some N, either both the C/C++11 model and the total model give undefined behaviour, or the sets of consistent executions in each, projected down to the pre-execution and the reads-from relation, are equal.*

Proof Outline The proof first involves several steps of simplifying the C/C++11 model for programs that do not use low-level atomics. The remaining proof can be split into one part for race-free programs and another for racy ones. For race-free programs there are two cases.

Given a consistent execution in the C/C++11 model, we must construct a consistent execution in the total model with the same pre-execution and reads-from relation. The union of happens-before and SC order is acyclic, so we extend this to a total order and show that that is consistent according to the total model. In the other direction, given a consistent execution in the total model, we project partial relations from the total relation that serve as modification

order, SC order and lock order in a C/C++11 candidate execution, and then show that it is consistent.

Given a racy execution in one model, e.g. the execution in the total model on the left below, we construct a (potentially different) racy execution in the other, e.g. the C/C++11 execution on the right. As one might expect, given a race in the C/C++11 model, constructing a consistent racy execution in the total model is quite involved, and this execution might be very different to its progenitor. Perhaps surprisingly, the other direction is similar: a direct translation, with identical read values, of a consistent execution in the total model is not necessarily consistent in C/C++11. Take the execution on the left below: reads-from would violate the C/C++11 non-atomic reads-from condition that requires the write to happen before the read, so we have to construct a different execution with a race.



To build the execution, we rely on several definitions and an assumption about the *thread-local semantics*: the part of the semantics that enumerates the pre-executions of a particular program. We illustrate these on the example executions above. We define a *prefix* as a part of an execution where every sequenced-before or thread-synchronisation predecessor of any action within the part is also included: e.g. all nodes above the "prefix" lines in the executions above. The *fringe actions* of a prefix are all actions that are not in the prefix, but are immediate sequenced-before or thread-synchronisation successors of an action in the prefix, e.g. precisely c in the left and central executions above. The central diagram above is just the pre-execution of the consistent execution on the left, and hence is allowed by the thread-local semantics. We must assume that the thread-local semantics is *receptive*: for any read or lock in the fringe of a prefix of a pre-execution, allowed by the thread-local semantics, e.g. c in the centre above, and for every other value or lock outcome, there exists a pre-execution with the same prefix, but where the fringe action is changed accordingly, e.g. c in the underlying pre-execution of the right-hand diagram.

Given a racy execution in the total model, we find the first race according to the total relation, e.g. b and c above left, and take the prefix made up of all strict predecessors of the later action (c) with respect to the total order. The prefix is consistent and race free, so we can translate it to a consistent prefix in the C/C++11 memory model with the same set of fringe actions. We extend this to a consistent prefix containing the second racy action, appealing

to receptiveness to change its value if necessary for consistency, producing the execution on the right above, and we show that there is a race in the extended prefix, again between `b` and `c`. This is all inside an induction on the size of the prefix: we show that for each larger finite prefix size, n, either there exists a racy consistent execution, or a racy consistent prefix with at least n actions. Finally, we appeal to the boundedness of executions to establish that there is a racy consistent execution of the program in the C/C++11 memory model.

Given a racy execution in the C/C++11 model, the steps involved in the proof are similar, but finding the first race differs. For each race in the execution, we identify the set containing the racy actions and all of their happens-before predecessors. The execution is finite, so the set of all such sets is finite, and the subset relation is acyclic over them, so we can find a subset-minimal set made up of a pair of racing actions and their happens-before predecessors. We identify one of the racy actions and the happens-before predecessors of both as a race-free prefix. This prefix is consistent, so we can translate it to a consistent prefix in the total model. We then add the previously-racy fringe action to the prefix, and establish that it is consistent and racy, appealing to receptiveness, if necessary for consistency. In a similar fashion to the previous case, we complete the consistent racy prefix to get a consistent racy execution in the total model.

4 The thin-air problem has no per-candidate-execution solution

The question of "thin-air" reads is a longstanding issue in the design of memory models for C and C++, specifically for C/C++11 relaxed atomics: accesses for which races are permitted but which should be implemented with normal load and store instructions, without the cost of additional barriers or synchronisation instructions. Related questions arise in the semantics of C as used in the Linux Kernel (for `ACCESS_ONCE` accesses), and for normal accesses in Java [25].

The C++11 standard [9] included text intended to forbid thin-air executions (29.3p9), and it says explicitly (29.3p10) that that text forbids the LB+data example below, but the text was already recognised as flawed: a non-normative note in the standard (29.3p11) observed that *"The requirements do allow [the LB+ctrldata+ctrl-single example below]. However, implementations should not allow such behavior."*. Batty et al. identified further problems [8, §4], and their formal model does not attempt to capture that text or to exclude thin-air executions in any other way. The current proposal [12] for C++14 acknowledges difficulties with the C++11 version and proposes a deliberately vague placeholder as an interim replacement: *"Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation."*.

