# AWAY FROM LINEAR MODELS OF CONCURRENT PROGRAMS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF PHD.

By

Daniel Wright

October 2022

# Abstract

Traditional approaches to imperative programming language semantics rely on first defining how each individual statement modifies the memory state, and then composing these definitions into a whole program via the interpretation of the sequential composition operator: the humble semicolon. The creation of the multiprocessor and advent of parallelism began to challenge this model. No longer was a program a single, linear sequence of statements, but it had statements which might occur in one order or another, or even simultaneously. To add to the complexity, compilers and hardware began to optimise their input programs, reordering and removing statements to improve runtime performance. The resulting stack of transformations and complications caused runtime executions to drift progressively further away from the program that a programmer believed they were writing.

Several approaches to this have appeared: process calculi which forbid processes from sharing memory and instead force them to communicate directly, maintaining sequential consistency, in which an execution must at least *appear* to be respecting the ordered sequence of statements model, and permitting weak memory ordering, in which an execution must maintain orders involving explicitly synchronised accesses but is free to reorder everything else. While weak memory is preferred by engineers building high-performance code, due to the relatively high cost of both passing messages and maintaining sequential consistency, the problem of creating a sound weak memory semantics for a real-world programming language with shared memory concurrency has yet to be fully solved.

Here we present a weakly ordered semantics for shared memory concurrency, given as an extension to a previously published model. We show that the existing model can be integrated into reasoning techniques which rely on an operational semantics, and that program transformations which cannot introduce new behaviours can be expressed as a relation over the objects of this semantics. We then add a layer of abstraction to the model which allows us to represent dynamic memory allocation in a weak memory context for the first time.

# Acknowledgements

My sincere thanks go to my supervisor Mark for letting me run off into the weeds and return with this, to my mother Oleander for raising a computer scientist, and to my sister Jade for being my rock for so many years.

# Contents

# Chapter 1

# Introduction

Computer programmers generally consider it a virtue to know what a program will do without needing to execute it first. This is a difficult virtue to attain, in part because computers are very complicated. Interacting with memory involves a surprisingly large number of specialised components designed to make memory operations as fast as possible - not only does the hardware make decisions about execution order, allowing instruction pipelining, determining when to flush cache lines, and even more complex optimisations such as branch predictions and speculative execution, but programs are almost always written using compilers and assemblers (or interpreters), not as raw machine code, and these tools can likewise make decisions about which parts of the program should run at which times, and which can be delayed or even optimised away entirely. To avoid requiring that all programmers understand every nuance of both the hardware and software involved in creating and running a computer program, we rely on abstractions.

Typically, a programmer will view their program as a linear sequence of statements which are executed in the order in which they are written. Programming languages usually describe themselves from this perspective, both those with mathematical descriptions of their behaviour like Haskell or ML family languages and those with natural language definitions such as C. Great efforts have been made by compiler and hardware engineers to preserve the soundness of this abstraction, even while implementing optimisations that alter execution order at runtime.

When trying to execute a program that accesses the same shared memory in multiple concurrent threads, however, this abstraction breaks down despite these efforts. These optimisations are now observable — we are now not only observing memory state when our own thread performs a read, but also whenever another thread could potentially perform a read. Swapping the order of two writes to unrelated locations is no longer undetectable. However, these optimisations are valuable. Disabling them risks counteracting the performance improvements that concurrency is intended to provide. An ideal solution would be to define exactly when any given statement in the program *might* run, so that programmers can still predict what their program will do even after these reorderings, but this is a complicated task. A single optimisation can be described easily in terms of the contexts in which is occurs and the change it makes to the program, but describing an entire chain of such optimisations simply by composing their individual semantics is undesirable. The description would only work for that chain — the same C program would have one "meaning" under GCC with one set of flags and another under Clang with a different set of flags.

We want *one* semantics for our language, but we want one that captures all of these optimisation chains, using all common components, in any order. Or, equivalently, we want a semantics that captures all of the ways we *can't* reorder the statements in our program. This problem is captured in the idea of *dependencies*. If one statement *depends* on the state observed by another, then they should never be reordered. If two statements are *independent*, then any optimisation should be free to reorder them. The goal of a weak memory model is to combine some idea of dependency with the set of explicit and implicit synchronisation operations provided by a particular language or architecture to describe what an input program may be allowed to do. This thesis discusses one such model, called the *Modular Relaxed Dependencies* model, and compares it to other models in the same space.

## 1.1 Structure

This thesis is divided into 6 chapters beyond the introduction. In chapter 2, we give a brief overview of why weak memory models are used and how four existing models, namely the Promising semantics, and the Java memory model, the WeakestMO model, and the "Bubbly" model, explain what an input program may or may not do. We also introduce event structures, which are used as the basis for the Modular Relaxed Dependencies model.

In chapter 3, we present the Modular Relaxed Dependencies, or MRD, model in its entirety, with a small change from the definitions given in the initial publication in Paviotti et al. (2020). We show by example how the behaviours permitted and forbidden by the MRD model differ from the models introduced in chapter 2, and how this approach is unique. I was a co-author on this publication, but joined late in the development of the model and predominantly worked on the refinement relation described in the next chapter.

In chapter 4, we introduce a relation over MRD structures which can be used to reason about the validity of program transformations which change the observable behaviour of a program but preserve its correctness, by taking in a set of significant locations as a parameter and allowing more transformations of accesses to the remaining insignificant locations. By parameterising the relation over a set of locations whose value is considered significant to correctness, we can discuss particular cases of optimisation in detail without relying on a global, application-insensitive definition of "correctness-preserving". This appears in the appendix of Paviotti et al. (2020), but not in the main body of the publication, and is entirely my own work.

We further investigate program correctness in a weak memory context in chapter 5, where we construct a program logic to create correctness proofs over weakly ordered concurrent programs, published in Wright, Batty and Dongol (2021). In the process, we construct an operational semantics which uses a partial order over program events to nondeterministically choose an execution order from a set of permissable orders, demonstrated with the semantic dependency relation generated by MRD. We then construct an indexing function which can guide a Hoare logic-style pre- and postcondition chain across this operational semantics. This presents a novel approach to the verification of weak memory programs by decoupling the traditionally syntax-driven method of program logics from the syntax-induced ordering of program statements. I constructed the proof that the operational semantics describes executions of the underlying model, while the semantics and program logic were produced in collaboration with Mark Batty and Brijesh Dongol. This logic has subsequently been formalised in Isabelle/HOL by Dalvandi et al. (2022).

In chapter 6, we introduce Symbolic MRD, a generalisation of MRD structures designed to allow more compact representations of larger input programs. This construction was made in part to increase the size and complexity of the litmus tests for which the model could give a legible output in a reasonable amount of computation time, but also to remove the restriction that all values read in a program come from a finite, pre-determined set. This paves the way for the work in chapter 7, which introduces addresses as values to symbolic MRD and opens the door to the inclusion of pointer manipulation and dynamic memory allocation in programs using shared memory concurrency. These chapters are entirely original contributions, and as yet unpublished.

# Chapter 2

# Background and Related Work

## 2.1 Semantics for Sequential Programs

The semantics of any language is generally comprised of a definition for each type of statement in the language, and a description of how these definitions interact. It allows one to model what the language is saying about the world outside of the language itself. Where natural languages tend to talk about the real world, and have sentences to describe objects, places, properties, and so on, programming languages prefer to talk about the computer. As a result, statements in a programming language generally describe either an expression whose value is to be computed or the storage or retrieval of a value from memory.

When defining the semantics of a programming language, its creators must balance the thoroughness of the definition against the ease with which any program written in the language may be reasoned about. We tend to want a large amount of abstraction for a semantics, especially from the details of the hardware. A compiler for a given language on a given target architecture is functionally a semantics, as it describes all the statements of the language in terms of assembly instructions, but it isn't a very useful one. It has too many complexities which don't influence the properties of a program that a programmer is likely to care about, it is too hard to read, and it only describes behaviour under one specific architecture. At the other end of the scale, a simple description of what each operation should do in a natural language such as English can also be a semantics. For instance, writing "The + operator takes two parameters and returns their sum" is an acceptable definition of "+", provided the reader understands concepts such as return values and summation. However, this sometimes doesn't give enough information: what happens if I were to try using letters in place of numbers? What if my arguments cause side effects when evaluated and I care about what order they execute in? The more details I include, the longer the sentence becomes, and the harder it is to read. Instead of either plain English, which is too imprecise, or machine instructions, which contain substantially more detail than is desirable, semantics for programming languages are often given in mathematics. When this mathematics is being written, a large number of design decisions, both conscious and unconscious, are made. Authors of a semantics must decide which parts of the execution of a statement are important to the meaning of a program and should be represented, and which parts are unimportant and should be left to the implementor's discretion.

If we were to build a toy programming language, we would likely want programs written in it to be *able* to do the following:

- Evaluate arithmetic expressions, as in `1 + 2`
- Compute truth values, as in `1 < 2`
- Store values in variables, as in `a := 1`
- Substitute the values in those variables during arithmetic expressions, as in `a + 2`
- Change their behaviour based on truth values, as in `if (a < 2) { b } else { c }`
- Repeat statements, as in `GOTO` or `while (a) { b }`

These means we want a mathematical description of what each of these types of statements does.

Describing the first two is easy; we can delegate this to the mathematics we're already using to describe everything else. We don't need to do any heavy lifting yet.

The next two points are trickier, since they need to refer to some piece of memory. We don't want to think about machine addresses or page tables or any of the other complexities of accessing memory in a computer, so we abstract them away into some fairly simple model of a bag of labelled boxes. To evaluate `a := 1`, fetch the box labelled "a" from memory and update the contents to be "1". To evaluate `a + 2`, fetch the box labelled "a" again and evaluate the expression after you replace all instances of "a" with its contents. The mechanical details of doing this are left to the engineers who implement our language. The mathematical details usually rely on tracking the current set of mappings from symbols to values, often in terms of a partial function which is updated every time a write is interpreted.

The final two points require us to start modelling, or at least referring to, the rest of the program text somehow. If we break the text into statements and number each of them sequentially, we can define actions like "evaluate the program starting at statement 3", or "evaluate everything between statement 4 and statement 10". We now assume that whenever we finish evaluating statement $n$, we maintain our memory state and start evaluating statement $n + 1$ unless the program tells us to do otherwise.

These decisions seem very intuitive to us. They might seem so intuitive that we don't notice them being made at all. We might even spend decades writing semantics which make decisions just like these, and then, after successfully engineering computers that can run multiple instructions at once, start adding ideas like threads to our semantics to make better use of this new hardware. That might not work.

## 2.2 Failure Modes of Sequential Semantics

While purely sequential styles of semantics are intuitive, the actual evaluation of a program is vastly more complex. There is a sequence of transformations which takes us from source code to hardware behaviour, and each of them can slightly alter the behaviour of its input. Even in relatively simple cases, first the source code must be compiled down to assembly and may be optimised by the compiler during this process. It can remove trips to memory via constant propagation, lift statements out of loop bodies or conditionals, and generally deviate from the assumed behaviour of the program as long as its output is consistent with the sequential semantics of the language. The assembly must then be assembled into machine instructions and executed by the hardware, which promises particular outcomes given particular inputs via the architecture specification but often provides weak guarantees about ordering. It may include cache hierarchies, branch predictors, or pipelines which reorder memory accesses. More portable languages such as Java may compile down to bytecode to be executed by a virtual machine, which will have its own set of optimisation steps.

In this section we give an overview of how these various components can alter the order in which program statements execute, and how concurrent threads can change their behaviour based on these reorderings.

### 2.2.1 Reordering Optimisations

The term *Sequential Consistency* or *SC*, articulated by Lamport (1979), applies to any specification for a multiprocessor in which the observed behaviour of said processor is indistinguishable from a purely sequential execution of the same program. Contemporary multiprocessors often violate this restriction in various ways for the sake of performance improvements.

### Store and Load Buffering

Suppose we have a program fragment that looks like this:

```
x := 1;
r1 := y;
```

When we run this program fragment, it turns out that accessing $y$ is very fast - in fact, it's in our L1 cache. Unfortunately, it's been too long since we last accessed anything near $x$ and as a result it isn't mapped to any of our cache space. The write to $x$ is going to take a very long time (in computer terms) to complete. We could sit here and twiddle our thumbs for a while, idling the entire machine until the write finishes, but this would be a very inefficient thing to do. We wouldn't see any difference, from the perspective of a single-threaded program, if we performed the load before we were done with the store. We refer to this as *store buffering*: we are allowed to perform other operations while we wait for a store to commit to memory, as long as the store wouldn't impact their outcome.

This is precisely what even the relatively restrictive x86 hardware is allowed to do in this situation. Specifically, according to the litmus tests provided by Intel, the following program is allowed to terminate with both the EAX and EBX registers containing 0 (Sewell et al. (2010)):

```
MOV [x]  ← 1    || MOV [y]  ← 1
MOV EAX  ← [y]  || MOV EBX  ← [x]
```

A simple operational semantics closely linked to program order cannot arrive at this outcome. There's no way of executing this while modelling memory as a bag of instantaneously available bits and program execution as only starting statement 2 with the environment it arrives at after executing statement 1 that arrives at this outcome.

