Symbolic MRD: Dynamic Memory, Undefined Behaviour, and Extrinsic Choice

JAY RICHARDS, University of Kent, United Kingdom
DANIEL WRIGHT, University of Surrey, United Kingdom
SIMON COOKSEY, Nvidia, United Kingdom
MARK BATTY, University of Kent, United Kingdom

We present the first thin-air free memory model that admits compiler optimisations that aggressively leverage knowledge from alias analysis, an assumption of freedom from undefined behaviour, and from the extrinsic choices of real implementations such as over-alignment. Our model has tooling support with state-of-the-art performance, executing a battery of tests orders of magnitude quicker than other executable thin-air free semantics. The model integrates with the C/C++ memory model through an exportable semantic dependency relation, it allows standard compilation mappings for atomics, and it matches all tests in the recently published desiderata for C/C++ from the ISO.

CCS Concepts: • **Software and its engineering** → **Semantics**; **Concurrent programming languages**; *Software verification*; • **Computer systems organization** → *Multicore architectures.*

Additional Key Words and Phrases: thin-air problem, relaxed memory, C++, concurrent language semantics.

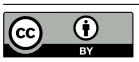
ACM Reference Format:
Jay Richards, Daniel Wright, Simon Cooksey, and Mark Batty. 2025. Symbolic MRD: Dynamic Memory, Undefined Behaviour, and Extrinsic Choice. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 146 (April 2025), 32 pages. https://doi.org/10.1145/3721089

1 Introduction

Recent work on concurrency models for systems languages has been preoccupied with solving the *thin-air problem* [6–8, 16, 19, 28], where deficiencies in the language specification either reject compiler optimisations or allow nonsensical values to be conjured from cyclic reasoning. The thin-air problem is but one aspect of building a programming model for a systems language that can accommodate optimising compilers. Optimisations leverage constraints implicit in the use of dynamic memory, implicit in the programmer's responsibility to avoid undefined behaviour (UB), and extrinsic constraints entirely separate from the program text. In this paper, we present a model that captures the implicit and extrinsic information used in optimisation. This model matches the International Organization for Standardization (ISO) desiderata for the C and C++ memory models [34], encompassing these hitherto unmodelled details of compiler optimisation, as well as solving the thin-air problem.

One way in which the thin-air problem has been addressed is by using *semantic dependencies* [17, 25, 28]: a relation over memory accesses identifying the dependencies that are left in place by compiler optimisations. Thus far, models using semantic dependencies have only partly bridged the gap as they still disallow many optimisations [17, 23] and work only on concrete locations.

Authors' Contact Information: Jay Richards, University of Kent, Canterbury, United Kingdom, jr636@kent.ac.uk; Daniel Wright, University of Surrey, Guildford, United Kingdom, daniel.w.0108@gmail.com; Simon Cooksey, Nvidia, Canterbury, United Kingdom, scooksey@nvidia.com; Mark Batty, University of Kent, Canterbury, United Kingdom, m.j.batty@kent.ac.uk.



ACM 2475-1421/2025/4-ART146
https://doi.org/10.1145/3721089

Undefined behaviour. Consider an example where undefined behaviour allows the optimiser to remove a syntactic dependency from a program. Here, as in all litmus tests that follow, we assume initial writes setting `x = y = 0`. The statement `y = 1 / !r1` only has defined behaviour when the division is by a non-zero value, so for the program to be well-defined, `r1` must equal 0.

```
1 int r1 = x;     ||  3 int r2 = y;
2 y = 1 / !r1;    ||  4 x = r2;
```

Example 1.1a: LB+UB+data

□

```
1 int r1 = x;     ||  3 int r2 = y;
2 y = 1;          ||  4 x = r2;
```

Example 1.1b: LB+UB+data

No restriction is imposed upon a program with a UB failure like division by zero: the semantics is described as *catch fire* because that is an allowed outcome. The optimiser uses these corners of the program more constructively: it assumes that these failures will be avoided in execution, and leverages this information in its analysis prior to optimisation. Here, we may assume that `!r1` is 1, and thus the write to `y` may be optimised to a constant write of 1. In writing 1 rather than an expression derived from `r1`, this transformation has broken the dependency from the read of `x` to the write of `y`, producing the program on the right above. The apparent data dependency in the left-hand thread of our original program was a *false dependency*, and the target is now free to reorder.

Are transformations that leverage undefined behaviour correct? If a transformation only changes the program in cases that lead to undefined behaviour, then it is sound because either: the transformation is over dead code in a well-defined program; or the whole program has undefined behaviour, and any transformation would be valid. Indeed, we see this optimisation performed by both GCC and Clang[1].

Example 1.2: LB+deps and LB+po+dep

The behaviour of Example 1.1a is expressed in C and C++ as a set of *execution graphs*, with nodes representing memory accesses and labelled edges representing program order, ⊑, and reads from, rf. Example 1.2a displays a possible execution graph for the program above. The best formalisation of the C/C++ model is RC11 [21], which is faithful to C/C++ in all but one respect: RC11 forbids cycles in ⊑ ∪ rf and C/C++ does not. In our example, RC11 would forbid the outcome `r1 = 1 && r2 = 1` because of the cycle, whereas C/C++ would allow it. Unfortunately, forbidding the outcome, as in RC11, makes the transformation to a constant write of y unsound. In contrast, allowing all ⊑ ∪ rf cycles, as in C/C++, permits erroneous thin-air behaviour [6, 19, 21].

There has been significant recent work to define a thin-air free semantics for C/C++ that permits aggressive optimisations [8, 16, 19, 23, 29]. One strategy is to identify dependencies in the program that constrain where the implementation can re-order memory accesses [17, 28]. We have annotated the executions above with such a dependency relation, dp. Note, that after the compiler transformation from Example 1.1a to Example 1.1b the dependency in the left-hand thread has been removed: the dependency that appeared in the syntax of the original program was a *false dependency*. By ruling out cycles in dp ∪ rf we can forbid thin-air executions and allow compiler optimisations. The devil is of course in the detail: we must precisely define dp.

To match the behaviour of its compilers, the C/C++ memory model must admit the outcome `r1 = 1 && r2 = 1`. The optimisation leveraged the possibility of division leading to undefined

[1]Many of the examples of compiler optimisations presented in this paper include links to Compiler Explorer [14], that can be used to see the optimisation in action.

behaviour, but this sort of reasoning is not well handled by any existing model — not in models that attempt to define a dp for inclusion in an axiomatic semantics, nor in operational models like the Promising Semantics or its successors [19, 23]. Indeed, despite the advance it represents, the Promising Semantics 2 [9, 23] incorrectly attributes UB to Example 1.1a (see Appendix F).

Extrinsic choice. In the program above, the compiler derived information about the value of a part of the program from its syntax: the denominator of the division could not be zero, because that would be undefined. The compiler can use additional constraints when optimising, beyond what can be derived from the program syntax alone — we call these *extrinsic choices*. For example, the C language specification leaves compilers free to over-align objects in memory. In the following example, the write appears to be dependent on the 16-byte alignment of the object pointed to by p. The extrinsic choice to over-align removes this dependency and admits a reordering.

```
1 int* r1 = p;
2 if (r1 % 16 == 0)
3     y = 1;
```

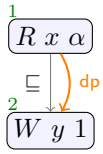
Example 1.3a: alignment

□

```
1 int* r1 = p;
2 y = 1;
```

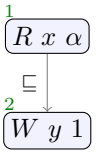
Example 1.3b: alignment

Here, the optimisation was contingent on a choice made *entirely* separately from the code. We can look at the executions again, and see how the compiler's extrinsic choice removed a dependency from the execution graph. This thread can build an LB pattern similarly to Example 1.1a, and then the user can observe a non-SC outcome.



Example 1.3c: alignment

□



Example 1.3d: alignment

Dynamic memory. When a region of memory is allocated, languages like C guarantee disjointness with existing objects [13, [basic.stc.dynamic.allocation]], and this information can be used by the compiler to resolve questions of aliasing. In the following example, if *p and x alias, then the accesses of Lines 3 and 4 are at the same location and cannot be reordered by the compiler. However, the allocation of p provides a guarantee of disjointness that the compiler can leverage to reorder as follows.

```
1 atomic int x = 0;
2 atomic int* p = malloc(sizeof(int));
3 int r1 = *p;     ||  5 int r2 = x;
4 x = 1;           ||  6 *p = r2;
```

Example 1.4a: LB+alias+data

□

```
1 atomic int x = 0;
2 atomic int* p = malloc(sizeof(int));
3 x = 1;           ||  5 int r2 = x;
4 int r1 = *p;     ||  6 *p = r2;
```

Example 1.4b: LB+alias+data

The memory behaviour of this program is contingent on whether $*p$ and x alias, so the memory model must take this into account.

1.1 Criteria for a Programming Language Memory Model

A recent ISO white paper [34] outlines desiderata for semantic dependencies along with a suite of examples. Each of these examples probe at the definition of semantic dependencies and refine it based upon the expectations of programmers and compiler implementers. Given this, any model for these languages must satisfy the following criteria.

(C1) Be able to reason about dynamic memory. This includes allocation, dereferencing, and reclamation.

(C2) Avoid throwing out any optimisation *at all*. This includes optimisations that: fuse successive stores, fuse successive loads, de-duplicate repeated stores, eliminate dead code, and hoist invariant stores.
(C3) Be aware of guarantees and invariants extrinsic to the program syntax, as semantic dependencies can rely on compiler assumptions about alignment or undefined behaviour, and on global analysis.

In this paper we will present a model using semantic dependencies that meets all of these criteria as well as two additional criteria.

(C4) Provide a tool for automatically deriving the semantic dependencies for a given program.
(C5) Provide a *justification* structure that can be interrogated to explain *why* a particular outcome is allowed.

Throughout this paper we will work through examples, each of which will motivate our justification structure. We will first introduce justifications and their elaboration as a precursor to the derivation of semantic dependencies.

1.2 Justification, Elaboration, and Executions

We describe a new thin-air free memory model called *Symbolic MRD* (sMRD), built on prior work [28]. sMRD interfaces with axiomatic models like RC11 where a set of executions are derived from the program text and filtered by a collection of axioms. sMRD intervenes on the axiomatic model in the generation of executions. Typically these are built from the program syntax with a simple enumeration of the read and write events in each path of control flow. sMRD replaces this simple enumeration with an extensive calculation of the dependencies inherent in each thread, communicating these to the axiomatic model as a dependency relation, dp, attached to each prospective execution.

The key difference from a typical axiomatic model is that executions must be *justified*: we must choose a set of *justifications* that covers the writes, recording — for each write — the read and allocation events that it depends on. This justifying set will be used to derive dp, and thin-air behaviour will be forbidden with a new axiom that forbids cycles in $\mathsf{dp} \cup \mathsf{rf}$.