There is not a precise definition of what it means for a read to be "out of thin air" (if there were, the problem would be solved, as the semantics could simply exclude those). Rather, there are some example executions for which there is a consensus that the language should forbid them, and that current hardware and compiler optimisations do not exhibit. This is a high-level-language specification

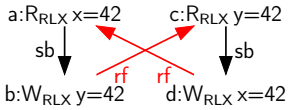
problem: there is no suggestion that thin-air executions occur in practice with current compilers and hardware; the problem is rather how to exclude them without preventing desired compiler optimisations.

In this section, we describe the thin-air problem via a series of examples, and we show that thin-air executions cannot be forbidden without restricting current compiler optimisations by any per-candidate-execution condition using the C/C++11 notion of candidate executions.

For each example we identify a particular execution by specifying the values read, and discuss whether it should be allowed by the semantics or not.

Example LB (language must allow)

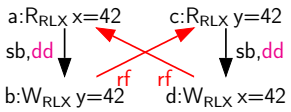
```
r1=load_rlx(x) //reads 42
store_rlx(y,42)
-----------------------
r2=load_rlx(y) //reads 42
store_rlx(x,42)
```



Here `r1` and `r2` are thread-local variables (which do not have memory actions in the model), while `x` and `y` are shared variables; initially all are `0`. This execution (the dual of the first example of §2) is permitted by the ARM and IBM POWER architectures (presuming the code is compiled in the obvious way into machine load and store instructions): the actions of the each thread are to manifestly different addresses and so can be done out of order; it is moreover experimentally observable on current ARM multiprocessors [33]. Hence, the language semantics must allow it for relaxed atomics.

Example LB+datas (language can and should forbid)

```
r1=load_rlx(x) //reads 42
store_rlx(y,r1)
-----------------------
r2=load_rlx(y) //reads 42
store_rlx(x,r2)
```



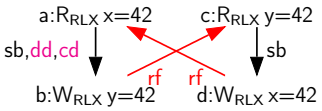
There are two paradigmatic kinds of thin-air execution, the *thin-air read value* executions like this one, in which a value (here `42`) "appears out of thin air", and the *self-satisfying conditional* example we discuss below. This example is architecturally forbidden on current hardware (x86, ARM, and IBM POWER), we do not expect future hardware to adopt the load-value prediction that would be required to make it observable, and to the best of our knowledge it cannot be exhibited by any reasonable current compiler optimisation combined with current hardware. Hence, the language semantics could forbid it.

Moreover, it is clearly desirable to forbid it, to make the language semantics as intuitive as possible. Boehm and Demsky [14] give examples where programming with relaxed atomics that permit thin-air values would be problematic, and in languages that aim to preserve implementation invariants at some types (such as that all pointer values point to allocated memory) it would be essential.

As for *how* it might be forbidden, the example suggests that one might simply forbid candidate executions with cycles in the union of the reads-from and dependency relations (the model has a data dependency relation shown as *dd* above). But the next two examples show that a combination of hardware behaviour and compiler optimisations make that infeasible.

Example LB+ctrldata+po (language must allow)

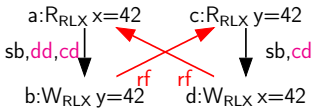
```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
-------------------------
r2=load_rlx(y) //reads 42
store_rlx(x,42)
```



This is architecturally allowed on ARM and Power (for the same reason as LB), and likewise observable on ARM, hence the language must allow it.

Example LB+ctrldata+ctrl-double (language must allow)

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
-------------------------
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
else
  store_rlx(x,42)
```

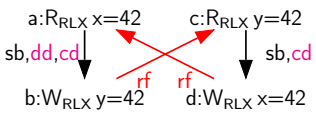


This is forbidden on hardware if compiled naively, as the architectures respect read-to-write control dependencies, but in practice compilers will collapse conditionals like that of the second thread, removing the control dependencies from the read of `y` to the writes of `x` and making the code identical to the previous example. As that example is allowed and observable on hardware (and we presume that it would be impractical to outlaw such optimisation for C or C++), the language must also allow this execution. But this execution has a cycle in the union of reads-from and dependency, so we cannot simply exclude all those.

Then one might hope for some other adaptation of the C/C++11 model, but the following example shows at least that there is no per-candidate-execution solution.

Example LB+ctrldata+ctrl-single (language can and should forbid)