Store buffering has a similar counterpart, known as *load buffering*. This time, instead of waiting for a store to finish updating memory, we would be waiting for a load to finish retrieving a value from memory.

A similar cache hierarchy can cause load buffering in this example:

```
r1 := y;
x := 1;
```

Suppose the location $y$ is rarely used and takes multiple cycles to access, so we want to avoid waiting for the value if we could be performing other operations in that time. We don't need to be sure that we have the correct value in `r1` before executing the store, so we may as well perform it before the load has returned.

This can likewise result in counterintuitive behaviours in a concurrent context, as in this program fragment where the outcome $\texttt{r1} = \texttt{r2} = 1$ is permitted, though not currently observed, on POWER multiprocessors (Sarkar et al. (2011a)) and both permitted and observed on ARMv7 (Alglave et al. (2011)):

```
LDR R0, [x] || LDR R0, [y]
MOV R1, #1  || MOV R1, #1
STR R1, [y] || STR R1, [x]
```

In the following, we can try to describe which conditions cause non-sequentially consistent optimisations to be allowed. We can start with our initial observations about reordering stores and loads:

*If a pair of memory accesses in a thread are unrelated, they may be executed in any order. If two accesses are to different variables, they are by default unrelated.*

**Branch Elimination**

We have mentioned that *unrelated* reads and writes may be reordered, accesses to the same variable are not the only type of related accesses. In the following case, it's clear that performing the read after the write is impossible:

```
r1 := x;
if (r1 = 1) {
 y := 1;
}
```

We can't know if the guard will evaluate to true or false if we don't have the correct contents for `r1`. This means we need to tighten our definition of "unrelated":

*If a load event is used by a branch statement, the load is related to any stores within the scope of the branch statement.*

As branch instructions are relatively slow, however, it's preferable for program optimisations to remove any unnecessary branch statements. This might happen in cases where the guard is always true:

```
r1 := x;
if (r1 - r1 = 0) {
 y := 1;
}
```

In this case, a compiler could determine that the check always passes and remove the branch entirely. This means we need to further tighten our definition of relatedness:

*If the value observed at a load operation cannot impact the truth of the guard condition, the load event and the writes below the guard are unrelated.*

There is another case where a load and store might be unrelated despite the syntactic presence of a branch condition, but this time due to the presence of another store:

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

In this case, while we do need the value of the load to determine the truth of the guard, we don't need the truth of the guard to determine whether or not to perform the store. Branch elimination optimisations are well-known (Bodík, Gupta and Soffa (1997), Mueller and Whalley (1995)) and widely implemented (Kotzmann et al. (2008), Calder, Grunwald and Zorn (1994), llv (2014)).

*If a store is within the scope of a branch statement but an identical store appears in the other branch, the store is unrelated to any loads required by the guard.*

### 2.2.2 Non-Multicopy Atomic Hardware

Multicopy atomicity is the guarantee that if one thread reads some value $v$ from a global variable, every other thread must also read $v$ if they access it between that read and any subsequent write. It promises that regardless of the complexity of our memory layout, which might sometimes have multiple copies of the same address holding different values at the same time, the programmer can still view memory as being a single, universally accessible object.

Hardware designed around multiple physical cores can violate this abstraction by having a small per-core cache that does not need to flush its contents to main memory between every operation. A program that one would expect to write to a global location can write to this local cache and perform several other operations before that write becomes visible. This means that if location $x$ contains the value 0 and thread $t_1$ performs a write of 1 to it, thread $t_1$ can observe that $x$ now contains 1 while thread $t_2$ observes that it still contains 0.

This type of behaviour is not permitted on x86 (Sewell et al. (2010)) or ARMv8 (Pulte et al. (2017))[1], but it is permitted by POWER (Mador-Haim et al. (2012)).

To give an example of non-MCA behaviour in action, consider the following program:

[1]Non-MCA behaviour was allowed in an initial version of the standard but later removed.

```
x := 0;
x := 1;      ||   r2 := y;
r1 := x;     ||   if (r2 = 2) {
y := r1;     ||    r3 := x;
y := 2;           }
```

At the end of this program, we are allowed to observe the state `r1 = 1, r2 = 2, r3 = 0`. This means that the final load into `r3` has observed the value from the first write to $x$, while the load into `r1` has observed the second one. We can tell that every line in the first thread is related to its neighbours, but the second thread is still free to observe them out of order.

*In non-MCA settings, relatedness is only visible within the thread containing the memory accesses.*

### 2.2.3 The Thin Air Problem

Gathering together our requirements from the previous section, we end up with the following:

- Memory accesses which are unrelated are reorderable.
- Memory accesses to the same variable within a thread are related.
- Loads whose value is used in a guard are related to any stores which only appear in either the true or false branch of the statement which uses the guard.
- In a non-MCA environment, accesses which are related in one thread may be observed to have been reordered by another thread.

If we want to allow the reordering of loads and stores within a thread, we encounter a problem whenever we try to interpret a load within a program. That is: we can't know, at the time of interpreting it, what the memory state might be. We can be sure that any related stores that come before it syntactically have happened, but we can't make assertions about what the other thread could have done so far. However, if a semantics is too permissive about what a load can observe, it allows so-called *thin air* values. It's clear that if, say, the value 42 is never written to the location $x$ during the execution of a program, the value 42 should never be observed in a load from $x$. It is likewise true that if the value 42 can only be stored *after* the load in question, it should similarly never be allowed to observe 42.

```
             x := 0;
             y := 0;
r1 := x;  ||   r2 := y;   // both read 42
y := r1;  ||   x := r2;   // both write 42
```

In the above example, a naïve semantics allows both threads to read 42 at their respective accesses and copy the values into global memory, and then asserts that, since 42 has been written to both global variables, the reads must be possible. Thread 1 can observe 42 at $x$ prior to thread 2 observing 42 at $y$ and vice versa. The key observation here should be that *both of these things can't simultaneously be true*:

$$\begin{tikzcd}
\text{Read 42 from } x \arrow[d, "\text{then}"'] & \text{Read 42 from } y \arrow[d, "\text{then}"] \\
\text{Write 42 to } y \arrow[ur, "\text{then}"'] & \text{Write 42 to } x \arrow[ul]
\end{tikzcd}$$

The role of a memory model therefore cannot be described as simply defining when operations may be reordered according to their context, but also to detect when a particular behaviour relies on these types of causality cycles.

## 2.3 Competing Semantics for Weak Memory Programs

The field of weak memory models, while relatively young, is somewhat crowded. Out-of-order accesses have the potential to cause undiagnosable bugs and security vulnerabilities, to violate type safety, and to invalidate formal correctness proofs which assume sequential consistency (Boehm and Demsky (2014)). They have also been incorrectly described in a publicly available standards documents for major languages (Pugh (1999), Ševčík and Aspinall (2008), Boehm and Demsky (2014), Vafeiadis et al. (2015), Lahav et al. (2017)).

In this section we offer a brief overview of a handful of existing models, with worked examples, and later we show how Modular Relaxed Dependencies handles the same programs.

### The Initialisation Program

Before diving into any particular memory model, it is helpful to mention a feature they often have in common: the inclusion of an initialisation program. Whenever a program is compiled or interpreted and a new variable is declared, one of three things may happen: the compiler or runtime may set the value to an acceptable default, it may refuse to compile if a value is not set before the variable is accessed, or the language standard may declare it "undefined behaviour" to access the variable prior to initialising it and cause the program to have no semantics at all.

As memory models intend to be as explicit as possible about the state of memory at all times, they produce unreliable outputs, or outright halt, if "invisible" writes such as variable initialisation are left out. The common solution is to have a section at the beginning of any program, sometimes called the initialisation program or $Init$, which sets the value of every global variable to 0.

### 2.3.1 Promising

The Promising semantics uses program order driven execution, but allows non-sequentially consistent interleavings by allowing a thread to observe writes which have yet to be interpreted. A thread can promise that at some point later in execution it will perform a write, given its current state would allow it to do so. Other threads can then read this promise as though it were a write that had actually been performed. It is a per-execution model, meaning it describes when an individual execution of a program is permitted.

This section, as with all subsequent descriptions of memory models, begins with a technical description of the semantics and proceeds into a series of examples demonstrating how these descriptions evaluate a program.

#### Definitions

Any write to a location in memory is represented as a *timestamped message* $\langle x, v@t \rangle$, where:

- $x$ is the location being written to
- $v$ is the value being written
- $t$ is the time at which the write occurred

The memory is not represented as a map from locations to values, but instead as a set of these messages which grows as the program executes.

The *thread state* is represented as a tuple $(\sigma, V, P)$, where:

- $\sigma$ is a function from thread-local variables to values.
- $V$ is a *view* function, which maps global variables to the timestamp of the last memory access observed by this thread.
- $P$ is a set of outstanding promises made by this thread.

Whenever a thread performs a load of $x$, it must load from a message which has a timestamp value greater than or equal to $V(x)$. Whenever a thread performs a store to $x$, it must create a message which has a timestamp value strictly greater than $V(x)$. Both of these operations update $V(x)$ with the new latest value.

$$\textsc{silent}\ \frac{\sigma \xrightarrow{silent} \sigma'}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V, P\rangle, M\rangle} \qquad \textsc{step}\ \frac{\langle\mathcal{TS}(i), M\rangle \to^{+} \langle\mathcal{TS}', M'\rangle \quad \langle\mathcal{TS}', M'\rangle \text{ is consistent}}{\langle\mathcal{TS}, M\rangle \to \langle\mathcal{TS}[i \mapsto \mathcal{TS}'], M'\rangle}$$

$$\textsc{read}\ \frac{\sigma \xrightarrow{R(x,v)} \sigma' \quad \langle x, v@t\rangle \in M \quad V(x) \le t \quad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P\rangle, M\rangle} \qquad \textsc{write}\ \frac{\sigma \xrightarrow{W(x,v)} \sigma' \quad M' = M \hookleftarrow \langle x, v@t\rangle \quad V(x) < t \quad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P\rangle, M'\rangle}$$

$$\textsc{promise}\ \frac{M' = M \hookleftarrow m \quad P' = P \hookleftarrow m}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma, V, P'\rangle, M'\rangle}$$

$$\textsc{fulfill}\ \frac{\sigma \xrightarrow{W(x,v)} \sigma' \quad \langle x, v@t\rangle \in P \quad P' = P \setminus \{\langle x, v@t\rangle\} \quad V(x) < t \quad V' = V[x \mapsto t]}{\langle\langle\sigma, V, P\rangle, M\rangle \to \langle\langle\sigma', V', P'\rangle, M\rangle}$$

Figure 1: The operational rules for the Promising semantics handling only reads and writes.

After performing any memory access or updating its local state, a thread must validate that each of the promises in $P$ are *consistent* with the new global memory state. A promise $P$ is consistent with memory state $M$ and thread states $\mathcal{TS}$ if executing the remainder of the program under these states could lead to the promise being fulfilled, meaning that a message is generated writing the promised value to the promised location at the promised time.

We give the formal operational rules for the Promising semantics in Fig. 1.

**Permitted Reordering by Example**

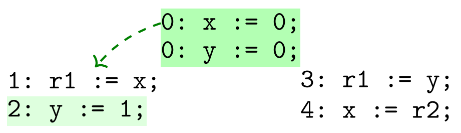
```
              0: x := 0;
              0: y := 0;
1: r1 := x;   ||   3: r2 := y;
2: y := 1;         4: x := r2;
```

Figure 2: Allowed: $\texttt{r1} = \texttt{r2} = 1$

The program above shows a permitted non-SC reordering potentially being observed by another thread. There is no interleaving of statements between the first and second thread which allows the outcome $\texttt{r1} = \texttt{r2} = 1$, but the reordering of line 2 above line 1 is permitted and results in this outcome. We now step through an execution of this program under the Promising semantics which does exactly this.

We begin by promising a message $\langle y, 1@1\rangle$ in thread 1 and verifying its consistency. Since the value written at line 2 must be 1 regardless of context, this is trivial. Here we show the steps taking by Promising so far, next to the program source. We highlight the point at which we need to fulfil a promise in light green, and link the proposed source of a message observed by a read using a dashed green arrow.

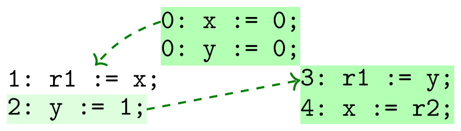
0: Write $\langle x, 0@0\rangle, \langle y, 0@0\rangle$
Promise: $\langle y, 1@1\rangle$
    1: Read $\langle x, 0@0\rangle$
    2: Write $\langle y, 1@1\rangle$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                 3: r1 := y;
2: y := 1;                  4: x := r2;
```

Thread 2 then copies the value of 1 into $x$, creating the message $\langle x, 1@1\rangle$. This doesn't touch any of the statements used in the initial verification of our promise, so we elide the re-verification step. We highlight program statements in darker green to indicate that they have executed.

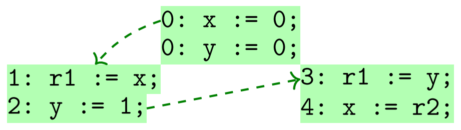
Promise: $\langle y, 1@1\rangle$
    1: Read $\langle x, 0@0\rangle$
    2: Write $\langle y, 1@1\rangle$
3: Read $\langle y, 1@1\rangle$
4: Write $\langle x, 1@1\rangle$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                 3: r1 := y;
2: y := 1;                  4: x := r2;
```

Thread 1 reads this value, and re-verifies that it can still fulfil its earlier promise. Finally, it creates the message $\langle y, 1@1\rangle$.