We build a set of justifications, $\mathbb{J}$, that represent all the ways in which each write might occur. We populate $\mathbb{J}$ initially with $\mathbb{J}_0$, the justifications derived directly from the program syntax. We extend this initial set through a series of *elaborations*. Each elaboration accommodates a class of compiler transformations, and adds new justifications to $\mathbb{J}$. When we choose the justifying set, J_X, for an execution, X, it must be a subset of $\mathbb{J}$.

Read events and allocations introduce symbolic values that will be resolved through constraints applied later. These symbols are used in the representation of control and data dependencies. An *origin* function maps the symbolic values back to the reads and allocations that introduced them.

Justifications take the form, $(P, D) \vdash^{\delta} w$ where:

- P, *control dependency*, is a symbolic predicate collecting the conditions enforced in the choice of control flow that lead to the write w;
- D, *data dependency*, is a set of symbols present in the expressions for the address and value of w;
- δ, the *forwarding context*, captures pairs of accesses that have been fused through forwarding and elision;
- and, w is the write that is being justified.

We drop the δ and write $(P, D) \vdash w$ when the forwarding context is empty.

The justifications of an execution are flattened into a *semantic dependency relation*, dp, by taking each justification; collecting the origins of all symbols in P and D; and taking the cross product with the write w, making each read a dependency of the write.

Data dependencies. To calculate the behaviour of Example 1.1a, we begin by deriving its initial justifications. On each thread, a read introduces a new *symbol* for the value read, and the write stores the value of an expression that uses this symbolic value. Thread 1 introduces α when loading from x on Line 1, and Thread 2 introduces β when loading from y on Line 3. In future, symbol names will be chosen to match the program, but remain distinct from the syntactic variables. The initial justifications for Example 1.1a are:

$$(\top, \{\alpha\}) \vdash (2\colon W\, y\, 1/!\alpha) \qquad (\top, \{\beta\}) \vdash (4\colon W\, x\, \beta)$$

The justifications record α and β, respectively, as the data dependencies of the writes. $\mathbb{J}_0$ is sufficient to justify an execution of Example 1.1a. We derive dp from this pair of justifications to produce the execution in Example 1.2a. This execution has a cycle in dp ∪ rf and is excluded by the axiomatic constraints.

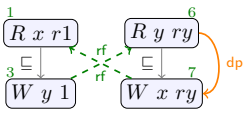
A series of elaboration steps (explained in detail in the following sections) introduces a further justification, $(\top, \emptyset) \vdash (2\colon W\, y\, 1)$. This justification applies to the store in Line 2, but the write event in the justification uses a constant value of 1, rather than the expression $1/!\alpha$. This change mirrors the observation that the only defined value for the division is 1. This justification is added to $\mathbb{J}_0$ to produce the set $\mathbb{J}$.

Now there is a choice in how we justify an execution of Example 1.1a. Choosing $(\top, \emptyset) \vdash (2\colon W\, y\, 1)$ and $(\top, \{\beta\}) \vdash (4\colon W\, x\, \beta)$ to justify the writes, and then deriving dp, leads to the execution in Example 1.2b, with no dependency in Thread 1. This execution has no cycle in dp ∪ rf and is allowed by the axiomatic constraints, accommodating the optimisation performed by GCC and Clang that leverages the possible undefined behaviour inherent in the division.

a)

```
1 int r1 = x;
2 if (r1 == 1)
3   y = r1;
4 else
5   y = 1;
   ||
6 int ry = y;
7 x = ry;
```

b)



Example 1.5: LB+vafalsedep

Control dependencies. Example 1.5a features a conditional with a write in each branch. The branch leads to two initial justifications: the first for the write in the true branch, whose predicate, $r1 = 1$, records the symbolic constraint implicit in taking the first branch; and the second, similarly, for the false branch. The third initial justification is for the write on the second thread. The initial set of justifications, $\mathbb{J}_0$, is given below.

$$(r1 = 1, \{r1\}) \vdash (3\colon W\, y\, r1) \qquad (r1 \neq 1, \emptyset) \vdash (5\colon W\, y\, 1) \qquad (\top, \{ry\}) \vdash (7\colon W\, x\, ry)$$

A compiler may optimise the program: first recognising that the conditional constrains the value written on Line 3 to 1, and second recognising that the store is invariant on the value of r1. There is no real dependency in the first thread. Steps of elaboration mirror these transformations: the symbol $r1$ is assigned the value 1, according to the predicate carried by the first initial justification. Then an elaboration recognises equivalent writes under predicates $r1 = 1$ and $r1 \neq 1$, adding justifications that unify these predicates to $\top$, removing the control dependencies.

$$(r1 = 1, \emptyset) \vdash (3\colon W\, y\, 1) \qquad (\top, \emptyset) \vdash (3\colon W\, y\, 1) \qquad (\top, \emptyset) \vdash (5\colon W\, y\, 1)$$

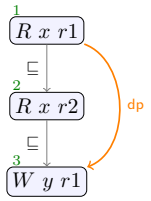
Now we can justify an execution of the program using $(\top, \emptyset) \vdash (3\colon W\, y\, 1)$ and $(\top, \{ry\}) \vdash (7\colon W\, x\, ry)$ to produce the execution in Example 1.5b, with no dependency in the first

thread, allowing the behaviour, and by extension the motivating series of optimisations. We see this series of optimisations being applied in both GCC and Clang.

a)

```
1 int r1 = x;
2 int r2 = x;
3 y = r2;
```

b)



Example 1.6: FWD

Forwarding context. Example 1.6a has two successive reads, Lines 1 and 2, with no intervening events, followed by a write of the second value read. The initial justification is on the left below, and it writes value $r2$. A compiler might fuse the two reads, performing only the first. The value of the first read is then forwarded to the write. Elaboration produces the second justification, and it writes the value of the first read, $r1$.

$$(\top, \{r2\}) \vdash (3\colon W\, y\, r2) \qquad (\top, \{r1\}) \vdash^{\{(1,2)\}} (3\colon W\, y\, r1)$$

The elaborated justification carries a forwarding context, $\{(1, 2)\}$, that records that the load on Line 1 has been forwarded to elide the load on Line 2. In justifying an execution, we must choose a set of justifications with consistent forwarding contexts.

2 Compiler Optimisations

In this section, we present a series of example programs. For each, we will discuss the optimisations that may be applied and the mechanisms in sMRD that decide the behaviours of the test. Our first examples cover the list of elaborations: value assignment, weakening, strengthening, forwarding, and lifting. We close the section with a discussion of Example 2.8a, whose behaviour is contingent on a contentious series of compiler optimisations. Our elaborations are abstract enough to capture broad classes of optimisations, and extrinsic constraints can capture implementation-defined choices. Our model allows all optimisations described by McKenney for ISO WG21, C++ [34], but if new optimisations were recognised, and they did not fit within this machinery, then the set of elaborations could be extended.

Value assignment. In Example 1.5a, the conditional on Line 2 constrained the value of `r1` to 1. This observation can be used to transform the write on Line 3, from a write of `r1` to a write of constant 1. sMRD accommodates this with the value assignment elaboration. The initial justification of write 3 is given on the left below, and has a predicate $r1 = 1$. The predicate is sufficient to resolve the value of symbol $r1$ to constant 1. The value assignment elaboration introduces a new justification of write 3, on the right below, with refined value and location expressions: the constant 1 is substituted for symbol $r1$ in each.

$$(r1 = 1, \{r_1\}) \vdash (3\colon W\, y\, r1) \quad \overset{\text{va}}{\square} \quad (r1 = 1, \emptyset) \vdash (3\colon W\, y\, 1)$$

This new justification can be further elaborated to remove the control dependency on $r1$ in a step called lifting.

Lifting. With the new justification, $(r1 = 1, \emptyset) \vdash (3\colon W\, y\, 1)$, gained through value assignment, we can now apply a *lifting* elaboration. Lifting operates on pairs of justifications for writes with equivalent locations and values: it merges their predicates to form a new justification. In Example 1.5a, the initial justifications and one step of value assignment have given the justifications of writes 3

and 5 on the left below. Lifting combines the predicates of the two justifications and introduces new justifications with the predicate $r1 = 1 \vee r1 \neq 1$, or equivalently $\top$.

$$\begin{array}{l}(r1 = 1, \emptyset) \vdash (3\colon W\, x\ 1)\\ (r1 \neq 1, \emptyset) \vdash (5\colon W\, x\ 1)\end{array} \overset{\text{lift}}{\square} \begin{array}{l}(r1 = 1 \vee r1 \neq 1, \emptyset) \vdash (3\colon W\, x\ 1)\\ (r1 = 1 \vee r1 \neq 1, \emptyset) \vdash (5\colon W\, x\ 1)\end{array} \square \begin{array}{l}(\top, \emptyset) \vdash (3\colon W\, x\ 1)\\ (\top, \emptyset) \vdash (5\colon W\, x\ 1)\end{array}$$

Weakening and global guarantees. Compilers often take advantage of information extrinsic to the program syntax, and we use Ω to represent these constraints.

```
1 int r1 = x;
2 if (r1 <= INT_MAX)
3   y = 1;
```

Example 2.1a: INT_MAX

```
1 int r1 = x;
2 y = 1;
```

Example 2.1b: INT_MAX

The initial justification of Example 2.1a, $(r1 \leq \texttt{INT\_MAX}, \emptyset) \vdash (3\colon W\, y\ 1)$, recognises a control dependency on `r1`. One source of extrinsic guarantees are the value ranges for types, we know that an `int` must range between `INT_MIN` and `INT_MAX`. We record this knowledge in the program-wide guarantee: $\Omega \Rightarrow r1 \leq \texttt{INT\_MAX}$. The *weakening* elaboration adds a new justification with a weaker control dependency. This control dependency together with Ω must imply the control dependency of the original justification. Here, $\Omega \Rightarrow r1 \leq \texttt{INT\_MAX}$, so the new justification can break the dependency on `r1`, by using $\top$ on the right below. This new justification admits optimising Example 2.1a to Example 2.1b; both GCC and Clang perform this optimisation.

$$(r1 \leq \texttt{INT\_MAX}, \emptyset) \vdash (3\colon W\, y\ 1) \overset{\text{weak}}{\square} (\top, \emptyset) \vdash (3\colon W\, y\ 1)$$

```
1 int r1 = x;
2 if (!r1 != 0)
3   y = 1 / !r1;
4 else
5   y = 1 / !r1;
```

Example 2.2a: LB+UB+data

```
1 int r1 = x;
2 if (!r1 != 0)
3   y = 1;
4 else
5   y = 1 / !r1;
```

Example 2.2b: LB+UB+data

```
1 int r1 = x;
2 y = 1;
```

Example 2.2c: LB+UB+data

Strengthening and undefined behaviour. Returning to Example 1.1a, we relied on an assumption of UB freedom to justify the execution in Example 1.2b: $\Omega \Rightarrow !r1 \neq 0$, or equivalently, $\Omega \Rightarrow r1 = 0$. In order to allow the execution we must break the data dependency on `r1`. We cannot immediately perform a weakening elaboration as we do not have a control dependency, only a data dependency.