```
r1=load_rlx(x) //reads 42
if (r1 == 42)
  store_rlx(y,r1)
-------------------------
r2=load_rlx(y) //reads 42
if (r2 == 42)
  store_rlx(x,42)
```



This is the paradigmatic "self-satisfying conditional" example. It is forbidden on hardware if compiled naively (both ARM and POWER architectures prevent speculative writes becoming visible to other threads), and applying reasonable thread-local compiler optimisation does not change that. Hence, the language could forbid it. Moreover, it is problematic for informal and formal compositional reasoning [14, 6, 39], so the language should forbid it.

But the candidate execution that we want to forbid here is identical to the execution of the previous example that we have to allow. This immediately gives:

Theorem 2. *No adaptation of the C/C++11 per-candidate-execution definition that uses the same notion of candidate execution can give the desired behaviour for both of these examples.*

The basic point here is that compiler optimisations (such as the collapse of the LB+ctrldata+ctrl-double conditional) are operating over a representation of the *program*, covering all its executions, while the C/C++11 definition of candidate execution and consistency for those considers each candidate *execution* independently (it ignores the set of all executions); it is not able to capture the fact that the conditional is unnecessary because the two candidate executions corresponding to taking the two branches are equivalent. We develop this observation in §6.

Restricting optimisation involving relaxed atomics? One might think that it would be feasible to restrict just compiler optimisations involving relaxed atomics, e.g. requiring that the compiler should respect all dependencies between relaxed atomic operations, while permitting more optimisation elsewhere. But (as observed by Boehm [11]) dependencies can be via functions in other compilation units that only involve non-atomic accesses, e.g. as in the version of LB+ctrldata+ctrl-double below, where the second thread's conditional is factored out into a function `f()` that does not involve atomics and that is in a different compilation unit. When compiling `f()` the compiler cannot tell whether it might be used in a dependency chain between atomic accesses, and so it would have to preserve all such dependencies. The cost of that is unknown, and worth investigating experimentally, but we suspect it to be unacceptable.

```
// in one compilation unit
void f(int ra, int*rb) {
  if (ra==42)
    *rb = 42;
  else
    *rb = 42; }

// in another compilation unit
r1=load_rlx(x) //reads 42  | r2=load_rlx(y) //reads 42
if (r1 == 42)              | f(r2,&r3)
  store_rlx(y,r1)          | store_rlx(x,r3)
```

In practice, GCC (checked with 4.6.3 on x86) does optimise away the control dependency in `f()`, at `O1`, `O2`, or `O3`.

Preventing load-store reordering If one relaxes the requirement that relaxed atomics must be implementable with simple machine accesses, one might restrict all shared-variable load-to-store reordering, as proposed by Boehm and Demsky [12, 14], adding barriers and somewhat restricting compiler optimisation. The cost has not yet been quantitatively assessed. For C/C++ it might be viable due to the small number of relaxed atomics (though if practitioners resorted to in-line assembly instead, that would defeat the purpose). But for normal Java accesses on ARM or Power, the cost seems likely to be prohibitive.

5 Integrating non-atomics and atomics leads back to thin air

We now show that the thin-air problem is not confined to relaxed atomics. The C++11 standard prose refers to *"atomic objects"* as if they are quite different from non-atomic objects, and the mathematical model of Batty et al. [8] for the C++11 and C11 concurrency primitives followed suit by imposing a simple type discipline: a *location kind* map in each candidate execution partitioned locations into atomic, nonatomic, and mutex locations. The definition of consistent execution permitted atomic accesses only at atomic locations, and the only nonatomic accesses allowed at atomic locations were atomic initialisations[1].

However, when one considers generalising that semantics for the concurrency primitives to cover more of C, it becomes clear that an up-front location-kind distinction is unrealistic, for several reasons:

1. In C it is permitted to reuse a region of allocated storage (e.g. from `malloc`) at a new type, simply by overwriting the bytes of memory with a new value. Restricting that to prevent strong updates from atomic to nonatomic (or v.v.) would not give a usable language.
2. In C one can inspect the representation bytes of a value by casting a pointer to `(char *)`, or by type-punning via a union.
3. In C one can copy a value by copying its representation bytes, e.g. using `memcpy`. This could perhaps be deemed illegal for structures containing atomic values (indeed, it would have to be if atomic values had to be registered somewhere in the implementation), but it would be preferable, and in keeping with the rest of the language, to permit it.
4. In C11 one can construct atomic versions of structure and union types (with `_Atomic(`*`type-name`*`)` or the `_Atomic` qualifier), but their members can be accessed only via a non-atomic object which is assigned to or from the atomic object, not directly [2, 6.5.2.3p5].

Hence, contrary to [8], we have to allow mixtures of atomic and nonatomic accesses at the same location, at least where the nonatomic accesses do not race with each other or with any atomic accesses.

[1] It is desirable to have nonatomic initialisations so that they do not require fences, but then to obtain a DRF-SC result initialisation had to be limited to be happens-before all other accesses, and without reinitialisation.

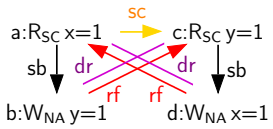
But what should the semantics be for these? The standard text does not directly address these mixtures, but for the entirely nonatomic and entirely atomic cases it and the formal model [8] are clear:

- for the non-atomic case, the definition of consistent execution requires, in *consistent_non_atomic_rf*, the read to read from the most recent happens-before-visible write to the same location; while
- for the atomic case, the analogous *consistent_atomic_rf* lets the read read from any write that is not after it in happens-before (subject to the other predicates of the model).

Neither of these predicates are suitable to govern mixtures of atomic and non-atomic accesses, as the following two examples show. Our first example program uses `memcpy` to mix atomic and non-atomic accesses at the same location. The C/C++11 memory model as it stands suggests that the mixed accesses would be governed by *consistent_atomic_rf*, because the location has an atomic type. However, this breaks DRF-SC: the example program is race-free in every SC execution, but it has racy executions in the C/C++11 memory model:

```
// parent thread
size_t s = sizeof(atomic_int)
atomic_int x = 0
atomic_int y = 0
atomic_int a = 1
```

`int r1 = load_sc(x)`	`int r2 = load_sc(y)`
`if(r1 != 0)`	`if (r2 != 0)`
`  memcpy(&y,&a,s)`	`  memcpy(&x,&a,s)`

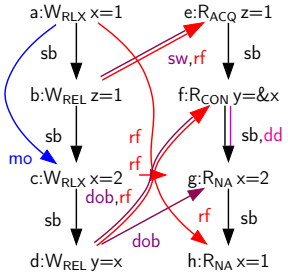


In the execution above, each atomic load reads from the non-atomic write implicit in the `memcpy` of the other thread. The execution is consistent and has data races. Breaking DRF-SC makes *consistent_atomic_rf* unsuitable to govern non-atomic reads from atomic writes. By swapping the atomics and non-atomics in the example, we see that it is also not suitable to govern atomic reads from non-atomic writes.

Our second example establishes that we also cannot use the *consistent_non_atomic_rf* predicate for mixtures. In the program below, there is a reading thread that spins until it sees the other thread's writes of `z` and `y`, and then reads from `x` twice: once with acquire memory order and

once with consume. After the loop, there are two `memcpy`'s of location `x`:

```
// parent thread
size_t s = sizeof(atomic_int)
atomic_int n=0, x=0, y=0, z=0
```

$\texttt{store}_{\texttt{rlx}}\texttt{(x,1)}$ $\texttt{store}_{\texttt{rel}}\texttt{(z,1)}$ $\texttt{store}_{\texttt{rlx}}\texttt{(x,2)}$ $\texttt{store}_{\texttt{rel}}\texttt{(y,\&x)}$	

```
do { r1 = load_acq(z)
     r2 = load_con(y)}
while (r1==0 || r2==0)
memcpy(&n,r2,s)
memcpy(&n,&x,s)
```

In the candidate execution on the right above, the loop exits (we elide the implicit write of the `memcpy`'s, and the initialisation writes). The first `memcpy` happens after all atomic writes of `x`, but before the write implicit in the second `memcpy`, so according to *consistent_non_atomic_rf*, it must read write `c`. The second `memcpy` reads a pointer provided by the consume read, creating a dependency and forcing it to read `a`, but this execution, shown above, contains a CoRR coherence violation between accesses `a`, `c`, `g` and `h`, making the execution inconsistent, so the only behaviour that the model allows of this program is spinning on the conditional of the loop (similar executions arise if we swap atomics with non-atomics and vice versa), when in fact the program contains a race. Using *consistent_non_atomic_rf* for the mixtures cuts out executions we need to allow: it can make reasonable executions of race-free programs inconsistent and remove racy executions from racy programs, making them race-free and well-defined.

Vafeiadis et al. provide another alternative semantics for non-atomic reads [38]: modification order and coherence are extended to cover all locations (including non-atomics), atomic reads use the existing condition for reads at atomic locations, and the condition on non-atomic reads is replaced with a requirement that a new relation, the union of happens-before and `rf` edges to or from non-atomic accesses, is acyclic. This semantics provides the desired behaviour in the examples above, but, as noted by Vafeiadis et al., it forbids compiler optimisations from reordering loads followed by stores. Morisset et al. observe that this sort of reordering results from loop invariant code motion [29], an optimisation performed by both GCC and LLVM [18, 24], so this attractive semantics comes with the unacceptable cost of forbidding routine compiler optimisations over blocks of non-atomic code.

We have seen that using *consistent_nonatomic_rf* to govern the behaviour of non-atomic reads at locations accessed atomically removes too many behaviours; we cannot use *consistent_atomic_rf* to govern such reads either (that would break DRF-SC); and the suggestion of Vafeiadis et al. comes at too high a cost. It is not clear what the semantics of non-atomic reads should be in C11.

6 An out-of-order operational construction

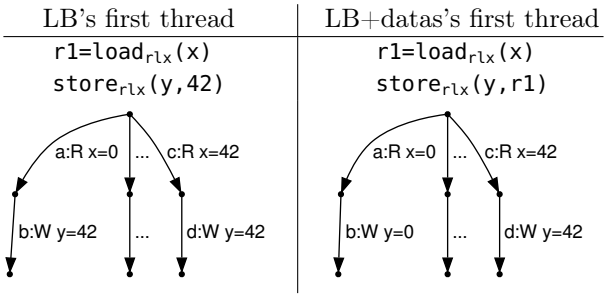
The examples of §4 showed that, for relaxed atomics, the language semantics has to admit reorderings that are enabled by removals of syntactic control dependencies, where those removals can be justified only by examination of multiple control-flow paths (not just inspection of a single candidate execution). For example, consider again the second thread of LB+ctrldata+ctrl-double:

```
r2=load_rlx(x)   --compiler-->  r2=load_rlx(y)   --h/w-->  store_rlx(x,42)
if (r2 == 42)                   store_rlx(x,42)            r2=load_rlx(y)
  store_rlx(x,42)
else
  store_rlx(x,42)
```

The key fact here is that the `store_rlx(x,42)` is possible on all control-flow paths of this thread, and a sufficiently "smart" compiler can detect that and then remove the control dependency from the read of `y`. In this section we generalise this observation: we give a semantics for relaxed and nonatomic accesses (and locks and fences) that correctly accounts for all the thin-air examples of §4 in an interesting and reasonably clean way. But those examples only involve reorderings; in §6.2 we use this semantics to highlight difficulties with other common optimisations.

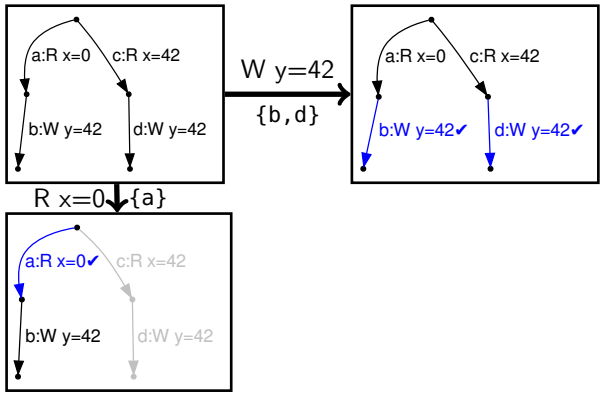
6.1 The semantics for reorderings

We start from a standard labelled transition system (LTS) semantics for each thread in isolation, describing its interactions with memory by transitions labelled `a:R x=v` and `b:W x=v` for a read or write of value `v` at location `x`. This thread-local base semantics does not constrain the values read from memory in any way; it simply has a transition for each possible read value. For example, looking at some of the threads from the §4 tests, we have:



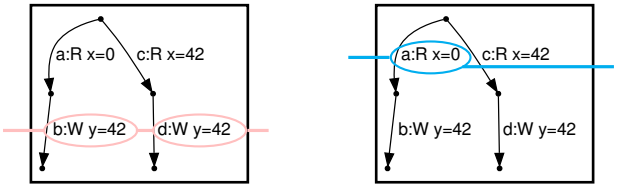
In LB's first thread, there is a write of 42 to `y` in all branches of the LTS, and we will allow the thread to write 42 before reading, letting both threads read 42. On the other hand, in LB+datas's first thread, it is not the case that a write of 42 is available in all branches, so it will have to do the read first, preventing LB+datas from exhibiting out-of-thin-air behaviour.

We capture this by constructing a derived *out-of-order* labelled transition system for each thread. Its states are copies of the entire base in-order LTS with some edges *ticked*. The initial state is the base LTS with no edge ticked. For example, part of the out-of-order LTS for LB's first thread is shown below. From now on, we only show the branches for some interesting values; in reality there is one branch per possible value, as we assume the base LTS is receptive.



The transitions are labelled with the same memory actions as the base semantics; each transition of the derived LTS corresponds to ticking a set of base transitions. But the base transitions can be performed out-of-order, when they are not blocked (as defined below) in any branch by coherence or fences. Specifically: a set of edges can be ticked iff it forms a *frontier*, that is, (1) it is non-empty, (2) the edges are not ticked, (3) the edges have the same memory action label, (4) each non-discarded path either has a single edge in the frontier, or becomes discarded by this ticking, and (5) no edge is blocked (see below). Here an edge is *discarded* if it has a ticked sibling, and a path is discarded if it contains a discarded edge.

For example, the horizontal transition above is justified by the frontier on the left below consisting of all the W y=42 edges (b, d, and all the similar edges in elided paths), while the vertical transition is justified by the frontier on the right below consisting just of a (and there is a similar transition, not shown, for each base transition with a different read value).



An edge is blocked by another if its action cannot be reordered before the other's. To maintain coherence (the fact that execution respects a per-location total order over writes to each location, consistent with program order, as guaranteed by standard hardware and by C11 relaxed atomics), actions to the same

location cannot be reordered. Fences cannot be reordered before or after actions, so that all the actions before the fence have to be ticked before the fence can be ticked, and all the actions before the fence and the fence itself have to be ticked before actions after the fence can be ticked. Unlock and lock actions cannot be reordered before and after actions, respectively, but can in some cases be reordered the other way around, to allow for roach motel reordering.

Handling nonatomics Non-atomic accesses can be executed out-of-order, like relaxed accesses, but in addition, they can also cause races, which the semantics has to be able to detect.

Non-multi-copy-atomic memory For two-thread examples, one can combine the derived LTS of each thread with an underlying sequentially consistent shared memory (and that is what we have done for the testing described below). But in general the language semantics must also admit the lack of multi-copy atomicity permitted by the Power and ARM architectures, as described in §2. This can be handled by taking the parallel composition of the thread subsystems given by the derived LTSs with a storage subsystem following that of Sarkar et al. [33], which provides a generic non-multi-copy-atomic memory by keeping track of (a) the coherence commitments made among write events, and (b) the lists of writes and barriers propagated to each thread. The storage and thread subsystems are then synchronised on write requests, read requests and responses, etc.