Promise: $\langle y, 1@1\rangle$
    1: Read $\langle x, 0@0\rangle$
    2: Write $\langle y, 1@1\rangle$
3: Read $\langle y, 1@1\rangle$
4: Write $\langle x, 1@1\rangle$
1: Read $\langle x, 1@1\rangle$
Verify: $\langle y, 1@1\rangle$
    2: Write $\langle y, 1@1\rangle$
2: Fulfil $\langle y, 1@1\rangle$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                 3: r1 := y;
2: y := 1;                  4: x := r2;
```

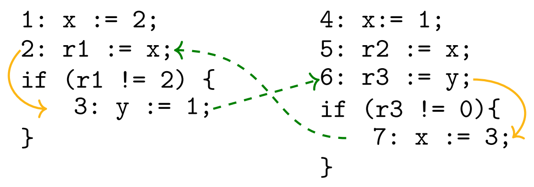
**Permitted OOTA Execution**

```
1: x := 2;              4: x := 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2){     ||    6: r3 := y;
  3: y := 1;            if (r3 != 0){
}                         7: x := 3;
                        }
```

Figure 3: The litmus test OOTA7, with the forbidden outcome $\texttt{r1} = 3 \wedge \texttt{r2} = 2 \wedge \texttt{r3} = 1$.

In this program, due to Jagadeesan, Jeffrey and Riely (2020), an execution which terminates with the forbidden outcome $\texttt{r1} = 3 \wedge \texttt{r2} = 2 \wedge \texttt{r3} = 1$ may be considered a thin-air execution. We take the description of the permitted execution from (Chakraborty and Vafeiadis (2019)):

- Line 2 reads from line 7
- The execution of line 3 depends on the value at line 2
- Line 6 reads from line 3
- The execution of line 7 depends on the value at line 6

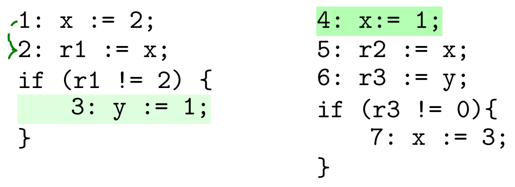


```
1: x := 2;              4: x:= 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2) {          6: r3 := y;
    3: y := 1;          if (r3 != 0){
}                           7: x := 3;
                        }
```

Drawing these constraints as arrows, we see the figure 8 shape in the code reflect the circular reasoning required to observe the forbidden behaviour. The green "reads from" arrows are relatively trivial to construct using any style of semantics, but the calculation of the orange "depends on" arrows is the role of a weak memory model.

We begin by executing the write to `x` on line 4 to obtain memory state $M_1 = \{\langle x, 1@2\rangle\}$. We can now promise that the write to `y` on line 3 will execute: line 1 could create message $\langle x, 2@1\rangle$, allowing line 2 to read the value 1. This gives us $M_2 = \{\langle x, 1@2\rangle, \langle y, 1@1\rangle\}$.

4: Write $\langle x, 1@2\rangle$
Promise: $\langle y, 1@1\rangle$
    1: Write $\langle x, 2@1\rangle$
    2: Read $\langle x, 2@1\rangle$
    3: Write $\langle y, 1@1\rangle$

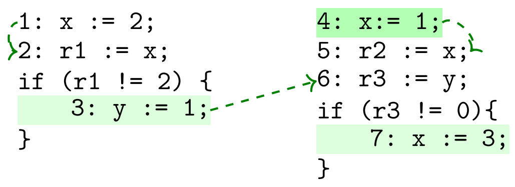


```
1: x := 2;              4: x:= 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2) {          6: r3 := y;
    3: y := 1;          if (r3 != 0){
}                           7: x := 3;
                        }
```

Figure 4: Execution up to memory state $M_2 = \{\langle x, 1@2\rangle, \langle y, 1@1\rangle\}$.

We can now promise that the write to `x` on line 7 will execute, as the read of `y` on line 6 can observe the new value of `y` in $M_2$. We choose timestamp $t = 4$, leading to $M_3 = \{\langle x, 1@2\rangle, \langle x, 3@4\rangle, \langle y, 1@1\rangle\}$.

4: Write $\langle x, 1@2\rangle$
Promise: $\langle y, 1@1\rangle$
    1: Write $\langle x, 2@1\rangle$
    2: Read $\langle x, 2@1\rangle$
    3: Write $\langle y, 1@1\rangle$
Promise: $\langle x, 3@4\rangle$
    5: Read $\langle x, 1@2\rangle$
    6: Read $\langle y, 1@1\rangle$
    3: Write $\langle x, 3@4\rangle$

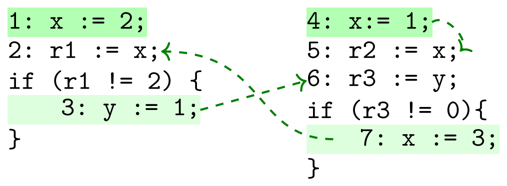


```
1: x := 2;              4: x:= 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2) {          6: r3 := y;
    3: y := 1;          if (r3 != 0){
}                           7: x := 3;
                        }
```

Figure 5: Execution up to memory state $M_3 = \{\langle x, 1@2\rangle, \langle x, 3@4\rangle, \langle y, 1@1\rangle\}$.

When we execute the write to `x` on line 1, we ignore the trace we used to verify the promises and instead generate message $\langle x, 2@3\rangle$. This allows the outcome `r2` $= 2$, but doesn't invalidate either promise: thread 2's promise to write to `y` is unaffected by the value stored in `r2`, and thread 1 can read the promised $\langle x, 3@4\rangle$ instead of $\langle x, 1@2\rangle$. We end with memory state $M_4 = \{\langle x, 1@2\rangle, \langle x, 2@3\rangle, \langle x, 3@4\rangle, \langle y, 1@1\rangle\}$.

- 4: Write $\langle x, 1@2\rangle$
- Promise: $\langle y, 1@1\rangle$
  - 1: Write $\langle x, 2@1\rangle$
  - 2: Read $\langle x, 1@2\rangle$
  - 3: Write $\langle y, 1@1\rangle$
- Promise: $\langle x, 3@4\rangle$
  - 5: Read $\langle x, 1@2\rangle$
  - 6: Read $\langle y, 1@1\rangle$
  - 7: Write $\langle x, 3@4\rangle$
- 1: Write $\langle x, 2@3\rangle$
- Verify: $\langle y, 1@1\rangle$
  - 2: Read $\langle x, 3@4\rangle$
  - 3: Write $\langle y, 1@1\rangle$

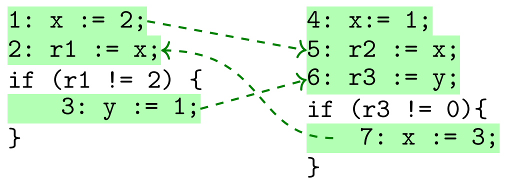


```
1: x := 2;               4: x:= 1;
2: r1 := x;              5: r2 := x;
if (r1 != 2) {           6: r3 := y;
    3: y := 1;           if (r3 != 0){
}                           7: x := 3;
                         }
```

Figure 6: Execution up to memory state $M_4 = \{\langle x, 1@2\rangle, \langle x, 2@3\rangle, \langle x, 3@4\rangle, \langle y, 1@1\rangle\}$.

The sequence of reads then executes as described by the desired outcome. Line 2 observes $\langle x, 3@4\rangle$, and line 5 observes $\langle x, 2@3\rangle$. Each thread finishes by fulfilling its promise

- 4: Write $\langle x, 1@2\rangle$
- Promise: $\langle y, 1@1\rangle$
  - 1: Write $\langle x, 2@1\rangle$
  - 2: Read $\langle x, 1@2\rangle$
  - 3: Write $\langle y, 1@1\rangle$
- Promise: $\langle x, 3@4\rangle$
  - 5: Read $\langle x, 1@2\rangle$
  - 6: Read $\langle y, 1@1\rangle$
  - 7: Write $\langle x, 3@4\rangle$
- 1: Write $\langle x, 2@3\rangle$
- Verify: $\langle y, 1@1\rangle$
  - 2: Read $\langle x, 3@4\rangle$
  - 3: Write $\langle y, 1@1\rangle$
- 2: Read $\langle x, 3@4\rangle$
- 3: Fulfil $\langle y, 1@1\rangle$
- 5: Read $\langle x, 2@3\rangle$
- 6: Read $\langle y, 1@1\rangle$
- 7: Fulfil $\langle x, 3@4\rangle$



```
1: x := 2;               4: x:= 1;
2: r1 := x;              5: r2 := x;
if (r1 != 2) {           6: r3 := y;
    3: y := 1;           if (r3 != 0){
}                           7: x := 3;
                         }
```

Figure 7: Completed execution, terminating with $r1 = 3 \land r2 = 2 \land r2 = 1$.

This execution is permitted because the lines of reasoning used to certify promises at each machine step are allowed to change. We are initially allowed to write `y := 1` under the condition that `x := 1` occurs after `x := 2`, and if we are allowed to write `y := 1` then we are allowed to write `x := 3`. When we proceed in our execution, however, we lose track of the requirement that `x := 1` happens last when we choose to read from `x := 3` instead. Note that these requirements are still transitive – `y := 1` no longer requires the orderings from before, instead it requires the execution of `x := 3`, but `x := 3` still requires `y := 1`. Since we have already assumed both of these events will happen, these requirements resolve each other and the cyclicity is never flagged as a problem.

**A Characteristic for the Promising OOTA Behaviour**

Suppose that each message in the Promising semantics is given an additional flag indicating whether or not it originates from a promise which has yet to be fulfilled. Promises are written (Promise $m$) while writes are written (Write $m$), while $Writes(M)$ gives the set of Write messages in memory $M$. We also add a precondition to our stored promises in the form of a message set:

$$\{\mu\}_{\langle \mathcal{TS},M,S\rangle}\ P\ \langle l, v@t\rangle \text{ iff } \langle \mathcal{TS}, Writes(M)\cup \mu, S\rangle \Longrightarrow \langle \mathcal{TS}', M', S'\rangle \text{ and } (\text{Write } \langle l, v@t\rangle) \in M'$$

Where the semantics of $\Longrightarrow$ are identical to $\rightarrow$ but without the promise and fulfil rules and $P$ is a unique identifier for the promise. Any promise we use during a speculative execution which generates the message promised by $P$ is noted in the precondition.

Whenever we take an operational step over program $S$, we take $S'$ to be the remaining program and re-validate all our promises, updating the stored objects $p$ from $\{\mu_1\}_{\langle \mathcal{TS}_1,M_1,S_1\rangle}\ p\ \langle l, v@t\rangle$ to $\{\mu_2\}_{\langle \mathcal{TS}_2,M_2,S_2\rangle}\ p\ \langle l, v@t\rangle$.

Finally, in order to validate promises we include the program text $S$ explicitly in the operational semantics.

$$\text{STEP'}\ \frac{\langle \mathcal{TS}(i), M, S\rangle \rightarrow^{+} \langle \mathcal{TS}', M', S'\rangle \quad \langle \mathcal{TS}', M', S'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M, S\rangle \rightarrow \langle \mathcal{TS}[i \mapsto \mathcal{TS}'], M', S'\rangle}$$

$$\text{PROMISE'}\ \frac{M' = M \hookleftarrow (\text{Promise } m) \quad P' = P \hookleftarrow (\mu, p_i, m) \quad \{\mu\}_{S,M}\ p_i\ m}{\langle\langle \sigma, V, P\rangle, M, S\rangle \rightarrow \langle\langle \sigma, V, P'\rangle, M', S\rangle}$$

$$\text{FULFILL'}\ \frac{\begin{array}{c}\sigma \xrightarrow{W(x,v)} \sigma' \quad (\mu, p_i, \langle x, v@t\rangle) \in P \quad P' = P \setminus \{(\mu, p_i, \langle x, v@t\rangle)\} \\ V(x) < t \quad V' = V[x \mapsto t] \quad M' = M \cup \{\text{Write } m\} \setminus \{\text{Promise } m\}\end{array}}{\langle\langle \sigma, V, P\rangle, M, (x := v; S)\rangle \rightarrow \langle\langle \sigma', V', P'\rangle, M', S\rangle}$$

Figure 8: The modified operational rules which track promise preconditions.

We can describe an OOTA execution as one in which two promises provide messages that are in each other's respective preconditions.

Any execution of the classic OOTA litmus test which produces a thin air value will at some point contain promises $p_1$ and $p_2$:

```
r1 := x;    ||  r2 := y;
y := r1;        x := r2;
```

$$\{\langle x, 42@1\rangle\}\ p_1\ \langle y, 42@1\rangle$$

$$\{\langle y, 42@1\rangle\}\ p_2\ \langle x, 42@1\rangle$$

Promising correctly forbids this because we cannot construct an initial speculative execution for either $p_1$ or $p_2$ without the other. We cannot make $p_1$ first, because we cannot construct an execution from a blank memory state which writes 42 to $y$. Likewise we cannot make $p_2$ first because no such execution will write 42 to $x$ either.

The sequence in Fig. 7, however, results in this OOTA shape by swapping one promise from a permissable precondition into a cyclical one *after* both promises have been made.

4: Write $\langle x, 1@2\rangle$
Promise: $\emptyset_{\langle \mathcal{TS}_1, M_1, S_1\rangle}\ p_1\ \langle y, 1@1\rangle$
    1: Write $\langle x, 2@1\rangle$
    2: Read $\langle x, 1@2\rangle$
    3: Write $\langle y, 1@1\rangle$
Promise: $\{\langle y, 1@1\rangle\}_{\langle \mathcal{TS}_1, M_1, S_1\rangle}\ p_2\ \langle x, 3@4\rangle$
    5: Read $\langle x, 1@2\rangle$
    6: Read $\langle y, 1@1\rangle$
    7: Write $\langle x, 3@4\rangle$
1: Write $\langle x, 2@3\rangle$
Verify: $\{\langle x, 3@4\rangle\}_{\langle \mathcal{TS}_2, M_2, S_2\rangle}\ p_1\ \langle y, 1@1\rangle$
    2: Read $\langle x, 3@4\rangle$
    3: Write $\langle y, 1@1\rangle$
        $\vdots$

Figure 9: The operational steps of the OOTA7 program with promise preconditions annotated.

It begins by making two promises from the same point in evaluation, where $S_1$ is the remainder of the program after executing line 4:

$$\emptyset_{\langle \mathcal{TS}_1, M_1, S_1\rangle}\ p_1\ \langle y, 1@1\rangle \qquad \{\langle y, 1@1\rangle\}_{\langle \mathcal{TS}_1, M_1, S_1\rangle}\ p_2\ \langle x, 3@4\rangle$$

In the next verification step, where $S_2$ is the remainder of the program after lines 1 and 4, we are forced to use the result of $p_2$ to verify $p_1$:

$$\{\langle x, 3@4\rangle\}_{\langle \mathcal{TS}_2, M_2, S_2\rangle}\ p_1\ \langle y, 1@1\rangle \qquad \{\langle y, 1@1\rangle\}_{\langle \mathcal{TS}_2, M_2, S_2\rangle}\ p_2\ \langle x, 3@4\rangle$$

We now have that $p_1$ provides the precondition for $p_2$ and likewise $p_2$ provides the precondition for $p_1$.

### 2.3.2 The Java Memory Model

The Java memory model, released as JSR-133: The Java Memory Model and Thread Specification by Pugh, Adve and Lea (2011), describes the permitted behaviour of concurrent programs running on the Java Virtual Machine using the *Thread* class. The goal of the model is to provide defined behaviour for all well-typed programs which use *Thread*, including those with data races, while simultaneously allowing existing optimisations and preserving type safety. Similar to Promising, it is a per-execution model which begins evaluating programs in syntactic order and progressively justifies out-of-order accesses. Unlike Promising, however, it correctly forbids justification cycles but as a collateral forbids optimisations which should be permitted.

The model divides program variables into local variables, which can only be accessed within a thread, and global variables. An assignment to a local variable from a global variable is a *read*. An assignment to a global variable from a local variable is a *write*. Any operations which only impact local variables and cannot impact global ones are handled by the existing intra-thread semantics for Java, and largely ignored by the memory model. It includes a handful of other special actions not given in detail here, but used for synchronisation operations and special circumstances such as interrupts and exceptions.

#### Definitions

The operations performed by a program are called *actions*, and given by a tuple

$$(t, k, v, u)$$

Where:

- $t$ identifies the thread in which the action occurs
- $k$ describes the *kind* of the action, which will be one of the categories mentioned above (for our purposes, read or write)
- $v$ gives the variable referenced by an action
- $u$ is a unique identifier for this action

A write has a static value, described by the function $V(w)$. The value observed by a read within a particular execution is described by the value of its corresponding write, where $W(r)$ gives the write being observed and $V(W(r))$ gives the resulting value. Note that it is impossible to define a read which does not have a corresponding write. All programs must begin with a series of initialising writes, which usually set all global values to 0.

An execution consists of a set of these actions, the program which generates them, and a series of relations over them. Executions are written as a tuple

$$(P, A, \sqsubseteq, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb})$$

Where:

- $P$ is the program
- $A$ is a set of actions
- $\sqsubseteq$ is the *program order* relation, which is the order in which actions occur in the program text (note that this is a partial order, as actions from one thread are not $\sqsubseteq$ before or after actions in a different thread)
- $\xrightarrow{so}$ is the *synchronisation order* relation, which is a total order over all synchronisation actions in $A$
- $W$ and $V$ are the observed write and value functions
- $\xrightarrow{sw}$ is the *synchronises-with* relation, which is not used in the examples given here but, in the full model, orders various synchronisation primitives and special actions
- $\xrightarrow{hb}$ is the *happens-before* relation, which is the transitive closure of program order and synchronises-with: $(\sqsubseteq \cup \xrightarrow{sw})^+$

The specification is careful to note that two events being related by $\xrightarrow{hb}$ does not cause them to always be performed by the JVM in this order. Instead, all executions must preserve a property called *happens-before consistency*.

Happens-before consistency requires that, if we have a read $r$ which observes write $w$ (such that $W(r) = w$), the following must hold:

- The write $w$ is not $\xrightarrow{hb}$-after $r$
- There is no write $w'$ to the same variable in $A$ such that $w \xrightarrow{hb} w' \xrightarrow{hb} r$

Not all happens-before consistent executions are permitted by the model. It also requires that one can define an order in which events are *committed*, using a somewhat operational-style construction.

An execution $E$ can be restricted to a *sub-execution* $E_i$ by taking a subset of the actions and restricting all relations in $E$ to this subset. The actions in $E_i$ are written $A_i$, the functions $V$ and $W$ as $V_i$ and $W_i$, and the relations likewise disambiguated by subscript. For each such execution we define a set of *committed* actions $C_i$ such that $C_i \subseteq A_i$.

The values written by all committed writes (meaning writes in $C_i$) must be the values written in the final execution, but we do *not* require this for values in $A_i \setminus C_i$. The writes observed by reads in the *previous* commit set $C_{i-1}$ must be equal to the writes observed by the complete execution, but likewise we do *not* require this for writes in $A_i \setminus C_i$ or in $C_i \setminus C_{i-1}$.

$$V_i|_{C_i} = V|_{C_i}$$
$$W_i|_{C_{i-1}} = W|_{C_{i-1}}$$

This means that we are allowed to "swap" the value observed by a read when we move it from $A_{i-1}$ to $C_i$, but we cannot swap again afterwards. This is necessary to observe potentially racy values, because we also require that whenever we initially add a read to our execution, it must observe a value from a write which is guaranteed to be observable by being $\xrightarrow{hb}$-before it. However, the write it observes in the final execution must also have been added to a previous commit set even if this is not the write it observes in the execution $E_i$.

$$\text{For any read } r \text{ in } A_i \setminus C_{i-1}, \text{ we have } W_i(r) \xrightarrow{hb_i} r.$$
$$\text{For any read } r \text{ in } C_i \setminus C_{i-1}, \text{ we have } W_i(r) \in C_{i-1} \text{ and } W(r) \in C_{i-1}.$$

We require that the union of all these commit sets is equal to the action set of our final execution: $A = \cup(C_0, C_1, ...)$. If the set $A$ is finite, then we will arrive at some final $C_n = A$. If $A$ is not finite, then the infinite union of $C_i$ must be equal to $A$.

This is best demonstrated by example.

**Reordering by Example**

```
              0: x = 0
              0: y = 0
1: r1 = x;  || 3: r2 = y;
2: y = 1;   || 4: x = r2;
```

Programs are assumed to begin with all variables set to 0. As each read requires a corresponding write, a series of initialising writes are included at the start of the program and denoted by the line number 0. For ease of reading, the line numbers are used here as identifiers. This is not mandated by the standard, as action identifiers need only be unique for each action performed by an execution and do not need to be unique for each potential action in the program, but it clarifies the relationship between actions and program syntax.

This code contains a data race: there's no synchronisation between the write on line 2 and the read on line 3, so the read may either observe the write of 1 or the initial write of 0. Due to the reorderings described in section 2.2.1, the outcome `r1` = `r2` = 1 is allowed here. The write on line 2 commits before the read on line 3, and the write on line 4 before the read on line 1. Using line numbers as action identifiers, this gives us the following components for our final execution $E$:

$$A = \{0, 1, 2, 3, 4\}$$

$$V(2) = 1 \quad V(4) = 1$$

$$W(1) = 4 \quad W(3) = 1$$

Our first execution $E_1$ must be a well-formed execution of the program, which means every read must have a corresponding write and the action set must correspond to an acceptable execution of each thread under the thread-local semantics. Since the initial commit set $C_0$ is the empty set, every read in the execution must be in $A_1 \setminus C_0 = A_1$, which means that all $r$ must satisfy the requirement $W_1(r) \xrightarrow{hb_1} r$. The only writes we can observe at lines 1 and 3 are the initialising writes of 0, as the other writes are in different threads and do not synchronise before their respective reads.

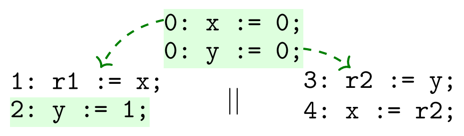
We then create the commit set $C_1 = \{0, 2\}$. This means we have committed the initial writes, as well as the write of 1 to $y$ on line 2. We don't want to commit line 4 yet, because its value in our final execution is not the value we observe here. As long as 4 isn't in $C_1$, then $V_1(4)$ and $V(4)$ can differ.

The program text in Fig. 10 has been annotated with the $W_1$ function shown as dashed green arrows and $C_1$ highlighted in light green.

$$A_1 = \{0, 1, 2, 3, 4\}$$

$$V_1(2) = 1 \quad V_1(4) = 0$$

$$W_1(1) = 0 \quad W_1(3) = 0$$



```
                      0: x := 0;
                      0: y := 0;
1: r1 := x;                          3: r2 := y;
2: y := 1;         ||                4: x := r2;
```

Figure 10: Execution $E_1$ and commit set $C_1$

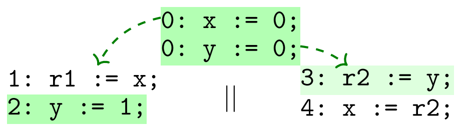
The next event we want to commit is 3, so $C_2 = \{0, 2, 3\}$. We can't initially set $W_2(3) = 2$, as in our final execution, because reads outside of $C_{n-1}$ can only observe writes which are $\xrightarrow{hb}$-before them. The only such event for line 3 is the initialising write of 0. We will be allowed to change the write being observed in our next execution, but for now our only option is to set $W_2(3) = 0$. We now need to verify that $W_2(3) \in C_1$ and $W(3) \in C_1$. This clearly holds, since both 0 and 2 are in $C_1$.

In Fig.11, the events we committed in the set $C_1$ are shown in a darker green.

$A_1 = \{0, 1, 2, 3, 4\}$

$V_1(2) = 1 \quad V_1(4) = 0$

$W_1(1) = 0 \quad W_1(3) = 0$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                 3: r2 := y;
2: y := 1;        ||        4: x := r2;
```

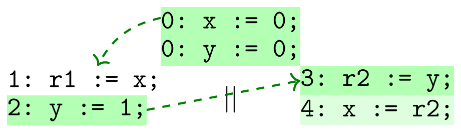
Figure 11: Execution $E_2$, which is the same as $E_1$, and commit set $C_2$.

Moving on to $C_3$, the next event we want to commit is the write on line 4. This gives us $C_3 = \{0, 2, 3, 4\}$. Because 3 is no longer in $A_i \setminus C_{i-1}$, we no longer need it to observe a write that comes before it in $\xrightarrow{hb}$. In addition, we now need to ensure that $W_3(3)$ is the same as $W(3)$: it must be true that $W_3|_{C_2} = W|_{C_2}$. We swap from $W_2(3) = 0$ to $W_3(3) = 2$, and this changes the value written on line 4 from 0 to 1. Having not yet committed 4, this value swap is permitted.