The strengthening elaboration introduces superfluous control dependencies: putting an arbitrary predicate in conjunction with the current control dependency. Performing a strengthening that constrains a symbol to a concrete value, and then performing a value assignment effectively transforms a data dependency into a control dependency.

To break the dependency on `r1`, we first perform a strengthening operation with predicate, $!r1 \neq 0$. We gain a new justification with this as a control dependency, on the right below. Example 2.2a is a rewrite of the program, introducing a branch corresponding to this new control dependency. This is for illustration only, the model merely elaborates the set of justifications.

$$(\top, \{r1\}) \vdash (2\colon W\, y\ 1\ /\ !r1) \overset{\text{str}}{\square} (!r1 \neq 0, \{r1\}) \vdash (2\colon W\, y\ 1\ /\ !r1)$$

The new justification above is a candidate for value assignment: $!r1 \neq 0 \Rightarrow r1 = 0 \Rightarrow !r1 = 1$, and we apply that elaboration below to remove the control dependency. This elaboration is consistent

with a syntactic rewrite to produce Example 2.2b.

$$(!r1 \neq 0, \{r1\}) \vdash (2\colon W\ y\ 1\ /\ !r1) \quad \overset{\text{va}}{\square} \quad (!r1 \neq 0, \emptyset) \vdash (2\colon W\ y\ 1)$$

Finally, we apply a weakening elaboration to remove the control dependency: we already know $!r1 \neq 0$ from Ω. This is consistent with a syntactic transformation of Example 2.2b to Example 2.2c.

$$(!r1 \neq 0, \{r1\}) \vdash (2\colon W\ y\ 1\ /\ !r1) \quad \overset{\text{weak}}{\square} \quad (\top, \emptyset) \vdash (2\colon W\ y\ 1)$$

The final step of elaboration adds an independent justification of the write, removing all dependencies and allowing the optimisation to Example 1.1b. This optimisation is performed by both GCC and Clang.

We can apply strengthening, value assignment, and weakening to produce executions where the value of a register appears to be inconsistent in multiple uses within a thread. Consider Example 2.3a, where a write of z has been added to the first thread of Example 1.1a.

```
1 int r1 = x;
2 y = 1 / !r1;
3 z = r1;
```
‖
```
4 int r2 = y;
5 x = r2;
```

Example 2.3a: LB+UB+data+z

```
1 int r1 = x;
2 y = 1;
3 z = 0;
```
‖
```
4 int r2 = y;
5 x = r2;
```

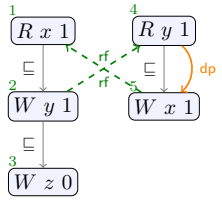
Example 2.3b: LB+UB+data+z

In this test, strengthening, value assignment and weakening lead to the same independent justification of the write on Line 2: $(\top, \emptyset) \vdash (2\colon W\ y\ 1)$. The initial justification of the write on Line 3 is given below on the left, together with its elaborated justifications:

$$(\top, \{r1\}) \vdash (3\colon W\ z\ r1) \quad \overset{\text{str}}{\square} \quad (!r1 \neq 0, \{r1\}) \vdash (3\colon W\ z\ r1)$$

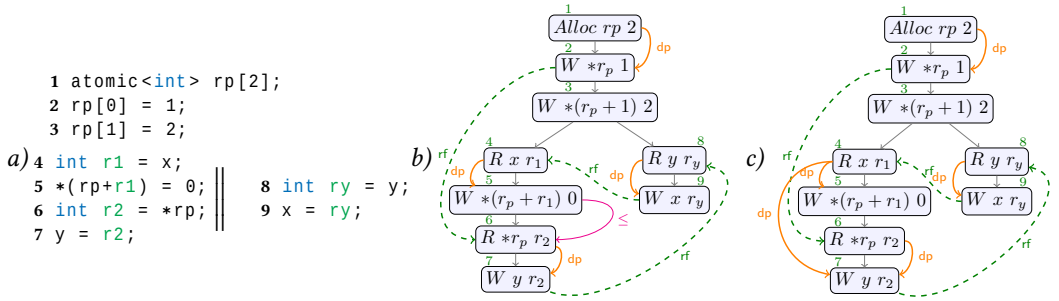
$$\overset{\text{va}}{\square} \quad (!r1 \neq 0, \emptyset) \vdash (3\colon W\ z\ 0) \quad \overset{\text{weak}}{\square} \quad (\top, \emptyset) \vdash (3\colon W\ z\ 0)$$

These elaborations are consistent with the transformation of Example 2.3a to Example 2.3b. Elaboration has added independent justifications $(\top, \emptyset) \vdash (2\colon W\ y\ 1)$ and $(\top, \emptyset) \vdash (3\colon W\ z\ 0)$ to $\mathbb{J}$, and with these we can justify the execution in Example 2.3c.



Example 2.3c: LB+UB+data+z

Example 2.3c features *inconsistent* values of `r1`: the first read has value 1, consistent with `r1` taking value `1`, and write 3 to z has value 0, consistent with `r1` taking value `0`. No execution of Example 2.3a exhibits undefined behaviour: this program does not have catch-fire semantics. We do not observe the optimisation directly in GCC or Clang, but GCC and Clang, do optimise Example 1.1a similarly. The consequence of optimising in this example is *surprising*: a local variable appears to have taken two different values in one execution. sMRD can be made to reject this behaviour, but we believe it is a natural consequence of the optimisations that GCC and LLVM perform and that it should be allowed.



Example 2.4: JCTC12

Pointers. So far our events have only had concrete locations; in order to support pointers we also accept symbolic ones. This brings with it several considerations. In Example 2.4a two pointers, on Lines 5 and 6, may *alias*: the expressions for the addresses that are accessed may be equal. Questions of aliasing fall into three cases:

(1) Two pointers always point to the same location. Under the context in which they are used, their expressions cannot be resolved to differing values.
(2) Two pointers possibly point to the same location. Their expressions can be resolved to the same value, but also to differing values.
(3) Two pointers never point to the same location. Their expressions cannot be resolved to the same value.

The first case is simple, they always alias, and the resulting events are at the same location, thus we can never reorder them. The third case is simple too, they never alias, thus we can always reorder the events. The second case is more nuanced: we cannot establish whether the locations alias or not. We initially assume that the pointers alias and keep them in order, although elaboration can transform the way this order is recorded, as we shall see.

Example 2.4b depicts the execution where we cannot rule out that the two pointers alias. This execution is the first we have covered that features a *preserved program order* edge, $\leq$. $\leq$ edges order events at the same location, as is the case here, but also order fences and synchronising events. Previous work [17, 28], folded the ordering captured by $\leq$ into dp, but we keep it separate. $\leq$ edges are added to dp and rf in the axiom that we use to rule out thin-air values, which becomes acyclic(dp $\cup \leq \cup$ rf). The execution in Example 2.4b features a dp $\cup \leq \cup$ rf cycle and is forbidden by the model.

Elaboration, and particularly strengthening, can be used to exclude the possibility that the two pointers alias. We can strengthen the predicate P in the justification of write event 7 with the property $r1 \neq 0$. This choice of elaboration removes the $\leq$ edge: the pointers no longer alias. It also changes dp, inducing a dependency from the origin of $r1$, read event 4, to write event 7. This elaboration leads to the execution in Example 2.4c, without a $\leq$ edge from event 5 to 6, but with a new dp edge from event 4 to event 7. This execution is also forbidden: the new dp edge completes a cycle in dp $\cup$ rf.

```
1 atomic<atomic<int>*> p = new int;
2 int* rp = p;     ||  5 int r2 = x;
3 *rp = 1;         ||  6 if (r2 = 1)
4 x = 1;           ||  7   free(p);
```

Example 2.5a: Free Race

```
1 atomic<atomic<int>*> p = new int;
2 int* rp = p;     ||  5 r2 = x.load(acq);
3 *rp = 1;         ||  6 if (r2 == 1)
4 x.store(1,rel);  ||  7   free(p);
```

Example 2.5b: Free Race

Introducing pointers also reintroduces a dynamic source of undefined behaviour: allocations, reclamations, and accesses can race with each other. We can use the derived relations of RC11, and

in particular *happens-before* [21], to identify these races. If we have an allocation and a reclamation or access to the same location that are not ordered by happens before we have a race. We also have a race when we have a reclamation and an access to the same location not ordered by happens before.

```
1 int r1 = x;
2 int r2 = x;
3 if (r1 == 1)      ||   7 int ry = y;
4   y = r2;         ||   8 x = ry;
5 else
6   y = 1;
```

Example 2.6a: LB+fwd

Load forwarding. In Example 2.6a the compiler might fuse the loads of `x` in lines 1 and 2, performing only the first load and reusing the value in Line 2. sMRD includes a *forwarding* elaboration step that reflects this possibility. Forwarding unifies the symbols of the fused loads, replacing all occurrences of `r2` with `r1`, and recording the pair $(r1, r2)$ in the forwarding context. After the elaboration we have a set of justifications that matches Example 1.5a. This optimisation is not currently performed in the concurrent context by either GCC or Clang, but it is valid under the specification of C++.

$$(r1 = 1, \{r2\}) \vdash^{(\emptyset,\emptyset)} (3\colon W\ y\ r2) \quad \overset{\text{fwd}}{\square} \quad (r1 = 1, \{r1\}) \vdash^{(\{(1,2)\},\emptyset)} (3\colon W\ y\ r1)$$

Further elaborations provide: store forwarding, store-store forwarding, and write elision. These are similar in nature to load forwarding and are explained in Appendix A.

```
1 int r1 = x;
2 if (r1 == 1) {
3   int rw1 = w;
4   z = rw1;        ||   9 int ry = y;
5 } else {          ||  10 x = ry;
6   int rw2 = w;
7   z = rw2;
8 }
```

Example 2.7a: Lift

```
1 int r1 = x;
2 if (r1 == 1) {
3   int rw1 = w;
4   int ry = y;     ||  10 int ry = y;
5   z = rw1;        ||  11 x = ry;
6 } else {
7   int rw2 = w;
8   z = rw2;
9 }
```

Example 2.7b: Lift

Lifting. We will now explore the matching of branch bodies and the hoisting of their events. In Example 2.7a, regardless of the value read into `r1` the constraints on the value read from `w` are equivalent. The compiler might recognise this symmetry and rewrite `rw1` to `rw2` (or vice versa) to produce equivalent branch bodies and then hoist above the condition. This optimisation is performed by both GCC and Clang.