This semantics gives the desired behaviour for each of the thin-air examples of §4: it is liberal enough to allow the reordering (introduced by compiler or hardware) that gives rise to the "must be allowed" examples, and restrictive enough to prevent the "should be forbidden" examples, ruling out thin-air executions basically by executing along a totally ordered trace of the derived LTS, with reads reading from previous writes in that trace. We have a precise Lem definition of the out-of-order semantics, and have built a tool that lets one explore the semantics of small examples, based on OCaml code generated from the Lem and integrated with an underlying SC memory. It has several good features:

- It is operational and relatively concrete, which makes it easier to understand than (say) the C11 axiomatic memory model.
- The construction is independent from the language syntax and thread-local operational semantics, which is highly desirable for tackling a complex language like C. This contrasts with explicit-speculation calculi, e.g. [15, 22].
- For entirely thread-local computation, as thread-local variables do not create memory events, optimisations are already factored into the computation of the thread-local LTS.
- It does not involve syntactic notions of dependency, which are difficult for optimising compilers to preserve.

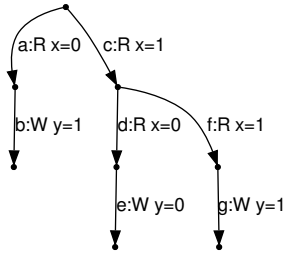
However, this semantics does not allow behaviour that is introduced by many other common compiler optimisations. Looking at these other optimisations highlights some subtle issues that any semantics for a C-like language will have to tackle.

6.2 Optimisations beyond reordering

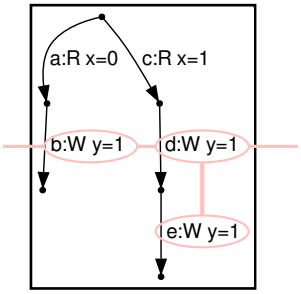
In contrast to hardware semantics, there is (to date) no good characterisation of the envelope of all compiler optimisations normally performed in practice. The syntactic optimisations that are performed by compilers are numerous (GCC and Clang each have of the order of 100 passes) and they have unclear effects and interactions. Ševčík [40] and Morisset et al. [29] consider some abstract classes of optimisations, but these are only thread-local. In this section we give a preliminary discussion of some optimisations that go beyond reordering, in the context of the out-of-order semantics.

Elimination of subsumed memory actions Many common compiler optimisations, like constant propagation and common subexpression elimination (CSE), can be explained in terms of eliminations of individual memory accesses [40]: read after read, read after write, write after read, and overwritten write elimination, which consist in conflating actions when the effect of one subsumes that of the others. For example, in the following program, the second read of `x` can be merged into the first as a very simple instance of CSE (by a read after read elimination); then, both branches of the conditional write 1 to `x`, so this write can be executed out-of-order, so there is an execution where both `r1` and `r3` are 1.

`r1=load`$_{\texttt{rlx}}$`(x)`	`r3=load`$_{\texttt{rlx}}$`(y)`
`if (r1 == 1)`	`store`$_{\texttt{rlx}}$`(x,r3)`
`  r2=load`$_{\texttt{rlx}}$`(x)`	
`  store`$_{\texttt{rlx}}$`(y,r2)`	
`else`	
`  store`$_{\texttt{rlx}}$`(y,1)`	



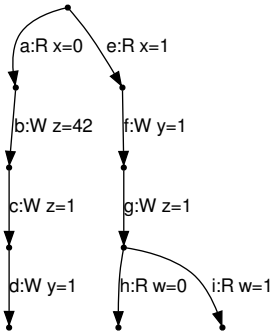
We conjecture that the notion of frontier can be relaxed to deal with these, e.g. with extended frontiers as below. We interleave optimisations (extended frontiers) with execution (ticking) on purpose to account for adaptive optimisations. When compilers perform this kind of optimisation, they effectively identify extended frontiers, and collapse them into elementary frontiers, but work on finite foldings of the LTSs, like SSA.



These optimisations need information about multiple paths, but only in a limited way: they only need the existence of particular actions (in a non-blocked path context) in each path. However, this is not the case for all optimisations, as we show next.

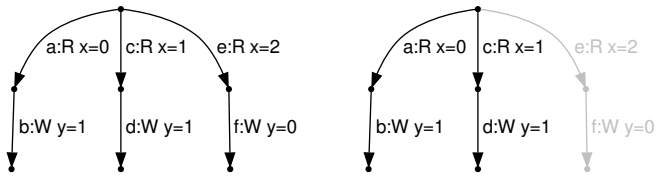
Irrelevant read elimination Intuitively, irrelevant read elimination consists in removing a read action when its result does not affect the thread's behaviour: for example, if the branches of a read have identical subtrees, it is certainly irrelevant. But in general a read is irrelevant if its subtrees are in some sense semantically equivalent, where equivalence is up to optimisations, including reordering, eliminations, and irrelevant read elimination. For example, in the following program, the read of `x` is irrelevant only up to reordering of the writes to `y` and `z`, overwritten write elimination of the first write to `z` in the `else` branch, and irrelevant read elimination of the read of `w`. This suggests a recursive construction of the memory model, but it is not clear at what level: thread-local read-irrelevance, whole-program read irrelevance, etc.