$A_3 = \{0, 2, 3, 4\}$

$V_3(2) = 1 \quad V_3(4) = 1$

$W_3(1) = 0 \quad W_3(3) = 2$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                 3: r2 := y;
2: y := 1;        ||        4: x := r2;
```

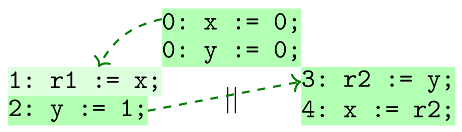
Figure 12: Execution $E_3$ and commit set $C_3$

Finally, we add line 1 to the commit set $C_4 = \{0, 1, 2, 3, 4\}$. Note again that we can't have $W_4(1) = 4$, because we're still restricted to writes which are $\xrightarrow{hb}$-before 4. This isn't a problem though, as we're still allowed to swap which read we observe in the step between executions $E_4$ and $E$ as we did with the previous read between $E_2$ and $E_3$.

$A_4 = \{0, 1, 2, 3, 4\}$

$V_4(2) = 1 \quad V_4(4) = 1$

$W_4(1) = 0 \quad W_4(3) = 2$



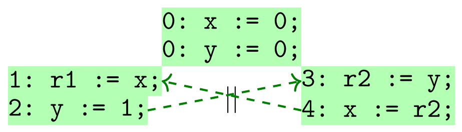
```
                0: x := 0;
                0: y := 0;
1: r1 := x;                3: r2 := y;
2: y := 1;       ||        4: x := r2;
```

Figure 13: Execution $E_4$ and commit set $C_4$

$A = \{0, 1, 2, 3, 4\}$

$V(2) = 1 \quad V(4) = 1$

$W(1) = 4 \quad W(3) = 2$



```
                0: x := 0;
                0: y := 0;
1: r1 := x;                3: r2 := y;
2: y := 1;       ||        4: x := r2;
```

Figure 14: Execution $E$ with all actions committed

### Branch Elimination by Example

The model is also able to represent optimisations in the presence of branches. We can illustrate this by adding a branch statement to the previous program and looking for the same execution:

```
1: r1 := x;
if (r1 = 1) {
  2: y := r1;        4: r2 := y;
} else {        ||   5: x := r2;
  3: y := 1;
}
```

The write on line 2 is always going to have value 1, thus we should be able to observe an execution in which `y := 1` happens above the guard.

We initially proceed as before, beginning with the write of the constant and propagating this value to the second thread, then copying it into `x`.

$$C_1 = \{0, 3\} \quad C_2 = \{0, 3, 4\} \quad C_3 = \{0, 3, 4, 5\} \quad C_4 = \{0, 1, 3, 4, 5\}$$

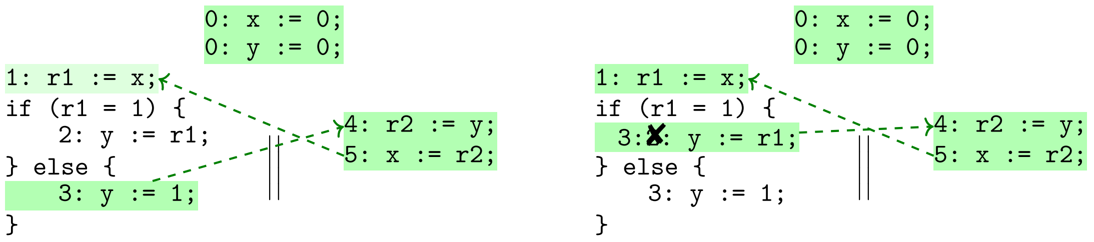


Figure 15: Commit sets $C_4$ and $C_5$

We now want to change from observing 0 at line 1 to observing 1. This means that the thread-local semantics will no longer allow us to execute line 3, and we must instead execute line 2. Recall, however, that the unique identifier corresponding to an action is *arbitrary*. We can perform an action at line 2 which is identical to line 3 simply by declaring that the identifiers are the same, since they are the same type of access to the same location with the same associated value. This results in our final commit set $C_5 = \{0, 1, 2, 4, 5\}$.

**Failure to Permit Known Optimisations**

This model has shown to be too restrictive with respect to the actual behaviour of the Hotspot compiler by Ševčík and Aspinall (2008). The following sequence of optimisations has been observed in practice, but results in behaviours that are forbidden by the model:

```
1: r1 := y;
if (r1 = 1) {
  2: r2 := y;
  3: x := r2;   →   1: r1 := y;   →   1: x := 1;
} else {            2: x := 1;        2: r1 := y;
  4: x := 1;
}
```

To work out how and why this is forbidden, let us first create a context that can distinguish the first program from the last:

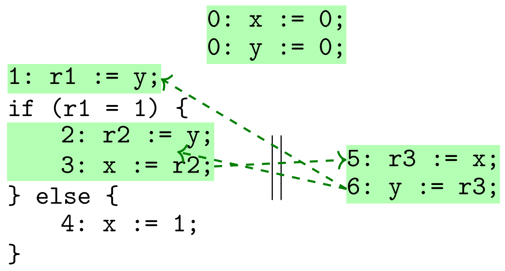
```
1: r1 := y;
if (r1 = 1) {
  2: r2 := y;   ||   5: r3 := x;
  3: x := r2;   ||   6: y := r3;
} else {
  4: x := 1;
}
```

If the reordering between the read from $y$ and the write to $x$ is permitted, then we can observe an execution where line 1 observes the value 1 from the write at line 6, while the read at line 5 observes the write on line 3, hoisted above the branch.

$A = \{1, 2, 3, 5, 6\}$

$V(3) = 1 \quad V(6) = 1$

$W(1) = 6 \quad W(2) = (6) \quad W(5) = 3$



```
                              0: x := 0;
                              0: y := 0;
1: r1 := y;
if (r1 = 1) {
    2: r2 := y;          ||
    3: x := r2;          ||   5: r3 := x;
} else {                 ||   6: y := r3;
    4: x := 1;
}
```

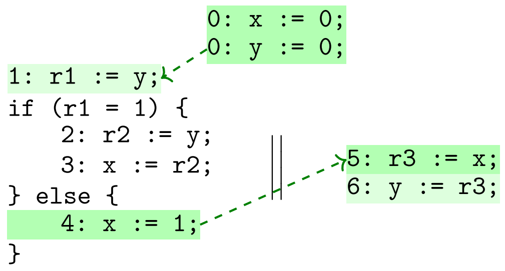
As before, we need to commit the hoisted write first in order to observe its value later, and then swap branch after altering the value observed at line 1. We take the following path through the program:

$$C_1 = \{0, 4\} \quad C_2 = \{0, 4, 5\} \quad C_3 = \{0, 1, 4, 5, 6\}$$

$A_3 = \{0, 1, 4, 5, 6\}$

$V_3(4) = 1 \quad V_3(6) = 1$

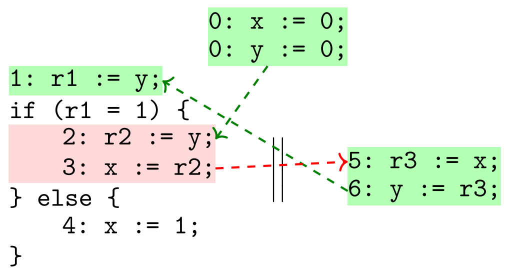
$W_3(1) = 0 \quad W_3(5) = 4$



```
                              0: x := 0;
                              0: y := 0;
1: r1 := y;
if (r1 = 1) {
    2: r2 := y;          ||
    3: x := r2;          ||   5: r3 := x;
} else {                 ||   6: y := r3;
    4: x := 1;
}
```

In order to let line 1 read from line 6 and swap branch, we need to ensure that line 3 writes a value of 1 to match line 4. As before, we must invoke the thread-local semantics to propagate the value in `r2`. This is a problem: line 2 can only observe the value of 0 until the commit set *after* this one. We need to have at least one set in which it takes its value from a write which is $\xrightarrow{hb}$-before it, and the only such write is the initialising write of 0. On the other hand, this is the last commit step where we can change the value we observe at line 1. If we take another step while reading from the initialisation write, then our final execution must also read from the initialisation write here.

Because we can't read 1 into `r2` in the next commit step, line 3 cannot perform the already-committed write of 1 to $x$ created at line 4. It would need to spend at least one commit step writing 0, which changes the commit set. This is illustrated in Fig. 16.