The *Lifting* elaboration applies to justifications of writes with equivalent locations and values: here $(r1 = 1, \{3\}) \vdash (4\colon W\ z\ rw1)$ and $(r1 \neq 1, \{6\}) \vdash (7\colon W\ z\ rw2)$. Lifting adds justifications for each write with a predicate formed of the disjunction of the predicates of the original justifications. In this example, the resulting justifications are independent, and the relaxed behaviour is allowed.

$$\begin{array}{l}(r1 = 1, \{3\}) \vdash (4\colon W\ z\ rw1) \\ (r1 \neq 1, \{6\}) \vdash (7\colon W\ z\ rw2)\end{array} \quad \overset{\text{lift}}{\square} \quad \begin{array}{l}(r1 = 1 \vee r1 \neq 1, \{3\}) \vdash (4\colon W\ z\ rw1) \\ (r1 = 1 \vee r1 \neq 1, \{6\}) \vdash (7\colon W\ z\ rw2)\end{array} \quad \square \quad \begin{array}{l}(\top, \{3\}) \vdash (4\colon W\ z\ rw1) \\ (\top, \{6\}) \vdash (7\colon W\ z\ rw2)\end{array}$$

Example 2.7b introduces a read on Line 4. This read has no impact on the justification of the write of `z` in that branch, lifting still applies, and the relaxed outcome is allowed.

There has been discussion around whether a semantic dependency relation can describe global analysis and load introduction. This is best demonstrated in the following example, presented by Lahav at the Future of Weak Memory (FoWM) workshop [20].

```
1 int a = X;
2 if (a == 1) {
3   Y = a;
4   print("foo");
5 } else {
6   int b = Z;
7   Y = b;
8 }
```

```
9 int r = Y;
10 X = r;
```

```
11 Z = 1;
```

Example 2.8a: Lahav - FoWM'24

Example 2.8a has three threads: the first thread prints `foo` if we read 1 from X, this sets up the first half of our LB shape. The second thread sets up the second side of the LB shape. Finally, the third writes a 1 to Z. It has been argued that printing `foo` should be an allowed outcome [20]. A chain of optimisations to thread 1 of this program under Clang could in theory permit printing `foo`:

(OPT1) introduce a redundant load of Z;
(OPT2) introduce a branch on the result of that load;
(OPT3) forward the result of the load into the branch;
(OPT4) and finally hoist an invariant store.

The examples below are annotated with each of these steps.

```
1 c = Z; //  OPT1
2 int a = X;
3 if (a == 1) {
4   Y = a;
5   print("foo");
6 } else {
7   int b = Z;
8   Y = b;
9 }
```

□

Example 2.8b: Lahav - FoWM'24

```
1 c = Z;
2 if (c == 1) { //  OPT2
3   int a = X;
4   if (a == 1) {
5     Y = a;
6     print("foo");
7   } else {
8     int b = Z;
9     Y = b;
10  }
11 } else {  ...  }
```

□

Example 2.8c: Lahav - FoWM'24

```
1 c = Z;
2 if (c == 1) {
3   int a = X;
4   if (a == 1) {
5     Y = 1;  //  OPT3
6     print("foo");
7   } else {
8     int b = 1; //  OPT3
9     Y = 1;
10  }
11 } else {  ...  }
```

□

Example 2.8d: Lahav - FoWM'24

```
1 c = Z;
2 if (c == 1) {
3   Y = 1; //  OPT4
4   int a = X;
5   if (a == 1) {
6     print("foo");
7   }
8 } else {  ...  }
```

Example 2.8e: Lahav - FoWM'24

When we finish these optimisations and reach the transformed program in Example 2.8e, we can see that composing this thread with Threads 2 and 3 of the original program in Example 2.8a can allow the program to print `foo`. In practice, we cannot observe this optimisation with any compiler we tried including Clang.

The first step of this chain of optimisations, load introduction OPT1, is contentious, and it is contentious whether Z is atomic or non-atomic. In both cases, load introduction can allow behaviours forbidden by the C/C++ semantics [34]. There is a note in the C++ specification to this effect [13, [intro.races]]. We are of the opinion that unconstrained load introduction should be forbidden.

Even so, we consider some forms of load introduction to be benign, and we recognise that implementations do perform benign load introduction. Consider the transformation demonstrated below.

```
1 int r1 = 0;
2 for (int i = 0; i < n; ++i) {
3   r1 += x;
4 }
```

Example 2.9a: Load introduction from load fusion

□

```
1 int r1 = n * x;
```

Example 2.9b: Load introduction from load fusion

This transformation has the effect of introducing a load of x in the case that n = 0. However, even though a new load has been performed in the n = 0 case, its value is nullified by the multiplication by zero. In the n = 0 case, there is an irrelevant load introduction, and in the n != 0 case there is a load fusion. This distinction makes this introduction tantamount to a hoist of a load rather than an introduction. It is the *use* of the hoisted load that would be folly, but that is neatly avoided here.

Symbolic MRD forbids unconstrained load introduction for two reasons. Firstly, and most simply, the introduction of new events is not possible in the elaboration mechanism. Secondly, one might attempt to use strengthening to introduce a dependency from the load of Z on Line 6, to the store of Y on Line 3. This is also not possible, because the read is not available in an execution where the store on Line 3 is present.

3 Symbolic MRDer: Decidable Automatic Evaluation

As with any complex model of a programming language, hand evaluation is error-prone and time-consuming. We have built a tool, Symbolic MRDer, to automatically evaluate programs under the sMRD model. This has been co-developed with the mathematical definition of the semantics, helping us to simplify rules and ensure that modifications have not introduced regressions. Tooling is essential to validate modelling choices and to help build confidence that a model adequately handles litmus tests from the literature. Indeed, developing memory models alongside tooling is now a well trodden practice. Several other thin-air free semantics have received this same treatment [10, 17, 23, 28] and tool development is completely typical for axiomatic semantics [3, 7, 11, 24, 35].

Symbolic MRDer takes programs as input and allows us to configure parameters for the test. Expected behaviour is defined in an assertion, extrinsic choices can be specified, and consistency is selected between RC11 [21] and IMM [30] modified to use our an acyclic($\mathsf{dp} \cup \leq \cup \mathsf{rf}$) axiom instead of the typical acyclic($\sqsubseteq \cup \mathsf{rf}$) axiom. Symbolic MRDer returns program behaviour as undefined, or if defined, as a set of executions, and it indicates if the assertion on expected behaviour is met. Symbolic MRDer presents these executions graphically, printing event structures, dependencies, and allowing interrogation of justifications. It is written in JavaScript and runs entirely within a web browser, using Z3 via WebAssembly [12].

The semantics is declarative, producing a set of justifications, generating executions from these, and then filtering to the allowed executions. In practice, the naïve generation of executions produces an intractably large set, so Symbolic MRDer filters this set eagerly. Predicates in justifications are maintained in conjunctive normal form, splitting disjunctions into separate justifications. Unsatisfiable justifications are removed, and so are those that are strictly stronger than others.

The simultaneous development of the tool and the semantics led to semantic choices in favour of tool performance. In fact, we have established that the semantics is decidable (see Appendix G).

3.1 Evaluation

Symbolic MRDer evaluates a suite of 173 litmus tests, drawn from literature discussing the specification of C/C++ [34] and Java [31], on the thin-air problem [6, 8, 16, 19, 29, 33], and on weak memory [21, 30, 32]. Each test probes one memory behaviour, specifying the expectation for C++

in its assertion. The run-time is orders of magnitude faster than other thin-air free semantic model checkers: the whole test suite is completed in less than two minutes on an M2 MacBook Pro. Similar tools for other thin-air free models take many minutes to hours to complete a subset of the test suite [17, 23, 28]. Symbolic MRDer provides the desired behaviour in all test cases, except where open questions of thread inlining and optimisations using global analysis leave the desired behaviour ambiguous. Whilst several of our litmus tests are larger than previous models could reasonably evaluate, they are still small relative to real codebases. Finding a way to scale this sort of analysis is left to future work.

4 Formal Model

We introduce Symbolic MRD by modelling the following language. Registers, $\mathbb{R}$, are ranged over by $r_1, r_2, \ldots$ and global variables, $\mathbb{L}$, are ranged over by $\mathtt{x}, \mathtt{y}, \mathtt{z}$ all of which are disjoint: $\mathtt{x} \neq \mathtt{y}$, $\mathtt{y} \neq \mathtt{z}$, and $\mathtt{x} \neq \mathtt{z}$. Memory orderings, $o \in \{rlx,\ rel,\ acq,\ sc\}$, are used to annotate fences, loads, and stores (following C/C++). Expressions, $\mathcal{E}$, are made up of registers, values, unary and binary operators, and we use ε to denote an expression.

Definition 4.1. (Grammar).

$$
\begin{aligned}
P ::= {} & \mathtt{skip} \mid P_1; P_2 \mid P_1 \parallel P_2 \mid \mathtt{if}\ (\varepsilon)\ \{P_1\}\ \mathtt{else}\ \{P_2\} \mid \mathtt{while}\ (\varepsilon)\ \{P\} \\
& \mid r_i := \varepsilon \mid r_i :=_o x \mid x :=_o \varepsilon \mid r_i := \&\varepsilon \mid r_i :=_o {}^*\varepsilon \mid {}^*\varepsilon_1 :=_o \varepsilon_2 \\
& \mid \mathtt{fence}_o \mid r_i := \mathtt{fadd}_{o_r,o_w}(l, \varepsilon) \mid r_i := \mathtt{cas}_{o_r,o_w}(l, \varepsilon_1, \varepsilon_2) \\
& \mid r_i := \mathtt{malloc}(\varepsilon) \mid \mathtt{free}(r_i)
\end{aligned}
$$

Programs will be interpreted into event structures.

Definition 4.2. A *symbolic event structure* is a tuple $(\mathbb{E}, \sqsubseteq, \#, \sqsubseteq^{\mathsf{rmw}}, \vee)$ where

(a) $\mathbb{E}$ is a set of labelled events.
(b) *Program order* over events, $\sqsubseteq$, mirrors the syntactic order of the source program.
(c) *Conflict*, $\#$, denotes events that cannot both appear in a single execution, chiefly events that occur in opposing branches of control flow.
(d) *Atomicity*, $\sqsubseteq^{\mathsf{rmw}}$, is a subset of program order that will exclude events from intervening on read-modify-write operations.
(e) *Value restriction*, $\vee$, is a function from events to boolean expressions, tracking constraints on symbolic values gathered through control flow choices.

An event is one of the following:

- a write with order o, $W_o\ x\ \varepsilon$, where x and ε are expressions for the location and value written;
- a read with order o, $R_o\ x\ s$, where x is an expression for the location read, and s is a symbol, introduced by the read, representing the read value;
- a memory fence with order o, F_o;
- a branch, $[\varepsilon]$, where ε is the expression that we branch upon;
- a memory allocation, $Alloc\ s\ \varepsilon$, where ε is the size of the allocation, and s is a symbol, introduced by the allocation disjoint from all $l \in \mathbb{L}$, representing the pointer to the allocated region; or
- a memory reclamation, $Free\ \varepsilon$, where ε is the location freed.

An event, e, is associated with a unique label, l, denoted $(l : e)$. Several maximal subsets are defined on $\mathbb{E}$: $\mathcal{W}, \mathcal{R}, \mathcal{F}, \mathcal{B}, \mathcal{A}, \mathcal{C}$ – the sets of all writes, reads, fences, branches, memory allocations, and memory reclamations, respectively. We define functions to extract information from events: *loc* to extract the location, *val* to extract the value, and *order* to extract the memory ordering. The

origin of a symbol, $O(\alpha)$, is the unique event e that introduces the symbol. Origin is raised to sets of symbols and, syntactically, to expressions.