```
r1=load_rlx(x)
if (r1 == 1) {
  store_rlx(y,1)
  store_rlx(z,1)
  r2=load_rlx(w)
} else {
  store_rlx(z,42)
  store_rlx(z,1)
  store_rlx(y,1)
}
```



Inter-thread optimisations The previous optimisations were all thread-local. Inter-thread optimisations (alias analysis, pointer analysis, ...) turn out to be even more challenging. The out-of-order construction makes no assumption about what values can be read, and thread-local LTSs thus have a branch for every value of each read. Identifying a value restriction amounts to discarding some "impossible" branches of the LTS. This can create more valid frontiers, and hence permit more out-of-order behaviour. For example, in the LTS below, if, by looking at all the writes to `x` by all the threads, the compiler determines that `x` can only contain values 0 and 1, then it can discard the branch where the value 2 is read, which makes $\{\mathtt{b}, \mathtt{d}\}$ into a frontier, which allows the write to `y` to be executed before the read from `x`:

```
r1=load_rlx(x)
if (r1 == 2)
  store_rlx(y,0)
else
  store_rlx(y,1)
```



Moreover, some optimisations restrict behaviour, which creates more opportunities for inter-thread analyses, so inter-thread optimisations cannot be separated to an initial phase, but have to be intertwined with the other optimisations. This again suggests a recursive construction of the memory model. For example, in the following program, the second read of `x` can be merged into the first (by read after read elimination); value-range analysis can then remove the conditional, which allows additional behaviour: `r1` and `r3` can be 42.

$\texttt{r1=load}_{\texttt{rlx}}\texttt{(x)}$	$\texttt{r3=load}_{\texttt{rlx}}\texttt{(y)}$
$\texttt{r2=load}_{\texttt{rlx}}\texttt{(x)}$	$\texttt{store}_{\texttt{rlx}}\texttt{(x,r3)}$
`if (r1 == r2)`	
$\texttt{store}_{\texttt{rlx}}\texttt{(y,42)}$	
`else`	
$\texttt{store}_{\texttt{rlx}}\texttt{(y,43)}$	

The additional behaviour introduced by the analysis can invalidate it, or enable more optimisations that can invalidate it, so the semantics cannot be defined by a naive fixpoint.

Thread-local and shared variables Finally, the out-of-order semantics is defined over a calculus that has a syntactic distinction between thread-local variables and potentially-shared variables. This distinction is important, as the semantics does not need to consider interference on thread-local variables, and thread-local optimisations on them are built into the base LTS construction, and can be much more aggressive. For example, in the following program, if `x` is determined to be thread-local, then constant propagation (in our framework, read after write elimination) can be done across the synchronisation.

```
x = 7
unlock(l)
...
lock(l)
r1 = x
```

However, C does not have such a distinction, and whether a variable behaves thread-locally depends on the dynamic behaviour of the program, which in turn depends on which variables behave thread-locally.

7 Concurrency and undefined behaviour

For our final contribution, we observe that there is a fundamental mismatch between the concurrency models of C/C++11 and the treatment of undefined behaviour in their preexisting specifications.

The C and C++ standards impose many constraints on programs by attributing *undefined behaviour* to programs that exhibit them (for C these are collected in [2, J.2]). Some of these are static properties (e.g. programs should define a `main` function) but many are dynamic, e.g. there should be no division

by zero or out-of-bounds array access (OOBAA). For programs with undefined behaviour, the standard does not say that execution fails or behaves arbitrarily at that point. Instead, the compiler is completely unconstrained in the code it produces for the whole program [2, §3.4.3#1]:

> NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

This is important because optimisations can involve significant code motion. For example, in an execution in which `x=0`, the following reaches a division-by-zero after the `puts`, both in the sequential execution model of the standard and in a non-optimising implementation. But an optimising compiler that does loop-invariant code motion might well hoist the `1/x` before the loop, reaching the division-by-zero error before the `puts`. That code motion is made legal in general by giving this program entirely undefined behaviour.