```
                    0: x := 0;
                    0: y := 0;
1: r1 := y;
if (r1 = 1) {
    2: r2 := y;
    3: x := r2;          ||   5: r3 := x;
} else {                      6: y := r3;
    4: x := 1;
}
```

Figure 16: Line 5 expects to see value 1, but the left thread is copying value 0 into `x`.

The main issue here is a failure to recognise that not all reads in a program will actually take their value from a write. There is a sequentially permissable optimisation which removes line 2 entirely, and propagates the value 1 into all in-scope references to `r3`. The model, meanwhile, is insensitive to the guarantees from the guard condition.

## 2.4 Moving Away From Trace-Based Semantics

The JMM and Promising have a particular feature in common which underlies both of their failures. They want to determine which values a read could potentially observe by starting at a sequentially consistent trace, then using a set of modification steps over that trace in order to change the observed value. Java does this execution-by-execution via the commit sets, while Promising does this step-by-step operationally via promises. In either semantics, the values a read is allowed to observe are given by the set of writes which "might" happen before it, and the machinery of the model is dedicated to filling that set prior to executing the read and continuing in the program. This causes the JMM to fail in cases where a read cannot observe the desired value within a single commit step, and Promising to fail in cases where earlier operational steps make assertions about program invariants that aren't preserved at later steps.

In either case, the potential ordering of statements in one thread is strongly dependent on any other threads running in parallel. Intra-thread reordering operations are never represented explicitly, thus optimisations which will always be permitted in one thread are not always visible to the semantics. This gives rise to degenerate contexts in which an intra-thread reordering operation which should be observable can be blocked by one component of a parallel thread but should be observed by another, as in the `x := y` context for the JMM (in which the load from $x$ observes a reordered write of 1 and the store to $y$ blocks the reordering). It can also cause a semantics to allow a reordering that is sequentially unsound, such as the removal of the guard in `if (r1 != 0) {x := 3}` allowed by the Promising semantics.

If the link between these is the requirement to validate the plausibility of the input to a semantic step prior to executing it and allowing the output, then we might propose a semantics which avoids this entirely. We instead move to validate every choice at a point of nondeterminism – be it the ordering of events or the value observed by a read – from the context of the entire program, rather than attempting during interpretation to construct a program fragment which might be plausibly before it.

Rather than the "permit, then proceed" approach shown here, other models take a "proceed, then permit" approach. However, this requires an overview of potential executions which is substantially more modular than a constructed set of execution traces. The MRD model uses *event structures* to represent the control and data flow within a program explicitly, which allows it to represent a theoretical execution without needing to validate beforehand that every step of the execution may happen given prior knowledge of the program.

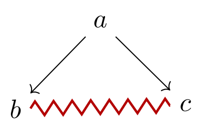
In the next section we introduce event structures, and give an overview of two other memory models which use them to make decisions about program execution.

### 2.4.1 Event Structures

Event structures were introduced by Nielsen, Plotkin and Winskel (1981) to combine various formalisms for the evolution of a computational state given internal computational steps and external inputs. Several memory models have already been proposed, by Ševčík and Aspinall (2008), Jeffrey and Riely (2016), Lahav et al. (2017), and Chakraborty and Vafeiadis (2019), which use event structures to describe concurrent programs.

An *event*, at its most basic, has three properties: it is atomic, it has a set of events which must occur before it can begin, and it has a set of events with which it cannot simultaneously occur. In the context of a program running on an abstract machine, any operational step can be considered an "event". In MRD, we consider only global memory accesses and fences to be events, while any arithmetic operation over local variables is invisible.

Event structures are written as a tuple $(E, \leq, \#)$, where $E$ is the set of events, $\leq \subseteq E \times E$ is a causality relation, and $\# \subseteq E \times E$ is a conflict relation. These are commonly presented as graphs, where $\leq$ is used as a parent-child relation.



$$E = \{a, b, c\}$$

$$\leq = \{(a, b), (a, c)\}$$

$$\# = \{(b, c)\}$$

Figure 17: A graph representation (left) and set representation (right) of the same event structure

If $e_1 \leq e_2$, then $e_2$ cannot occur until $e_1$ has finished. If $e_1 \# e_2$, then $e_1$ and $e_2$ cannot both happen.

When considering events as memory accesses and event structures as programs, we can use $\leq$ to denote the syntactic order in which these accesses are written. If line 1 causes event $e_1$ and line 2 causes event $e_2$, then we can assert that $e_1 \leq e_2$. This loosens the interpretation of the $\leq$ relation from "causality" to "ordering" – there doesn't need to be any causal relationship between the events being ordered, but nonetheless we would expect them to occur in the order given.

In the same setting, the conflict relation can be used for data and control flow. For instance, if $e_1$ happens inside an `if` block and $e_2$ inside the corresponding `else` block, we can assert that $e_1 \# e_2$. If $e_1$ represents loading the value 1 from location $x$ and $e_2$ represents loading the value 2 from location $x$, then, if $e_1$ and $e_2$ arise from the same statement, it should be the case that $e_1 \# e_2$.

A *configuration* is a subset of $E$ which is left-closed with respect to $\leq$ and contains no $\#$ edges. It represents a complete set of events which could plausibly happen: the closure under $\leq$ enforces that all predecessors are met, and the avoidance of $\#$ enforces that events in the same configuration cannot prevent or exclude one another.

In the graphic interpretation of event structures, this means a configuration must contain a path from all events to their root node, and contain no event-to-event path which uses a conflict edge.

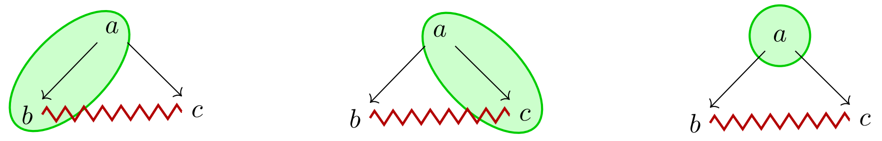


Figure 18: The three possible configurations, highlighted in green, of a single event structure

When event structures are interpreted as programs, configurations can be interpreted as executions. If we have executed a program until event $e$, then we should have executed everything "before" $e$, which is represented by the closure under $\leq$. If event $e$ has appeared in an execution, then no events which require mutually exclusive data or control flow to $e$ should appear in the same execution, which is represented by the lack of # edges.

Weak memory models which use event structures often add ancillary relations between events, such as the writer-to-reader edges included in the Java Memory Model and synchronisation edges between memory fences. They use these relations to impose additional constraints on configurations, or executions, forbidding some patterns of behaviour and permitting others.

### 2.4.2 WeakestMO

The WeakestMO model represents programs as event structures, derives several novel relations across events, then determines the allowed configurations of these structures under a set of restrictions.

Event labels in WeakestMO take the following forms:

- Loads, written `Ld(X, v)`, indicate that the value $v$ was loaded from global location $X$
- Stores, written `St(X, v)`, indicate that the value $v$ was stored in global location $X$
- Updates, written `U(X, v, v')`, indicate that the content of $X$ was atomically updated from $v$ to $v'$
- Fences, written `F`, indicate that a memory fence was inserted into the program at a specific point

These can be subscripted with a *memory order*, which can provide additional ordering constraints. To compare WeakestMO with the other models presented so far, we avoid using updates, fences, and memory orders and focus on relaxed store and load operations.

Events also store information about their containing thread, accessed via a `tid` field, and an identifier `id` unique to that event. The total signature of an event is a tuple:

$$(\texttt{id, tid, lab})$$

In the structures themselves, events are typically written using only the identifier and label, separated by a colon:

$$\texttt{id} : \texttt{lab}$$

Events are ordered by *program order*, which takes the place of the causality relation. Program order represents the syntactic order of the statements which triggered the respective events.

The WeakestMO structures take the form:

$$(E, \texttt{po}, \texttt{jf}, \texttt{ew}, \texttt{mo})$$

Where:

- $E$ is the set of all events in the structure
- po is the *program order* relation, which orders events according to the syntactic order of their respective statements in the program.
- jf is the *justifies* relation, which relates a write to a read which observes its value.
- ew is the *equal writes* relation, which relates conflicting writes of the same value to the same location.
- mo is the *modification order* relation, which must totally order all non-conflicting writes to the same location.

Conflict, written cf, is not explicitly represented in WeakestMO , but instead two events are declared to be in conflict if they are in the same thread but are not ordered by program order.

The structures used by the model are built event-by-event, with each generated event coming from an interpretation of the program text with respect to the already-built sections of the structure. The simplest way to build a WeakestMO model is to interpret the program line-by-line, which builds a single complete execution. However, it is possible to "re-execute" a line if this would place a new event in conflict with the existing events generated by the same statement.

An event can only be added if all the following conditions are satisfied:

- All statements syntactically before the statement being considered have been executed in the current execution. If a program consists of a store to $x$ on line 1 followed by a store to $y$ on line 2, then the event corresponding to the store to $y$ can only be added to a structure which already contains the store to $x$. This event must not be in conflict with the new store.
- A load event can only be added to the structure if a store event of the same value to the same location is already present in the structure. This causes a jf edge to appear from the store to the load.

An *execution* of the model is not defined solely as a restriction to a conflict-free set. Extracting executions from a WeakestMO structure requires the derived reads-from relation rf, and a property known as *visibility*.

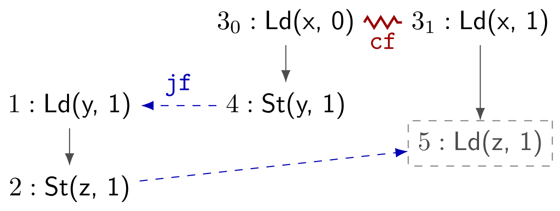
**Reads-from** An RF edge $e_1 \xrightarrow{\text{RF}} e_2$ arises whenever there is a jf edge such that either:

1. $e_1 \xrightarrow{\texttt{jf}} e_2$, or
2. There is some event $e$ such that $e_1 \xleftrightarrow{\texttt{ew}} e \xrightarrow{\texttt{jf}} e_2$

Every load in the proposed execution should have a corresponding store and rf edge in the same execution.

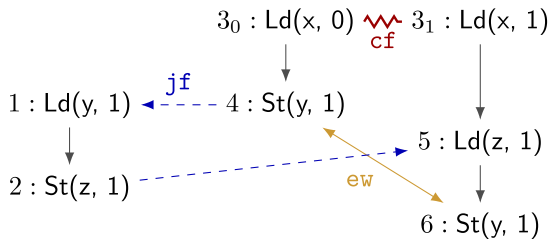
**Visibility** While it is possible for a store in one execution to justify a load in another, the execution containing the load cannot subsequently justify anything in the execution containing the store. This type of cycle is referred to as *invisibility*, and all events in an execution must be visible for the execution to be permitted by the model.

Here we show visibility and invisibility by example:



In this structure, event 2 is recursively justified by event 4 and justifies event 5. This means that event 5 is invisible, and cannot appear in an execution.

However, if another event equivalent to 4 under the ew relation appeared in the same execution as event 5, and in the same thread, event 5 would become visible.



The visibility predicate is given formally as:

$$Vis(e) \Leftrightarrow [W]; (\texttt{cf} \cup \mathsf{jfe}; (\texttt{po} \cup \texttt{jf})^*; \mathsf{jfe}; \texttt{po}^?); [\{e\}] \subseteq \texttt{ew}; (\texttt{po} \cup \texttt{po}^{-1})^?$$

Where jfe indicates inter-thread jf edges.

As the WeakestMO model then derives a large number of additional relations for use in consistency checking, we proceed by example and introduce the additional relations and checks when relevant. We do not cover every relation used by the model, only those which differentiate it from other models in the particular tests shown here.

### Constructing Event Structures by Example

We begin with the relatively simple load buffering litmus test, LB.

```
        0: x = 0
        0: y = 0
1: r1 = x; || 3: r2 = y;
2: y = 1;  || 4: x = r2;
```

In this test, we are interested in observing the outcome $r_1 = r_2 = 1$. This outcome indicates that the independent write on line 2 can be reordered above the read according to the model.

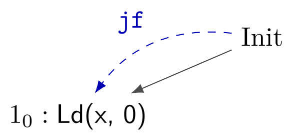
We let the first two lines be represented by a single Init event, which writes 0 to all variables. This gives us a structure with a single event and no relations.

We can now choose which event to add to the structure next: we could interpret the first line of the left-hand thread, creating a load from $x$, or the first line of the right-hand thread, creating a load from $y$. In either case, the only value we could observe is 0.

Choosing to begin with the left-hand thread, we add a new load event by:

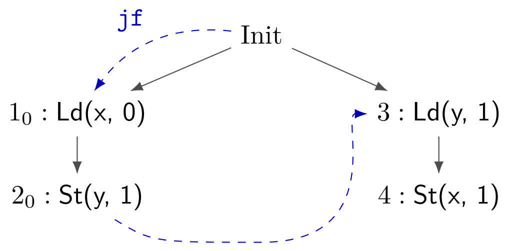
1. Creating a new event $1_0$ : Ld(x, 0) and adding it to the event set
2. Adding a po edge from the initial event to event $1_0$, indicating that the load occurs program order-after the initialising event
3. Adding a jf edge from the initial event to event $1_0$, indicating that the value read can be justified by the initialising store

This gives the following structure:



$$E = \{Init, 1_0\}$$
$$\texttt{po} = \{(Init, 1_0)\}$$
$$\texttt{jf} = \{(Init, 1_0)\}$$
$$\texttt{ew} = \emptyset$$
$$\texttt{mo} = \emptyset$$

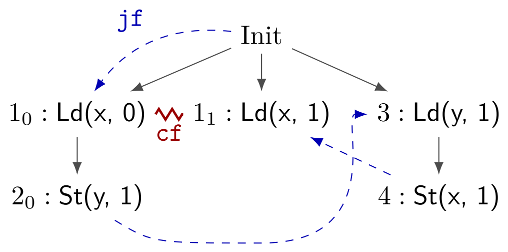
We can use the same steps to finish the left-hand thread and fully evaluate the right-hand thread, letting the right-hand thread observe a value of 1 at $y$ from the write in the left-hand thread:



$$E = \{Init, 1_0, 2_0, 3, 4\}$$
$$\texttt{po} = \{(Init, 1_0), (Init, 3), (1_0, 2_0), (3, 4)\}$$
$$\texttt{jf} = \{(Init, 1_0), (2_0, 3)\}$$
$$\texttt{ew} = \emptyset$$
$$\texttt{mo} = \emptyset$$

This gives a complete execution of the program, but not the execution of interest. However, we can continue to add events to the structure using the conflict relation.

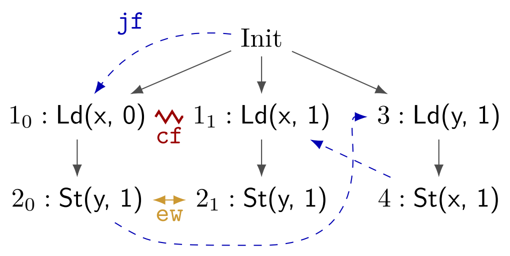
If we re-interpret the load from $x$, we can now observe a value of 1, justified by the store at event 4.



$$E = \{Init, 1_0, 1_1, 2_0, 3, 4\}$$
$$\texttt{po} = \{(Init, 1_0), (Init, 1_1), (Init, 3), (1_0, 2_0), (3, 4)\}$$
$$\texttt{jf} = \{(Init, 1_0), (2_0, 3), (4, 2_1)\}$$
$$\texttt{ew} = \emptyset$$
$$\texttt{mo} = \emptyset$$

This event appears in conflict with event $1_0$, and directly after the initialising event.

To complete this second execution, we only need to re-interpret the store to $y$. This creates a new event, $2_1$ : St(y, 1). As this has the same location and value as $2_0$ : St(y, 1) and is in conflict with $2_0$, this creates a new edge in the equal writes relation `ew`.



$$E = \{Init, 1_0, 1_1, 2_0, 2_1, 3, 4\}$$

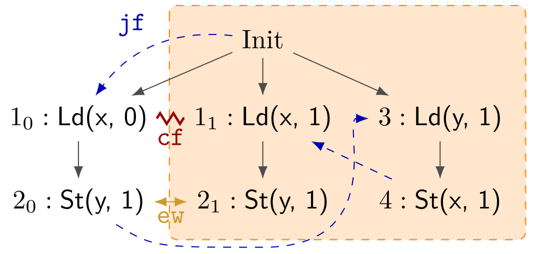
$$\texttt{po} = \{(Init, 1_0), (Init, 1_1), (Init, 3), (1_0, 2_0), (1_1, 2_1),$$

$$\texttt{jf} = \{(Init, 1_0), (2_0, 3), (4, 2_1)\}$$

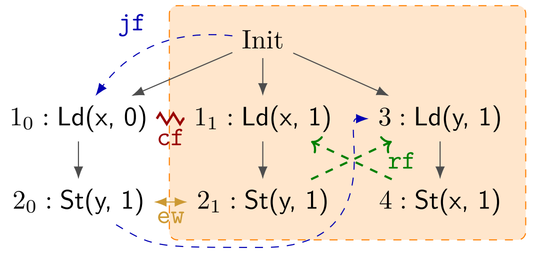
$$\texttt{ew} = \{(2_0, 2_1)\}$$

$$\texttt{mo} = \emptyset$$

We can now attempt to extract a new execution from this structure. We want this execution to include the load of 1 from $x$ and subsequent store to $y$, resulting in the desired outcome $r_1 = r_2 = 1$.



For this execution to be permitted by the model, every load must have a corresponding store and rf relation. We can convert the jf relation from event 4 to event $1_1$ into an rf edge directly, while using the series of edges $2_1 \xleftrightarrow{\texttt{ew}} 2_0 \xrightarrow{\texttt{jf}} 3$ to create a $2_1 \xrightarrow{\text{RF}} 3$ edge.



The re-interpretation step used to construct this model is similar to promise verification steps that swap which statement will ultimately fulfil the promise. A store appears in one execution and is used to justify behaviour in another which would otherwise be cyclical. Indeed, it is shown in Chakraborty and Vafeiadis (2019) that a variant of WeakestMO which lacks certain consistency checks allows all behaviours which are allowed by the Promising semantics.

In the next example we highlight the additional relation which causes WeakestMO to reject behaviours permitted by Promising.

**Differentiating WeakestMO From Promising**