4.1 Expression Interpretation

Our expressions are made up of symbols, values, and abstract unary and binary operators. We instantiate arithmetic and comparison operators ranging over integers, booleans, and pointer values. We provide the formal expression syntax in Appendix D. Function $\text{syms}(\varepsilon)$ extracts the set of symbols used in the expression ε.

The expression interpretation function, $[\![\varepsilon]\!]_f$, takes a map from symbols to expressions called an *environment*, f, and returns an expression. This acts as an eager partial evaluation for the expressions. We define semantic equality of expressions, $\equiv$, and predicate-restricted semantic equality, $\equiv_P$:

$$(\varepsilon_1 \equiv \varepsilon_2) \triangleq \forall f.\, [\![\varepsilon_1]\!]_f, [\![\varepsilon_2]\!]_f \in \mathbb{V} \implies [\![\varepsilon_1]\!]_f = [\![\varepsilon_2]\!]_f \qquad (\varepsilon_1 \equiv_P \varepsilon_2) \triangleq (P \implies \varepsilon_1 = \varepsilon_2) \equiv \top$$

Semantic equality allows us to declare that the expression $\alpha = 1 \wedge \bot$ is equal to $\bot$ regardless of environment, thus $(\ (\alpha = 1 \wedge \bot) \vee P\ ) \equiv (\bot \vee P) \equiv P$.

4.2 Building Symbolic Event Structures

For program P, $\langle P \rangle_{n\ \rho\ \kappa\ \varphi}$, is its semantic interpretation. Parameter n is a step counter used to ensure termination, ρ is an environment, mapping register symbols to expressions, κ is a function from environments to event structures, used as a continuation, and φ a predicate summarising the control flow leading to the current statement. $\kappa_\emptyset$ is a function from any environment to the empty event structure.

Sequencing. To interpret $p; P$, we first interpret p into an event, as described below, and then compose that with the event structure that results from interpreting P. [2] [3]

- r_i `:=` ε – updates the value of a register.
- r_i `:=`$_o$ x – loads a value from a global variable, generating a read event.
- x `:=`$_o$ ε – stores an expression into a global variable, generating a write event.
- r_i `:= &`x – loads the address of global variable x into r_i. This does not generate an event. The address of all global variables are assumed to be distinct.
- r_i `:=`$_o$ `*`ε – loads a value from a pointer, generating a read event.
- `*`ε_1 `:=`$_o$ ε_2 – stores an expression into a pointer, generating a write event.
- r_1 `:= fadd`$_{o_r,o_w}(l, \varepsilon)$ – performs an atomic fetch and add, generating a read event and write event.
- r_1 `:= cas`$_{o_r,o_w}(l, \varepsilon_1, \varepsilon_2)$ – performs an atomic compare exchange, generating a read, a branch, and a write.
- `fence`$_o$ – inserts a memory fence, generating a fence event.
- r_i `:= malloc`(ε) – allocates a block of memory of size ε, generating an alloc event.
- `free`(r_i) – frees a block of memory, generating a free event.

Prefixing, $\bullet$, composes an event e with an event structure.

Definition 4.3. (Event structure prefix). $e[\varphi] \bullet (\mathbb{E}, \sqsubseteq, \#, \sqsubseteq^{\text{rmw}}, \curlyvee) \triangleq (\mathbb{E}', \sqsubseteq', \#, \sqsubseteq^{\text{rmw}}, \curlyvee')$ where,

$$\mathbb{E}' = \{e\} \cup \mathbb{E} \qquad \sqsubseteq' = \sqsubseteq \cup (\{e\} \times \mathbb{E}) \qquad \curlyvee' = \curlyvee[e \mapsto \varphi]$$

[2] `cas` is the strong variant, with success and failure orderings being equal, an extension to asymmetric ordering is presented in Appendix E.

[3] Any C++ *fetch_key* function can be constructed in a similar way to `fadd` (fetch_add).

We define the semantic interpretation of commands:

$$\begin{aligned}
\langle r_i := \varepsilon\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq \kappa(\rho[r_i \mapsto [\![\varepsilon]\!]_\rho]) \\
\langle r_i :=_o x\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : R_o\ x\ \alpha)[\varphi] \bullet \kappa(\rho[r_i \mapsto \alpha]) \\
\langle x :=_o \varepsilon\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : W_o\ x\ [\![\varepsilon]\!]_\rho)[\varphi] \bullet \kappa(\rho) \\
\langle r_i := \&x\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq \kappa(\rho[r_i \mapsto x]) \\
\langle r_i :=_o {}^*\varepsilon\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : R_o\ [\![\varepsilon]\!]_\rho\ \alpha)[\varphi] \bullet \kappa(\rho[r_i \mapsto \alpha]) \\
\langle {}^*\varepsilon_1 :=_o \varepsilon_2\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : W_o\ [\![\varepsilon_1]\!]_\rho\ [\![\varepsilon_2]\!]_\rho)[\varphi] \bullet \kappa(\rho) \\
\langle r_i := \texttt{fadd}_{o_r,o_w}(x,\varepsilon)\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq rmw(\ (e_1 : R_{o_r}\ x\ \alpha) \bullet (e_2 : W_{o_w}\ x\ (\alpha + [\![\varepsilon]\!]_\rho)) \bullet \\
&\qquad \kappa(\rho[r_i \mapsto \alpha]), e_1, e_2\ ) \\
\langle r_i := \texttt{cas}_{o_r,o_w}(x,\varepsilon_1,\varepsilon_2)\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq rmw(\ (e_1 : R_{o_r}\ x\ \alpha) \bullet (e_2 : [\alpha = [\![\varepsilon_1]\!]_\rho]) \bullet \\
&\qquad ((e_3 : W_{o_w}\ x\ \varepsilon_2) \bullet \kappa(\rho[r_i \mapsto 1])) + \kappa(\rho[r_i \mapsto 0]), e_1, e_3) \\
\langle \texttt{fence}_o\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : F_o) \bullet \kappa(\rho) \\
\langle r_i := \texttt{malloc}(\varepsilon)\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : Alloc\ \alpha\ [\![\varepsilon]\!]_\rho) \bullet \kappa(\rho[r_i \mapsto \alpha]) \\
\langle \texttt{free}(r_i)\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : Free\ [\![r_i]\!]_\rho) \bullet \kappa(\rho)
\end{aligned}$$

We extend $\sqsubseteq^{\mathsf{rmw}}$ in `cas` and `fadd`: $rmw((\mathbb{E}, \sqsubseteq, \#, \sqsubseteq^{\mathsf{rmw}}, \curlyvee), r, w) \triangleq (\mathbb{E}, \sqsubseteq, \#, \sqsubseteq^{\mathsf{rmw}} \cup \{(r,w)\}, \curlyvee)$.

Interpreting `if` (b) $\{P_1\}$ `else` $\{P_2\}$ entails interpreting each branch: P_1 in a context where b holds, and P_2 in a context where $\neg b$ holds. The guarantees provided by b are reflected in a generated branch event e with condition $[\![b]\!]_\rho$. `while` (b) $\{P\}$ is interpreted as iteration of `if`, decrementing the step counter each time.

$$\begin{aligned}
\langle \texttt{if}\ (b)\ \{P_1\}\ \texttt{else}\ \{P_2\}\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : [[\![b]\!]_\rho]) \bullet \left(\langle P_1\rangle_{n\ \rho\ \kappa\ (\varphi \wedge [\![b]\!]_\rho)} + \langle P_2\rangle_{n\ \rho\ \kappa\ (\varphi \wedge \neg [\![b]\!]_\rho)}\right) \\
\langle \texttt{while}\ (b)\ \{P\}\rangle_{n\ \rho\ \kappa\ \varphi} &\triangleq \langle \texttt{if}\ (b)\ \{P; \texttt{while}\ (b)\ \{P\}\}\ \texttt{else}\ \{\texttt{skip}\}\rangle_{n-1\ \rho\ \kappa\ \varphi}
\end{aligned}$$

Coproduct, $+$, places two structures in conflict, extending the $\#$ relation.

Definition 4.4. (Coproduct).

$$\begin{aligned}
&(\mathbb{E}_1, \sqsubseteq_1, \#_1, \sqsubseteq_1^{\mathsf{rmw}}, \curlyvee_1) + (\mathbb{E}_2, \sqsubseteq_2, \#_2, \sqsubseteq_2^{\mathsf{rmw}}, \curlyvee_2) \triangleq \\
&(\mathbb{E}_1 \cup \mathbb{E}_2, \sqsubseteq_1 \cup \sqsubseteq_2, \#_1 \cup \#_2 \cup (\mathbb{E}_1 \times \mathbb{E}_2), \sqsubseteq_1^{\mathsf{rmw}} \cup \sqsubseteq_2^{\mathsf{rmw}}, \curlyvee_1 \cup \curlyvee_2)
\end{aligned}$$

For *parallel composition*, written $P_1 \parallel P_2$, we interpret $\langle P_1\rangle_{n\ \rho\ \kappa\ \varphi}$ and $\langle P_2\rangle_{n\ \rho\ \kappa\ \varphi}$ separately and then combine the resulting structures with the $\times$ operator.

Definition 4.5. (Product).

$$(\mathbb{E}_1, \sqsubseteq_1, \#_1, \sqsubseteq_1^{\mathsf{rmw}}, \curlyvee_1) \times (\mathbb{E}_2, \sqsubseteq_2, \#_2, \sqsubseteq_2^{\mathsf{rmw}}, \curlyvee_2) \triangleq (\mathbb{E}_1 \cup \mathbb{E}_2, \sqsubseteq_1 \cup \sqsubseteq_2, \#_1 \cup \#_2, \sqsubseteq_1^{\mathsf{rmw}} \cup \sqsubseteq_2^{\mathsf{rmw}}, \curlyvee_1 \cup \curlyvee_2)$$

4.3 Justification

The justification of a write w, $(P, D) \vdash^\delta w$, captures the circumstances and requirements for the write to occur. We say a write w is *independent* if $(\top, \emptyset) \vdash^\delta w$. And for a justification, j, we define: j_P, j_D, j_δ, and j_w to extract the respective components. From an event structure with write set $\mathcal{W}$, we derive a set of *initial justifications*, $\mathbb{J}_0$, and perform *elaborations*, $\square$, to generate the set of all justifications, $\mathbb{J}$[4].