```
for(int i=0; i<5; i++) {
  puts("foo\n");
  ret += i + 1/x;
}
```

Integrating the concurrency model into the language changes things. There are new sources of undefined behaviour: any program with a data race has undefined behaviour, which (for example) licenses the conventional implementation of bitfield operations mentioned in §2. But the overall form of the semantics also changes: instead of that simple sequential execution model (used to discover the division-by-zero on a reachable path) the definition calculates the set of candidate complete executions (essentially graphs like the examples shown in §4 and §5) that satisfy the *consistency predicate* of the concurrency model; if none of those contains a data race, then they are the allowable behaviour of the program (otherwise the program is undefined). There is a tension between this global completed-execution structure and the implicit use of the sequential execution model to discover the earlier forms of undefined behaviour.

For example, the C standard says that out-of-bounds array access is undefined behaviour [2, §6.5.6#8 (for an access from one-past an array)]. In the sequential setting (or indeed in an SC concurrent setting) there is a clear notion of execution prefix, and to identify such an undefined behaviour one only has to consider such a prefix leading up to it. But in the concurrency model, LB-like tests show that parts of a candidate complete execution that follow (in program order) the offending access might influence whether it is performed; we cannot restrict attention to simple prefixes. Consider the following example, where `x` and `y` are

atomic integers initialised to 0, and a is an integer array with two elements:

r1 = load$_{rlx}$(x)	r2 = load$_{rlx}$(y)
r3 = a[r1]	store$_{rlx}$(x,r2)
store$_{rlx}$(y,2)	

In any sequentially consistent execution of the program, the first thread loads 0 from x, and there is no OOBAA. But with the intended implementation of relaxed atomics above the ARM or Power architectures, there can be an execution where the second thread loads the store of 2 to y then writes to x, and the first thread loads 2 from x and then performs an OOBAA[2]. As a consequence, the language must provide this program with undefined behaviour.

But to identify this undefined behaviour, we need to consider executions that go past it in program order, and that means we need to choose some semantics for the out-of-bounds array access, and the other sources of undefined behaviour, to provide a context for the subsequent execution. This leads to a great many questions about the semantics of constructs that might introduce undefined behaviour. Taking out-of-bounds array access as an example, what should the semantics of an out-of-bounds load be, what if control flow is decided by the result of the load, what if the access is a store, or if the access loads or stores a function pointer? In each of these cases, it is unclear what the semantics should be. The point of undefined behaviour in the C and C++ semantics is to cover cases where the language semantics cannot easily reflect what an implementation might do, so one would prefer not to have to answer such questions.

8 Conclusion

The C/C++11 concurrency model remains the state of the art for the semantics of a general-purpose shared-memory concurrent programming languages; it is, to the best of our knowledge, sound with respect to the compiler optimisation behaviour of implementations [29] (in contrast to the JMM [16, 34]), it is provably compilable to relaxed hardware models [8, 7, 32], and our work here establishes a machine-checked DRF-SC theorem. But the thin-air problem shows that it allows too many behaviours, and we have seen here that that cannot be solved in a simple per-candidate-execution way, that the problem is not specific to relaxed atomics, that, while an operational solution for those examples is possible, it brings other difficulties, and that there are further problems with undefined behaviour.

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

[2] Note that this is not a thin-air execution, just a normal LB shape, with the reads and writes to x and y related by program order on the first thread and a data dependency on the second, extended just by using the read value of the first thread in an array access.

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.

Acknowledgements We would like to thank Hans Boehm, Paul McKenney, Jaroslav Ševčík, Ali Sezgin, Viktor Vafeiadis, and Francesco Zappa Nardelli for discussions about parts of this work. We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant), and a Gates Cambridge Scholarship (Nienhuis).

References

1. The SPARC architecture manual, v. 9. http://dev http://www.sparc.org/technical-documents/.
2. *Programming Languages — C.* 2011. ISO/IEC 9899:2011. http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf.
3. S. V. Adve and M. D. Hill. Weak ordering — a new definition. In *ISCA*, 1990.
4. J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM TOPLAS*, 36(2), 2014.
5. M. Batty. *The C11 and C++11 concurrency model.* PhD thesis, University of Cambridge, 2014. http://www.cl.cam.ac.uk/~mjb220/battythesis.pdf.
6. M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proc. POPL*, 2013.
7. M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, 2012.
8. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.
9. P. Becker, editor. *Programming Languages — C++.* 2011. ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.
10. H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, 2005.
11. H.-J. Boehm. Memory model rationales. http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html, March 2007.
12. H.-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm, September 2013.
13. H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.
14. H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proc. MSPC*, 2014.
15. G. Boudol and G. Petri. A theory of speculative computation. In *ESOP*, 2010.
16. P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.
17. D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *POPL*, 2013.
18. Free Software Foundation, Inc. RTL Passes — GNU Compiler Collection (GCC) Internals, October 2014. Available at https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html.
19. K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

20. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. 1996.
21. The HOL 4 system. `http://hol.sourceforge.net/`.
22. R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. ESOP*, 2010.
23. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.
24. LLVM Project. LLVM's Analysis and Transform Passes — LLVM 3.6 documentation, October 2014. Available at `http://llvm.org/docs/Passes.html`.
25. J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *POPL*, 2005.
26. L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. October 2012. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`.
27. D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *PLDI*, 2011.
28. Paul McKenney. Reordering and verification at the linux kernel reorder workshop in vienna summer of logic, july 2014. Invited talk at REORDER workshop, Vienna Summer of Logic, July 2014. `http://www2.rdrop.com/users/paulmck/scalability/paper/LinuxRCUVerif.2014.07.17a.pdf`.
29. R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. PLDI*, 2013.
30. B. Norris and B. Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proc. OOPSLA*, 2013.
31. W. Pugh. Fixing the Java memory model. In *Proc. ACM 1999 Conference on Java Grande*, 1999.
32. S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proc. PLDI*, 2012.
33. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.
34. J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.
35. P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010. (Research Highlights).
36. A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *Proc. ISCA*, 2012.
37. A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proc. OOPSLA*, 2014.
38. 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 *Proc. POPL*, 2015.
39. V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proc. OOPSLA*, 2013.
40. J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.
41. J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.