```
1: x := 2;              4: x := 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2){     ||    6: r3 := y;
  3: y := 1;            if (r3 != 0){
}                         7: x := 3;
                        }
```

Recall that in Section 2.3.1, the Promising semantics was able to observe the outcome $\texttt{r1} = 3 \land \texttt{r2} = 2 \land \texttt{r3} = 1$. This outcome is forbidden by WeakestMO by the inclusion of the `mo` relation, put permitted without it. The variant of the model which does not use a structure-level `mo` relation is named the Weakest model, and we first show how Weakest arrives at the permitted behaviour. We then introduce `mo`, and show how it prevents this outcome.

We begin by executing the initialising event, followed by the stores on lines 1 and 4.

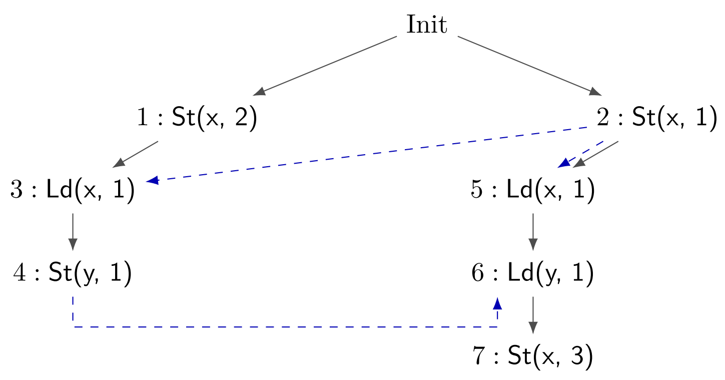
$$\begin{tikzcd}
 & \text{Init} \arrow[dl] \arrow[dr] & \\
1 : \mathsf{St}(x, 2) & & 2 : \mathsf{St}(x, 1)
\end{tikzcd}$$

At this point, the Promising semantics was able to first promise the future stores of 1 to $y$ on line 3 and then promise the store of 3 to $x$ on line 7, given a plausible future execution in which each respective store occurred. Similarly, in this model we first try to create an execution in which the store to $y$ occurs, and then use it to create a new `jf` edge into a new execution in which the store to $x$ occurs.

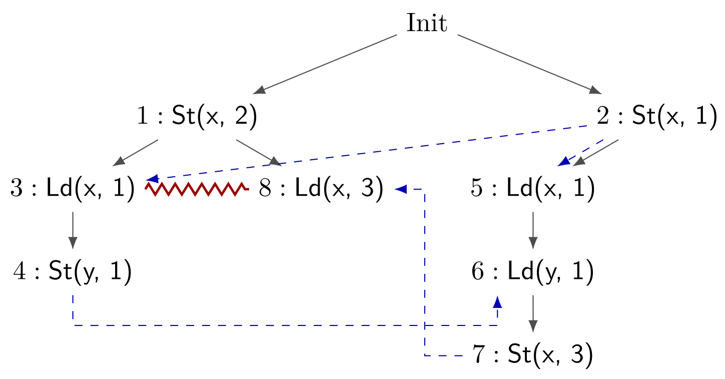
To reach a store to $y$ from this structure, we continue to execute the left-hand thread, using the store of 1 to $x$ from the right-hand thread to load a value which is not 2.

$$\begin{tikzcd}
 & & \text{Init} \arrow[dl] \arrow[dr] & \\
 & 1 : \mathsf{St}(x, 2) \arrow[dl] & & 2 : \mathsf{St}(x, 1) \arrow[dlll, dashed] \\
3 : \mathsf{Ld}(x, 1) \arrow[d] & & & \\
4 : \mathsf{St}(y, 1) & & &
\end{tikzcd}$$

We can then load from this new store in the right-hand thread, providing the store to $x$.

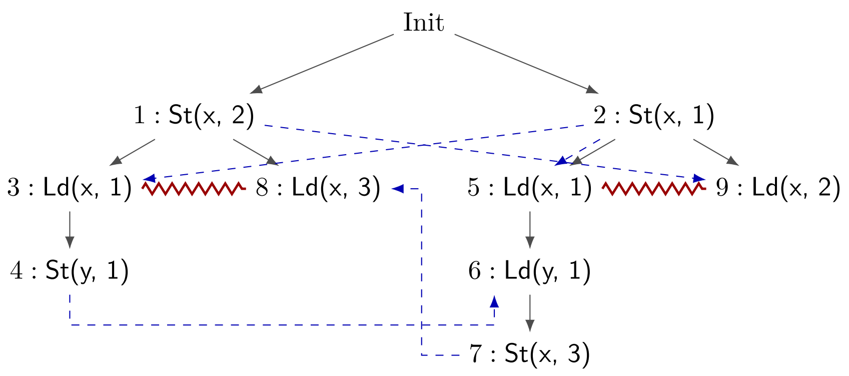


While this constitutes a permitted execution, it does not display the behaviour we are looking for. We can continue to add events to the structure, though, and use the new store to $x$ at event 7 to justify a new execution of the left-hand thread.

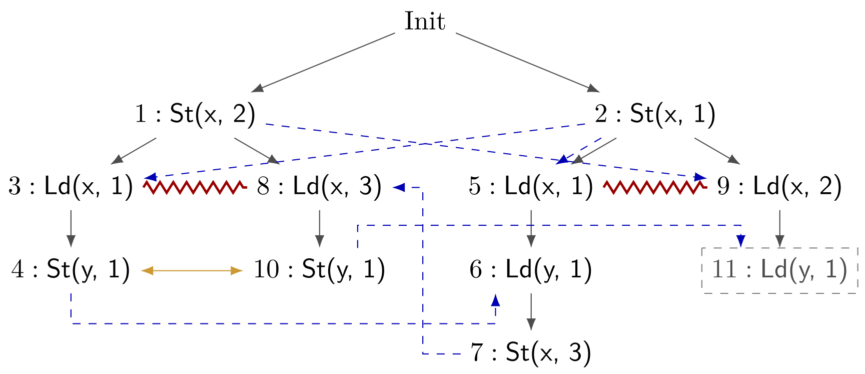


This step is similar to the execution-switching allowed in Promising's verification step: we may not actually execute this event, but we were able to construct a coherent execution which does, and we can subsequently load from that execution's store. To complete the execution, we needed to find an equivalent store that we *do* execute, and the same holds here. As event 7 is recursively justified by an event in conflict with 8, we cannot create an RF edge between them directly. To make event 8 visible again, we need to find an equivalent store which does not conflict with it.

We start by creating a new execution of the right-hand thread, choosing to load 2 from $x$ instead of 1.

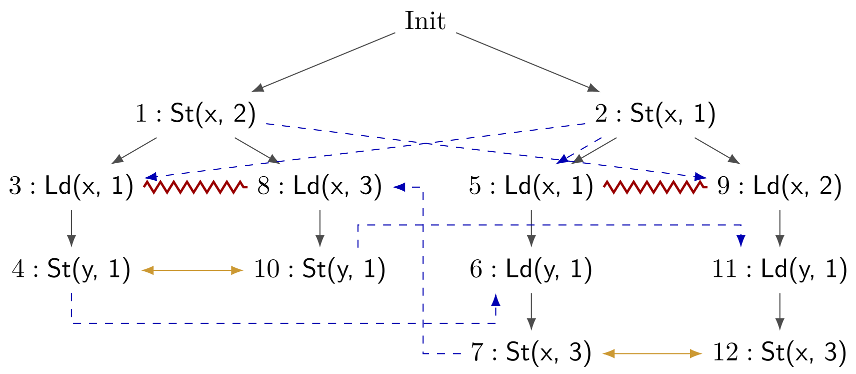


We can then create a new store to $y$ on the left and load from it on the right, but the new event will be invisible.



Event 11 is justified by event 10, which is recursively justified by event 7. As event 7 is in conflict with event 11, the latter is temporarily invisible.

The final step in creating our target structure makes event 11 visible again by adding in a store equal to event 7.



**Introducing Modification Order** The intuitive justification for forbidding this structure is as follows:

1. The target execution can only appear if event 1 occurs after event 2, overwriting the value at $x$ and allowing event 9 to occur.

2. This execution can only be built by justifying a load using event 7, which in turn can only be added to the structure if it contains event 4.

3. Event 4 can only occur if event 2 occurs after event 1, contradicting the first point above.

This is formalised by the *modification order* relation `mo`. In WeakestMO , every two non-conflicting stores to the same location must be ordered by the `mo` relation. In this case, as events 1 and 2 are not in conflict and both store some value to $x$, there must be an `mo` edge between them.

Returning to the step at which we added these two writes to the original structure, our new structure contains the required `mo` edge stating that event 2 must happen after event 1.

$$\begin{tikzcd}
& \text{Init} \arrow[dl] \arrow[dr] & \\
1 : \mathsf{St(x, 2)} \arrow[rr, dashed, "\texttt{mo}"] & & 2 : \mathsf{St(x, 1)}
\end{tikzcd}$$

If an execution now loads the value 2 from $x$ by reading it from event 1, this creates a *from-reads* `fr` edge. From-reads edges appear from a load to any stores `mo`-after the justifying store, or any equivalent stores. Such an edge indicates that the load must preceed the store, otherwise the value it intends to load would be overwritten.

$$\begin{aligned} \texttt{fr} &= \texttt{rf}^{-1} ; \texttt{mo} \\ &= (\texttt{ew} ; \texttt{jf})^{-1} ; \texttt{mo} \end{aligned}$$

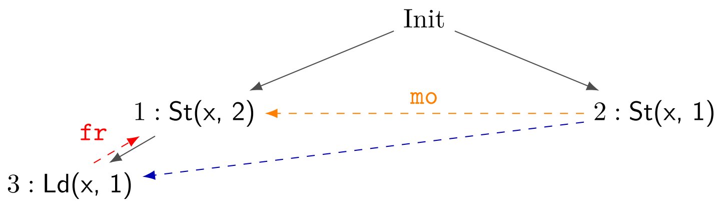
In our case, this happens when we try to add event 9:

$$\begin{tikzcd}
& \text{Init} \arrow[dl] \arrow[dr] & & \\
1 : \mathsf{St(x, 2)} \arrow[rr, dashed, "\texttt{mo}"] \arrow[drrr, dashed] & & 2 : \mathsf{St(x, 1)} \arrow[dr, shift right] & \\
& & & 9 : \mathsf{Ld(x, 2)} \arrow[ul, dashed, shift right, "\texttt{fr}"']
\end{tikzcd}$$

This indicates that event 9 cannot occur after event 2, due to the $1 \xrightarrow{\texttt{mo}} 2 \xrightarrow{\texttt{jf}} 9$ chain. This immediately causes a *coherence violation*: WeakestMO does not allow derived relational constraints to contradict program order. This is formalised in the (COH') rule, which we expand into the following:

$$\text{(COH'): } \texttt{po}; (\texttt{rf} \cup \texttt{mo} \cup \texttt{fr})^* \text{ must be irreflexive.}$$

A similar coherence violation occurs on the other side of the structure if we try to swap the order of events 1 and 2:



As we cannot arrive at the target behaviour if either event 1 or event 2 necessarily happens first, the behaviour is not permitted by the model. While the Promising model was able to first speculate that these writes would happen in one order, promise a later behaviour, and then change that order while preserving its promise, the WeakestMO model instead requires an order to be decided on upfront.

### 2.4.3 The Sewell and Pichon-Pharabod model

The model introduced in Pichon-Pharabod and Sewell (2016), never officially named in the publication but colloquially referred to as "Bubbly", is another event-structure driven model with some notion of execution. In this model, however, the entire program is interpreted to an event structure first, and executed as a second pass over the completed structure.

The event structures it generates are 4-tuples similar to those seen already:

$$(E, \leq, \sim, \lambda)$$

Where:

- $E$ is a set of unique event identifiers
- $\leq$ represents the program order relation, which in this case is transitive by default, and is used as this structure's causality relation
- $\sim$ is the conflict relation, which in this case is referred to as immediate conflict and not closed with respect to $\leq$
- $\lambda$ is a function from event identifiers to event labels

Events in this model can be reads or writes with an attached memory order, locks and unlocks, or undefined behaviour.

Rather than depicting particular behaviours, the model aims to represent optimisation passes directly. An event structure is used to represent the program text which has yet to be executed, while transition steps within the model alter the effective program text, which is the inverse of the WeakestMO model. It does not have a representation of the executed events, instead "executing" them by removing them from the structure and passing them a separate memory subsystem. The memory subsystem tracks which write events are visible to other threads, and thus what values a read event might be allowed to observe. As a result, the model itself contains no modification order or reads-from edges, making a direct comparison with other models challenging.

### Construction

As the structures represent potential future program actions, they must be constructed in their entirety before attempting to execute the program itself.

This means that any attempt to load from a location must be represented by a read event for every possible value, unlike in WeakestMO where a load event only appears if it can be justified by a store.

The construction is done by a semantic interpretation function $[\![\texttt{ss}]\!]_\mu$, where `ss` is the program text and $\mu$ is a *local register environment*, which maps thread-local registers to values. This function is defined case-by-case over the input statement.

Here we focus on loads and stores, which generate read and write events respectively. Interpreting a store creates a single write event and places it $\leq$-before all subsequently created events, while interpreting a load event creates one read event per potential value, each in immediate conflict.

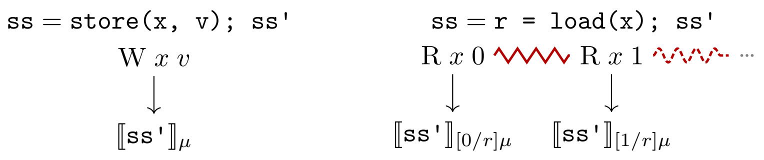


Figure 19: The interpretation of store (left) and load (right) operations into events

### Transitions

**Execution** An event $e$ can be executed if:

1. $e$ is minimal in $\leq$

2. The storage subsystem will allow an event with the label $\lambda(e)$

Executing $e$ will then remove the event itself and all events in conflict with $e$ from the structure.

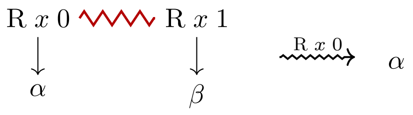


Figure 20: Example illustration of the execution of a single event

The definition of which events are permitted by the storage subsystem is complex and not summarised mathematically by any existing publication, but an English-language overview is given in Sarkar et al. (2011b). However, as many behaviours are described in terms of relative event ordering, when comparing this model to those shown prior we opt to focus more on the non-execution steps: deordering and merging.

**Deordering** The deordering operation removes a $\leq$ edge between events, potentially allowing for more than one $\leq$-minimal event to exist in the same structure.

The deordering operation requires:

1. A set $A$ of events in immediate conflict with each other, closed under conflict such that if $a' \sim a$ for $a \in A$ then $a' \in A$
2. That no events in $A$ are locks
3. A set $B$ of events constructed such that each $b_i$ in $B$ directly follows some $a_i$ in $A$ under $\leq$
4. That no event in $A$ shares a location with any event in $B$
5. That all events in $B$ share a label, and this label is not a read or unlock
6. That each $a_i$ in $A$ has only one child with the same label as events in $B$

If all conditions are met, then the structure may be rearranged as follows:

- Remove all events in $B$ from the structure except one, which we name $b$
- Remove all $\leq$ edges from events in $A$ to the remaining event $b$
- Add $\leq$ edges from $b$ to all events which follow some $b_i$ in $B$

- Restrict $\leq$, $\sim$, and $\lambda$ to the new structure by removing all edges containing some $b_i$ in $B$

Note that, in addition to $b$ gaining $\leq$ edges to all children of any $b_i$, any immediate parent of the nodes in $A$ will also become an immediate parent of $b$.

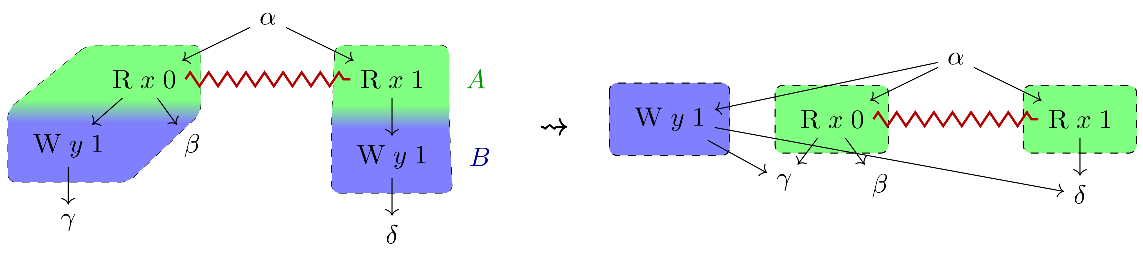


Figure 21: A deordering step in which $\leq$ edges between unrelated events are removed, but all other edges to and from these events are preserved.

This deordering operation is one potential representation of *relatedness*, where two events are considered unrelated if they satisfy the requirements. If the value to be written does not depend on the value read, then a write of the same label will appear under every possible read. If only certain values are followed by the write, then the write must depend on the read, and we are forced to execute the read before attempting the write.

In the LB litmus test, for example, the reads and writes are considered unrelated because, in both threads, an equivalent write always occurs regardless of the value read.

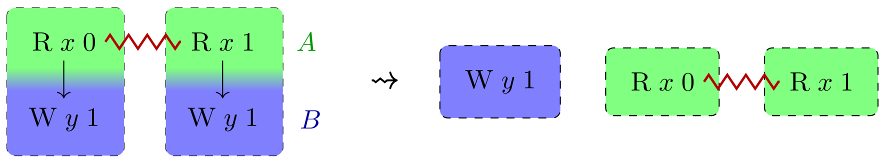


Figure 22: The deordering operation applied to the structure generated by LB

This mechanism allows read-to-write and write-to-write deordering, but read-to-read or write-to-read deordering requires a separate mechanism.

*Read deordering* requires:

1. A single event $a$, which is not a lock
2. A set $B$ of read events in immediate conflict, closed under the conflict relation
3. That every event in $B$ immediately follows $a$ in $\leq$
4. That $a$ does not share a location with the events in $B$

This allows the same removal of $\leq$ edges as the previous deordering operation.

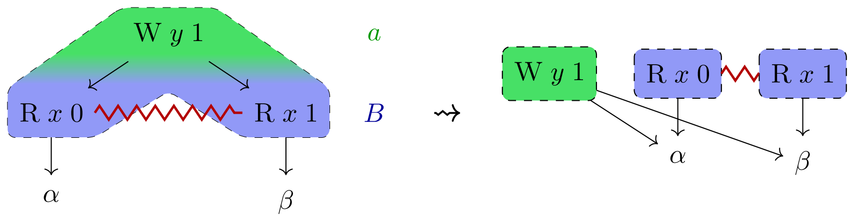


Figure 23: A read deordering step removing $\leq$ edges between two unrelated events

**Merging** Two events in a structure may be pre-emptively merged if their labels share a value and location, with a small number of additional conditions. This has a similar effect to executing one of the two events, removing it and all conflicting events from the structure, but does not invoke the memory subsystem. Intuitively, the merging operation determines that one of these events is irrelevant to the behaviour of the program given the presence of the other, such as the first of two consecutive writes to the same location, and alters the structure into one where the event has already been executed.

The model permits two separate merging operations: *forward merging*, in which a later event is removed based on the presence of an earlier event, and *backwards merging*, in which an earlier event is removed based on the presence of a later event.

**Forward Merging** Forward merging requires:

1. A pair of events $a$ and $b$ whose labels reference location $x$ and value $v$
2. That either $a$ is not a write or $b$ is not a read
3. That $a < b$
4. That there are no locks or other actions at location $x$ between $a$ and $b$ in $<$

If the above hold, then event $b$ and all events in conflict with $b$ are removed from the structure.

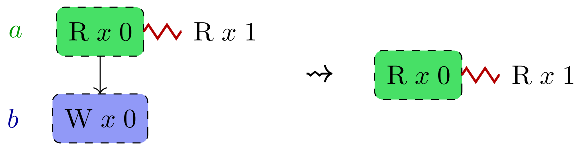


Figure 24: A forward merging operation in which a write is assumed to be redundant, as the value to be written has been read immediately beforehand and not subsequently updated.

**Backwards Merging** The other merging operation, backwards merging, requires:

1. A single write event $w$ and a set of other write events $W$, all to location $x$
2. That each configuration of the structure from $w$ onwards executes exactly one event in $W$, and that all writes in $W$ are in at least one such configuration
3. That there are no unlocks or other actions at $x$ between $w$ and any event in $W$ under $\leq$

If the above hold, the write $w$ may likewise be removed from the structure as though executed.

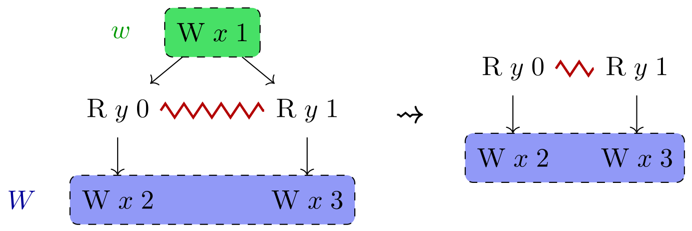


Figure 25: A backward merging operation in which a write is assumed to be redundant, as it is immediately overwritten without being read.

### Comparison with Weakest and Promising