[4]Under a finite value set, elaborations perform a search through a finite space in an indeterminate but finite number of steps, so the calculation is decidable (see Appendix G).

$$\mathbb{J}_0 = \{(\curlyvee(w), \mathrm{syms}(loc(w)) \cup \mathrm{syms}(val(w))) \vdash^{(\emptyset,\emptyset)} w \mid \forall w \in \mathcal{W}.\ \curlyvee(w) \not\equiv \bot\}$$

$$\mathbb{J}_{i+1} = \mathbb{J}_i \cup \{j \mid \exists j_1, j_2 \in \mathbb{J}_i.\ j_P \not\equiv \bot \wedge (j_1, j_2 \quad \square \quad j)\} \qquad \mathbb{J} = \bigcup_{i=0}^{\infty} \mathbb{J}_i$$

Definition 4.6. (Elaboration). Six predicates capture the guards and transformations of the possible elaborations, $G_{\mathrm{va}}, G_{\mathrm{str}}, G_{\mathrm{fwd}}, G_{\mathrm{we}}, G_{\mathrm{lift}}, G_{\mathrm{weak}}$ and are defined in §4.7.

$$j_1, j_2 \quad \square \quad j \triangleq \exists\, G \in \mathbb{G}.\ G(j_1, j_2, j) \qquad \mathbb{G} \triangleq \{G_{\mathrm{va}}, G_{\mathrm{str}}, G_{\mathrm{fwd}}, G_{\mathrm{we}}, G_{\mathrm{lift}}, G_{\mathrm{weak}}\}$$

We define $X_{\backslash S}$ as the restriction of X to all $(a, b) \in X$ such that both a and b are not in S. $A; B$ is the left composition of relations binding tighter than $\cup$.

4.4 Preserved Program Order

Definition 4.7. (Forwarding Context). A justification's forwarding context, $\delta = (f, we)$, captures the memory accesses that have been fused during elaboration (§4.13). ψ_δ are the constraints imposed by the forwarding context and $remap_\delta(e)$ returns the event that δ fuses e into.

$$\psi_{(f,we)} \triangleq \bigwedge_{(e_1,e_2)\in f} val(e_1) = val(e_2)$$

$$remap_{(f,we)}(e) \triangleq \begin{cases} remap_{(f,we)}(e_1) & \textbf{where } (e_1, e) \in (f \cup we) \\ e & \textbf{otherwise} \end{cases}$$

Preserved program order, $\leq$, captures ordering constraints and is composed of $\leq_{sync}$, $\leq_{rmw}$, and $\leq_{alias}$. $\leq_{sync}$ captures ordering imposed by synchronisation, $\leq_{rmw}$ extends synchronisation through atomic read-modify-write events, and $\leq_{alias}$ captures ordering imposed by potential aliasing.

$\leq^P_\delta$ remaps preserved program order under the forwarding context δ and predicate P. $pred_\delta(e, P)$ produces the set of immediate predecessors of e in the remapped $\leq$ relation.

Definition 4.8. (Preserved Program Order).

$$\leq_{sync} \triangleq (\sqsubseteq; \mathcal{W}|^?_{rel/sc} \cup \mathcal{R}|^?_{acq/sc}; \sqsubseteq \cup \sqsubseteq; \mathcal{F}|^?_{sc}; \sqsubseteq \cup \sqsubseteq; \mathcal{F}|^?_{rel}; \sqsubseteq_{\backslash \mathcal{R}} \cup \sqsubseteq_{\backslash \mathcal{W}}; \mathcal{F}|^?_{acq}; \sqsubseteq)_{\backslash \mathcal{F} \cup \mathcal{B}}$$

$$\leq_{rmw} \triangleq \leq_{sync}; \sqsubseteq^{\mathrm{rmw}-1} \cup \sqsubseteq^{\mathrm{rmw}-1}; \leq_{sync}$$

$$\leq^P_{alias} \triangleq \{(e_1, e_2) \in \sqsubseteq \mid \exists f.\ ([\![P \wedge loc(e_1) = loc(e_2)]\!]_f) \equiv \top\}$$

$$\leq^P_\delta \triangleq remap_\delta(\leq_{sync} \cup \leq_{rmw} \cup \leq^P_{alias}) \setminus \mathbb{E}^?$$

$$pred_\delta(e, P) \triangleq \{e' \mid e' \leq e \wedge \nexists e''.\, e' \leq e'' \leq e\} \quad \text{where } \leq = \leq^{P \wedge \psi_\delta}_\delta$$

4.5 Freezing

Freezing converts justifications into dp edges: for justification $(P, D) \vdash^\delta w$, we take the origin of symbols in P and D, and establish $e \xrightarrow{\mathrm{dp}} w$ for all events, e, in the derived origin set. In addition, freezing applies the forwarding context to preserved program order, and builds a constraint on rf that ensures: events used in rf and dp are not elided; rf covers the reads in E; and both the branching choices and the value and location equality implied by rf are satisfiable.

Definition 4.9. (Freeze). For justification set J, $freeze(J) \triangleq (E, \mathrm{dp}, \leq, \lambda \mathrm{rf}.\ \varphi, J)$, where:

- E is a maximal conflict free set of events.
- All justifications $j \in J$ share the forwarding context δ, and † are the events elided by δ.

- All writes in $E \setminus \dagger$ are uniquely justified.
- φ_{rf} enforces equivalence of the location and value of each write-read pair in rf.
- enforces disjointness of symbolic pointers introduced by allocation events if a free cannot intervene (Appendix I).

$$\mathsf{dp} \triangleq \bigcup_{j \in J} (O(j_P) \cup O(j_D)) \times \{w\} \qquad P \triangleq \bigwedge_{j \in J} j_P \wedge \psi_\delta \qquad \leq \triangleq \bigcup_{j \in J} \leq_\delta^{j_P} \cap \{e \in E \mid e \sqsubseteq^? j_w\}^2$$
$$\varphi \triangleq \dagger \cap \pi_1(\mathsf{rf} \cup \mathsf{dp}) = \emptyset \ \wedge \ E|_{\mathcal{R}} = \pi_2(\mathsf{rf}) \ \wedge \ \exists g. [\![P \wedge \varphi_{\mathsf{rf}}]\!]_g = \top$$

4.6 Program Semantics

We define the semantics of program P ,$[\![P]\!]_n$, under step index n in a series of steps. An event structure, $\langle P \rangle_{n\ \rho\ \kappa_\emptyset\ \top} = (\mathbb{E}, \sqsubseteq, \#, \sqsubseteq^{\mathsf{rmw}}, \curlyvee)$, and set of elaborated justifications, $\mathbb{J}$, are derived from P.

A set of *candidate executions*, $\mathcal{X}$, are derived from the event structure. Each execution is a tuple $(\mathbb{E}_X, \mathsf{rf}_X, \mathsf{mo}_X, \sqsubseteq_X^{\mathsf{rmw}}, \mathsf{dp}_X, \leq_X, J_X)$ where:

- $\mathbb{E}_X$ is a maximal conflict-free subset of $\mathbb{E}$.
- rf_X is a *reads-from* relation, pairing writes with the reads that take their value.
- mo_X, *modification order*, is a per-location total order over writes.
- $\sqsubseteq_X^{\mathsf{rmw}}$ links the load and store of successful compare-and-swap and fetch-and-add operations.
- dp_X is the *semantic dependency relation*, to be subsequently checked.
- $\leq_X$ is the *preserved program order* relation (4.8), again to be subsequently checked.
- J_X is the *justifying set*, the justifications used to justify this execution.

We filter candidate executions to a set of *complete executions*, where for each X: there is a $J_X \subseteq \mathbb{J}$ such that $\mathit{freeze}(J_X) = (\mathbb{E}_X, \mathsf{dp}_X, \leq_X, \varphi, J_X)$ and $\varphi(\mathsf{rf}_X)$ holds. The semantics of the program, $[\![P]\!]_n$, is the subset of the complete executions that are *coherent* according to the RC11 axiomatic constraints, excluding NO-THIN-AIR, and imposing that $\mathsf{dp} \cup \leq \cup\, \mathsf{rf}$ is acyclic.

There is a natural separation between the calculation of dp and the application of the axiomatic memory model, here RC11; another axiomatic model might be used in its place. In extending the language to dynamic memory we introduce a new class of undefined behaviour where programming errors lead to *use after free* or *use of uninitialised memory*. Candidate definitions for these errors are presented in Appendix H.

4.7 Elaborations

Value assignment. Value assignment allows us to remove dependencies for a write whenever we can infer the value of a symbolic variable. For example, if $(w : W\ x\ \alpha)$ appears under the branch $[\alpha = 0]$, then we can swap the label of w to be $W\ x\ 0$ and remove the data dependency on α.

Definition 4.10. (Value assignment).

$$\begin{aligned} G_{\mathrm{va}}(j_1, j_2, j) \triangleq \ & j_1 = j_2 \wedge j_1 = (P, D) \vdash^\delta (e : W_o\ x\ \varepsilon) \ \wedge \ j = (P, D') \vdash^\delta (e : W_o\ x'\ \varepsilon') \\ & \wedge \alpha \equiv_P v \wedge v \in \mathbb{V} \\ & \wedge x' = [\![x]\!]_{[\alpha \mapsto v]} \wedge \varepsilon' = [\![\varepsilon]\!]_{[\alpha \mapsto v]} \wedge D' = \mathrm{syms}(x') \cup \mathrm{syms}(\varepsilon') \end{aligned}$$

Strengthening. A control dependency, P, can be strengthened by an arbitrary predicate, P'. We restrict P' to predicates where all symbols are visible to the justified write.

Definition 4.11. (Strengthening).

$$G_{\text{str}}(j_1, j_2, j) \triangleq\ j_1 = j_2 \wedge j_1 = (P, D) \vdash^{\delta} w\ \wedge\ j = (P', D) \vdash^{\delta} w \\ \wedge S = O(P') \setminus O(P) \wedge remap_{\delta}(S) = S \wedge \forall e \in S.\ (e \sqsubseteq w \vee w \sqsubseteq e) \wedge w \not\leq^{P}_{\delta} e \\ \wedge P' \Rightarrow (P \wedge \bigwedge_{e \in S} \curlyvee(e))$$

Forwarding. The forwarding operation accommodates compiler optimisations that fuse same-location memory accesses. For justification j, forwarding, $\xrightarrow{\text{F}_j}$, and write elision, $\xrightarrow{\text{WE}_j}$, relate memory accesses that can be fused under the constraints imposed by j.

Definition 4.12. (Forwarding Relation).

$$e_1 \xrightarrow{\text{F}'_j} e_2 \triangleq\ j = (P, D) \vdash^{\delta} w\ \wedge e_1 \in pred_{\delta}(e_2, P) \wedge loc(e_1) \equiv_{P \wedge \psi_{\delta}} loc(e_2)$$

$$e_1 \xrightarrow{\text{F}_j} e_2 \triangleq\ e_1 \xrightarrow{\text{F}'_j} e_2 \wedge (e_1, e_2) \in (\mathcal{W} \times \mathcal{R}|_{rlx} \cup \mathcal{W} \times \mathcal{W}|_{rlx} \cup \mathcal{R} \times \mathcal{R})$$

$$e_1 \xrightarrow{\text{WE}_j} e_2 \triangleq\ e_1 \xrightarrow{\text{F}'_j} e_2 \wedge (e_1, e_2) \in (\mathcal{W} \times \mathcal{W})$$

For justification j_1, applying a forwarding, $e_1 \xrightarrow{\text{F}_{j_1}} e_2$, updates the justifying set and justifying predicate, removing the dependencies of the forwarded event and adding those of the forwarding event. A write elision simply records that an event is shadowed in the forwarding context.

Definition 4.13. (Forwarding).

$$G_{\text{fwd}}(j_1, j_2, j) \triangleq\ j_1 = j_2 \wedge j_1 = (P, D) \vdash^{(f, we)} (e\colon W_o\ x\ \varepsilon)\ \wedge\ j = (P, D') \vdash^{(f \cup (e_1, e_2), we)} (e\colon W_o\ x'\ \varepsilon') \\ \wedge e_1 \xrightarrow{\text{F}_{j_1}} e_2 \wedge env = [val(e_2) \mapsto val(e_1)] \\ \wedge P' = [\![P]\!]_{env} \wedge \varepsilon' = [\![\varepsilon]\!]_{env} \wedge x' = [\![x]\!]_{env} \wedge D' = \text{syms}(\varepsilon') \cup \text{syms}(x')$$

$$G_{\text{we}}(j_1, j_2, j) \triangleq\ j_1 = j_2 \wedge j_1 = (P, D) \vdash^{(f, we)} w\ \wedge\ j = (P, D) \vdash^{(f, we \cup (e_2, e_1))} w\ \wedge e_1 \xrightarrow{\text{WE}_{j_1}} e_2$$

Lifting. Thus far, the rules of the model have only introduced dependencies, e.g. at conditional branches, but if we have equivalent writes in each branch, then the write will happen regardless of the condition, and there should be no dependency. *Lifting* is the crux of the model: it is the mechanism that recognises when dependencies are false and removes them. In lifting, we will need to establish equivalence of expressions in conflicting branches.

A *relabelling*, Λ, is a environment that maps symbols, one-to-one, from one branch into symbols from a conflicting branch.

Definition 4.14. (Relabel-Equivalence). Two expression-predicate pairs, ε_1 under P_1 and ε_2 under P_2, are *relabel equivalent* with relabelling Λ, and context δ, written $P_1 : \varepsilon_1 \xrightarrow{\Lambda, \delta} P_2 : \varepsilon_2$, if:

$$\exists e.\ ([\![P_1 \Rightarrow \varepsilon_1 = e]\!]_{\Lambda} \wedge (P_2 \Rightarrow \varepsilon_2 = e) \equiv_{\psi_{\delta}} \top)$$

Raising to events, e_1 under P_1 and e_2 under P_2 are relabel equivalent, written $P_1 : e_1 \xrightarrow{\Lambda,\delta} P_2 : e_2$, if the expressions and locations of the events are relabel equivalent:

$$\begin{cases} P_1 : loc(e_1) \xrightarrow{\Lambda,\delta} P_2 : loc(e_2) \wedge P_1 : val(e_1) \xrightarrow{\Lambda,\delta} P_2 : val(e_2) & \textbf{where } (e_1, e_2) \in (\mathcal{R}^2 \cup \mathcal{W}^2 \cup \mathcal{A}^2) \\ P_1 : loc(e_1) \xrightarrow{\Lambda,\delta} P_2 : loc(e_2) & \textbf{where } e_1, e_2 \in C \\ \top & \textbf{where } e_1, e_2 \in \mathcal{F} \\ \bot & \textbf{otherwise} \end{cases}$$

Definition 4.15. (Closed Relabel-Equivalence). We close relabel equivalence under preserved program order to define *closed relabel equivalence*:

$$\begin{aligned} P_1 : e_1 \xrightarrow{\Lambda,\delta}{}^{*} P_2 : e_2 \triangleq\ & P_1 : e_1 \xrightarrow{\Lambda,\delta} P_2 : e : 2 \\ & \wedge\ \emptyset = pred_\delta(e_1, P_1) \Leftrightarrow \emptyset = pred_\delta(e_2, P_2) \\ & \wedge\ \forall e_1' \in pred_\delta(e_1, P_1), e_2' \in pred_\delta(e_2, P_2).\ P_1 : e_1' \xrightarrow{\Lambda,\delta}{}^{*} P_2 : e_2' \end{aligned}$$

Definition 4.16. (Lifting). Lifting adds a new justification with the disjunction of control dependencies from invariant stores on conflicting branches. It must check that there is a relabelling matching up locations and values in each branch over the write, the data dependencies, and their $\leq$-predecessors.

$$\begin{aligned} G_{\text{lift}}(j_1, j_2, j) \triangleq\ & j_1 = (P_1, D_1) \vdash^\delta w_1 \ \wedge\ j_2 = (P_2, D_2) \vdash^\delta w_2 \ \wedge\ j = (\llbracket P_1 \rrbracket_\Lambda \vee P_2, D_2) \vdash^\delta w_2 \\ & \wedge\ P_1 : w_1 \xrightarrow{\Lambda,\delta}{}^{*} P_2 : w_2 \wedge \llbracket D_1 \rrbracket_\Lambda = D_2 \\ & \wedge\ \forall s \in D_1.\ P_1 : O(s) \xrightarrow{\Lambda,\delta}{}^{*} P_2 : O(\llbracket s \rrbracket_\Lambda) \end{aligned}$$

Optimisations may rely on *extrinsic* value constraints that are described at the level of the whole program. To account for this we introduce *weakening*, and extend the semantic judgement to accept an extrinsic *program-wide guarantee* Ω: $\llbracket P \rrbracket_{n\,\Omega}$.

Definition 4.17. (Weakening).

$$G_{\text{weak}}(j_1, j_2, j) \triangleq\ j_1 = j_2 \wedge j_1 = (P' \wedge P, D) \vdash^\delta w \ \wedge\ j = (P', D) \vdash^\delta w \ \wedge \Omega \Rightarrow P$$

4.8 Derived Program-Wide Guarantees

The semantics above is sufficient to describe the behaviour of an optimising compiler with any program-wide invariant, but it must be specified as an extrinsic constraint. Some whole-program invariants might be *derived* by considering the behaviours of the program – for example an implicit constraint on the value of a variable. We provide an extended variant of the semantics that closes over consistent invariants.

Definition 4.18. (Extended semantics). For program-wide guarantee Ω_0, $(\!|P|\!)_{n\,\Omega_0}$ is the smallest set where:

(a) $\llbracket P \rrbracket_{n\,\Omega_0} \in (\!|P|\!)_{n\,\Omega_0}$
(b) Given $\llbracket P \rrbracket_{n\,\Omega} \in (\!|P|\!)_{n\,\Omega_0}$, if
  (a) Ω' holds over all executions in $\llbracket P \rrbracket_{n\,\Omega}$, and
  (b) $\Omega \wedge \Omega'$ holds over all executions in $\llbracket P \rrbracket_{n\,(\Omega \wedge \Omega')}$,
  then $\llbracket P \rrbracket_{n\,(\Omega \wedge \Omega')} \in (\!|P|\!)_{n\,\Omega_0}$.

The corpus of litmus tests (§3) relies primarily on extrinsic program-wide guarantees and simple value-range constraints that can also be expressed extrinsically. The extended semantics is bounded

above by the behaviour defined by ISO C, which is equivalent to removing all dp edges by setting a false program-wide guarantee: $[\![P]\!]_{n\ \bot}$. Programmers have an expectation that optimisations will be reasonable, and we believe this will require further restriction of derived guarantees, and an extension of the corpus of litmus tests to explore this boundary.

5 Meta-Theoretical Results

Established C/C++ compiler mappings correctly implement Symbolic MRD. We refer to the RC11 model [21], $[\![P]\!]_{\text{RC11}}$. Prior work shows that the established mappings, from RC11 memory accesses to target hardware accesses, are sound [30]: $[\![P]\!]_{\text{RC11}} \supseteq [\![\text{comp}(P)]\!]_{\text{IMM}}$.

Lemma 5.1 (Symbolic MRD is implementable over major hardware targets).
$[\![P]\!]_{n\ \top} \supseteq [\![P]\!]_{RC11} \supseteq [\![comp(P)]\!]_{IMM}$

Proof sketch. For a given execution in $[\![P]\!]_{\text{RC11}}$, we must find a matching execution of $[\![P]\!]_{n\ \top}$. We observe that we can always construct a dp such that $(\text{dp} \cup \leq) \subseteq \sqsubseteq$ by freezing the initial justification of each write, so $\text{acyclic}(\sqsubseteq \cup \text{rf}) \implies \text{acyclic}(\text{dp} \cup \leq \cup \text{rf})$, as required. □

Symbolic MRD provides DRF-SC. The DRF-SC property guarantees sequentially consistent (SC) behaviour to programs that do not exhibit a race under SC execution [21]. We denote the set of all racy executions as *DR*. A sketch of the structure of a proof of DRF-SC is now provided.

$$([\![P]\!]_{\text{SC}} \cap DR = \emptyset) \Rightarrow ([\![P]\!]_{\text{SC}} = [\![P]\!]_{n\ \top})$$

We follow the structure of the proof of DRF-SC for RC11. We adopt the same characteristic of a racing pair in the relaxed memory model: same-location, at least one write, unordered by happens-before. The work of the proof it to find the *first* race in some order, to remove that race, to show that the now race-free fragment of the execution is SC, and then add back the race and complete a racy SC execution.

RC11 uses a linearisation of $\sqsubseteq \cup \text{rf}$ to order the events of an execution, but in Symbolic MRD, this can be cyclic. We take a linearisation of $\text{dp} \cup \leq \cup \text{rf}$ as our order and find the first race in this. Races are over a pair of events, and we remove the second event in the linearisation to remove the race. Without any races, the prefix behaves according to the SC axioms, following the RC11 proof. Now, we must add back the event and establish that there was a race. If this event was not part of a cycle in $\sqsubseteq \cup \text{rf}$, then this is a valid RC11 prefix, and we follow the steps of the original proof. If it was part of a $\sqsubseteq \cup \text{rf}$ cycle, then the event to be added must be a write, w, and the $\sqsubseteq$ preceding event in the cycle must be a read, r that is not ordered by $\text{dp} \cup \leq$. We must find a racy write, w' that can be executed as the next step of an SC execution.

The absence of a dependency from r to w is used to derive the write w'. Consider the justification of w. If there is an initial justification that is independent of r, then the write can be executed under SC. If w has a justification resulting from a sequence of elaborations, then there is a lifting step that removes r from its dependencies, and this step requires the write to be invariant under read r, so there is a label-similar event that races for every value read, and can be executed under SC. We add this event to the prefix and complete under SC execution to produce our racy SC execution.

6 Related Work

Symbolic MRD is not the first thin-air free memory model, but it is first to correctly permit the optimisation demonstrated in Example 1.1a, without granting it undefined behaviour. Symbolic MRD builds on prior work by Paviotti et al., where values were expressed concretely, inflating the representation of program behaviour. More recent work by Jeffrey et al. describes a denotational model with strong compositionality properties. Their model uses predicates and predicate transformers to keep track of the conditions required to observe a given event. When those conditions

are a tautology the dp is removed. This is analogous to our independent writes (4.3). These models also project a dp relation that can be used in the C/C++ concurrency model. The Promising Semantics [19] uses an operational model of promises and certification to rule out most thin-air values [17].

MRD and Symbolic MRD are not the first works to use event structures to encode the overlaid divergent executions of programs in a thin-air free memory model. Pichon-Pharabod and Sewell, and separately Chakraborty and Vafeiadis use event structures in their work as the state in an operational semantics. Jeffrey and Riely used event structures as the state in a game-semantic exploration to find the writes which are always reachable.

Some memory models forbid thin-air behaviour by ruling out common compiler optimisations. There are several tools for such models, and these have been very important in the development, evaluation, and use of language and hardware models. CPPMEM allowed academics to present work on the C/C++ memory model to industrial practitioners [7]. RC11 [21] and hardware models like ARM [2], can be evaluated by the Herd suite of tools. Herd has been instrumental in testing axiomatic models and automatically generating litmus tests to validate against implementations [3].

Tools for thin-air free models have relied on specialised solver techniques to make the problem of recognising false dependencies tractable [10]. Later work used brute-force solving in OCaml [28]. Z3 has enhanced performance of models with small proof burdens in their semantics [17]. Tools for the full promising semantics look difficult to implement; results by Lee et al. show that only the release-acquire fragment of Promising 2.0 [1] is decidable.

Work on C dynamic allocation semantics has yielded rich tooling support as well. The Cerberus and Cerberus-BMC projects allow programmers to explore their programs' pointer semantics, in particular to uncover some unexpected subtleties of undefined behaviour [22, 27]. This work has been extended to support and explore choices of provenance models like VIP and PNVI [26].

There has been discussion about whether processor vendors should restrict concurrent memory accesses to forbid load-store reordering. If the compiler were similarly restricted, this could rule out thin-air values without the need to calculate a dp relation, and it would make the machinery of Symbolic MRD unnecessary. A statement from ARM stresses the performance cost of this approach and rules it out [15].

7 Proposed C++ Standard Changes

The development of sMRD has been supported by an extensive conversation at Workgroup 21 (WG21, C++) of the ISO. Much of the discussion is recorded in white papers. Notable papers include N4136 [5], an early record of discussion on the thin-air problem; N4323 [18], where the notion of semantic dependency was first introduced – without a definition; P3064R2 [34], listing the desiderata for semantic dependency as a set of litmus tests, and P1780R2 [4], describing a change to the standard to include the semantic dependency relation.

P1780R2 proposes the minimal change to the standard to incorporate semantic dependency, and defers its explicit definition to an external technical report (a 'TR' in the ISO parlance). This approach allows the TR to point initially to P3064R2, constraining semantic dependency by example. Later, the TR can be replaced with a specification embodying Symbolic MRD, developed in conversation with the ISO. Finally, Symbolic MRD might be introduced into the standard. The initial change to the standard, from P1780R2, is presented in Appendix J.

A move to implementation-defined behaviour, and less of it. The ISO standard constrains the semantics of C++ with imperative language like "must", but it also features *non-normative* text that imposes no constraints, using language like "should" and text contained in *note* brackets. Some constraints

are left as *implementation defined*, where there is freedom in some choice in the semantics that may vary across compilers and their configurations.

The text that forbids thin-air cycles is entirely non-normative in the current standard [13, atomics.order]. Appendix J introduces semantic dependency and forbids cycles in $\mathsf{dp} \cup \mathsf{rf}$, but leaves dp implementation defined. If an implementation defines dp as empty, then this is equivalent to the existing standard (no cycles are constructed), but this choice makes it impossible to reason about code [6]. To enable reasoning, the compiler must make an implementation-defined choice to specify a non-empty dp, invoking a definition like sMRD. With this change alone, we have converted a hole in the standard made up of non-normative text, into an implementation-defined hole, albeit a large one. Even so, this may be a sensible stepping stone towards a thin-air fix.

If the standard adopted sMRD, then it would shrink the part of the specification that is implementation defined, and provide a more ergonomic interface to making compiler defined choices. A compiler could specify *extrinsic choices* like alignment of pointers (Example 1.3a), memory layout choices, or integer limits, instead of having to invoke the larger sMRD definition, or an equivalent.

The inductive definition of justification (§4.3) is monotonic in extrinsic choice: enforcing more extrinsic choices can never forbid behaviours, it only allows more. At the limit, the compiler can make an extrinsic choice, $[\![P]\!]_{n\ \bot}$, with behaviour matching the existing standard.

With sMRD and a suitable compiler-defined extrinsic choice, one can reason about programs that include $\sqsubseteq \cup \mathsf{rf}$ cycles, where in the existing C++ specification, one cannot [6].

Avoiding complexity. There is no question that sMRD is intricate, but there is a lot to keep track of: all of the components of justification are necessary to model the optimisations we see above real compilers and hardware. Regular programmers can sidestep all of this complexity with simple guidelines, e.g. avoiding the `relaxed` memory order when using C++ atomics. In such code, there can be no cycles in $\sqsubseteq \cup \mathsf{rf}$, so there are no cycles in $\mathsf{dp} \cup \mathsf{rf}$, and the calculation of dp can be ignored.

Mathematical specifications. We believe that specifications should be written in formal mathematics rather than prose, supported by tooling for developing expert intuition – like Symbolic MRDer. There is no precedent for mathematical specification in the C++ standard, but it is the use of prose that allowed the standard to harbour errors and ambiguities like the thin-air problem.

8 Conclusion

We set out new criteria for a memory model of an optimised concurrent systems language in §1.1. We now revisit those criteria. C1 (*memory reclamation*), we support non-trivial allocations, reclamation, and alias analysis. C2 (*support for optimisations*) we have demonstrated support for a wide range of global and sequential optimisations, including those that rely on undefined behaviour. C3 (*extrinsic guarantees*) along with undefined behaviour, our semantics is parameterised over an Ω which provides an interface for facts about program execution which come from outside the program syntax, to be leveraged while removing dependencies from executions. C4 (*tooling support*), our tool has validated our model over 173 tests automatically. Thanks to a combination of a decidable semantics, and optimisations derived from monotone properties of predicates, the tool is orders of magnitude faster than previous work. C5 (*justification*) dependencies are derived from a justification relation. For every justification that is used, there is a sequence of elaborations from a syntactic dependency to the semantic dependency realised in the execution.

The thin-air problem afflicts optimised concurrent languages that admit relaxed memory behaviour. The crux of it is modelling an envelope of allowable behaviour around compiler optimisation. This is not just a theoretical problem: there is a live discussion among industrial standards writers to find a definition of C++ that permits aggressive compiler optimisation but also precisely defines the semantics of a program [34]. sMRD captures information that compilers do use to

inform optimisation, but that previous models fail to leverage. The result is a dp relation that solves the thin-air problem, but also that admits more optimisations than prior work.

9 Data-Availability Statement

The artefacts associated with this publication, external to the paper are listed below, together with where they can be found.

(1) Symbolic MRDer is available online and will be submitted for artefact evaluation.
(2) The corpus of tests is available on the Symbolic MRDer website.
(3) The appendix, included as supplementary material, includes the full formalisation and proofs.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by the EPSRC under grants EP/X015076/1, EP/X021173/1, and EP/V000470/1.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, S. Krishna, and Viktor Vafeiadis. 2021. The Decidability of Verification under PS 2.0. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 1–29. doi:10.1007/978-3-030-72019-3_1

[2] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (jul 2021), 54 pages. doi:10.1145/3458926

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 40. doi:10.1145/2594291.2594347

[4] Mark Batty, Simon Cooksey, Scott Owens, Anouk Paradis, Marco Paviotti, and Daniel Wright. 2014. P1780R2: Modular Relaxed Dependencies– A new approach to the Out-Of-Thin-Air Problem. (2014). https://graymalk.in/iso-papers/p1780/p1780r2.html

[5] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon, and Peter Sewell. 2014. N4136: C Concurrency Challenges Draft. (2014). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4136.pdf

[6] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 283–307. doi:10.1007/978-3-662-46669-8_12

[7] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/1926385.1926394

[8] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. doi:10.1145/3290383

[9] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular data-race-freedom guarantees in the promising semantics. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 867–882. doi:10.1145/3453483.3454082

[10] Simon Cooksey, Sarah Harris, Mark Batty, Radu Grigore, and Mikolás Janota. 2019. PrideMM: Second Order Model Checking for Memory Consistency Models. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 12233)*, Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Creissac Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas (Eds.). Springer, 507–525. doi:10.1007/978-3-030-54997-8_31

[11] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2017. Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models. In *Static Analysis - 24th International Symposium, SAS 2017, New*

York, NY, USA, August 30 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10422), Francesco Ranzato (Ed.). Springer, 299–320. doi:10.1007/978-3-319-66706-5_15

[12] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[13] International Organization for Standardization. 2023. *Working Draft, Standard for Programming Language C++*. Technical Report. International Organization for Standardization. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf

[14] Matt Godbolt. 2012. Compiler explorer. https://godbolt.org

[15] Richard Grisenthwaite. 2023. *Views on Relaxed Atomics in C++ from Arm's technical leadership team.* https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-technical-view-on-relaxed-atomics

[16] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 759–767. doi:10.1145/2933575.2934536

[17] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. doi:10.1145/3498716

[18] Paul E. McKenney Alan Jeffrey and Ali Sezgin. 2014. N4323: Out-of-Thin-Air Execution is Vacuous. (2014). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4323.html

[19] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. doi:10.1145/3009837.3009850

[20] Ori Lahav. [n. d.]. *A case against semantic dependencies.* https://www.cs.tau.ac.il/~orilahav/papers/sdep-fowm24.pdf

[21] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. doi:10.1145/3062341.3062352

[22] Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 387–397. doi:10.1007/978-3-030-25540-4_22

[23] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. doi:10.1145/3385412.3386010

[24] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270. doi:10.1145/3297858.3304043

[25] Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. *Out-of-Thin-Air Execution is Vacuous.* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html

[26] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL (2019), 67:1–67:32. doi:10.1145/3290380

[27] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An operational semantics for C/C++11 concurrency. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, The Netherlands). ACM, New York, NY, USA, 18 pages. doi:10.1145/2983990.2983997

[28] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 599–625. doi:10.1007/978-3-030-44914-8_22

[29] Jean Pichon-Pharabod and Peter Sewell. 2016. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 622–633. doi:10.1145/2837614.2837616

[30] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. doi:10.1145/3290382

[31] William Pugh. 2004. *Java Causality Test Cases.* https://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html Accessed: 2018-11-17.

[32] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. doi:10.1145/1993498.1993520

[33] Jaroslav Ševčík. 2009. *Program transformations in weak memory models.* Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/3132

[34] Alan Stern, Paul E. McKenney, Michael Wong, and Maged Michael. 2024. P3064R2: How to Avoid OOTA Without Really Trying. (2024). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3064r2.pdf

[35] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 190–204. doi:10.1145/3009837.3009838