```
1: x := 2;              4: x := 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2){     ||    6: r3 := y;
  3: y := 1;            if (r3 != 0){
}                         7: x := 3;
                        }
```

We once again run this example, reprinted above, through this new model to determine whether or not the target behaviour is permitted. The first step is the construction of the event structure, which unfortunately is prohibitively large to display in its entirety. To get around this, we display only the events of the target execution until other segments of the structure become relevant.

The initial event structure, with all syntactic orderings intact and restricted to our target execution, is as follows:

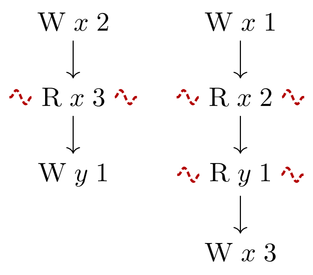


Figure 26: A truncated event structure for the example shown previously, only displaying events we intend to execute.

The sequence on the left represents our intended execution of the left-hand thread, and likewise the sequence on the right represents our intended execution of the right-hand thread.

We can first use the relatively unrestricted read deordering rule to break any $\leq$ edges between accesses of one value and reads of another. This means that the R $y$ 1 event on the right hand side becomes minimal in $\leq$: first the R $x$ 2 $<$ R $y$ 1 edge is removed, and subsequently the remaining W $x$ 1 $<$ R $y$ 1 edge.

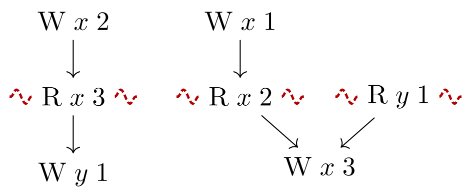


Figure 27: The same event structure following two consecutive read deordering operations.

We are unable to perform any further deordering steps on this structure. The deordering operation over writes requires that the events to be deordered are all to the same location, ruling out an attempt to break the R $x$ 2 $<$ W $x$ 3 edge. It also requires that, for read-to-write deordering, one copy of the write exists under each event in conflict with the read, ruling out an attempt to break either the R $x$ 3 $<$ W $y$ 1 edge or the R $y$ 1 $<$ W $x$ 3 edges due to the guards which check the reads' respective values before writing.

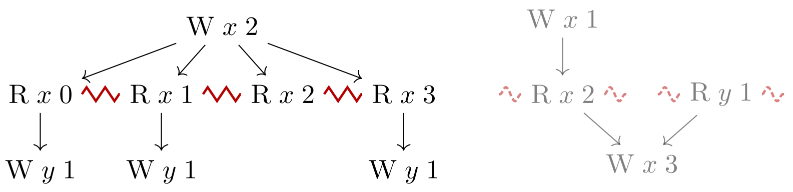


Figure 28: The lack of W $y$ 1 event below R $x$ 2 prevents any deordering here.

We might want to try backwards merging on the right-hand substructure, since the write of 1 to $x$ will be overwritten by the write of 3, but this is similarly impossible. Once again, backwards merging is only allowed if *every* configuration from the first write onwards contains a write to the same location, and there are configurations of this substructure which do not perform the write.

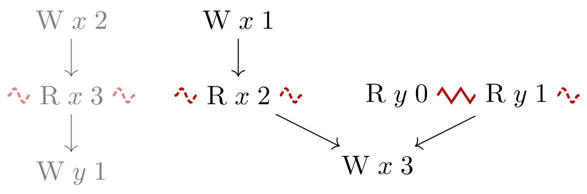


Figure 29: The configuration $\{\mathrm{W}\ x\ 1, \mathrm{R}\ x\ 2, \mathrm{R}\ y\ 0\}$ does not contain any W $x$ 3 event, preventing a backwards merge.

This means that the structure in Fig. 27 represents the final form of this structure, at least until we begin execution. However, we are unable to execute all events in this structure.

We can begin by executing both writes to $x$, and while events would typically be removed from the structure in execution, in the following diagrams we instead grey them out for a clearer execution trace. We use the purple dotted arrows to denote the order in which we executed events.

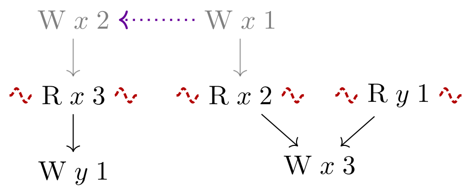


Figure 30: The structure from Fig. 27 after 2 execution steps.

We can now execute the R $x$ 2 event, having executed the write of 2 last, but this is the final event that we can execute. The only remaining events that are maximal in $\leq$ are reads of values that we have not written, and the memory subsystem will therefore not allow us to execute them.

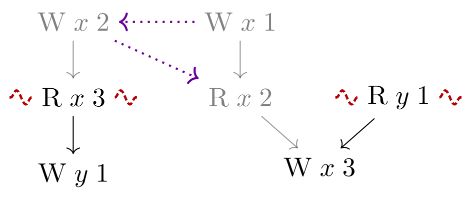


Figure 31: The maximum possible subset of these events which we can execute.

In this example, the execution phase (and hence the memory subsystem) is doing relatively little work. To determine why the model does not permit the behaviour, we must return to the deordering and merging operations that were not permitted.

To execute any other event from the remaining set, we would need to break either the R $x$ 3 $\leq$ W $y$ 1 edge or the R $y$ 1 $\leq$ W $x$ 3 edge. This remainder program is incidentally equivalent to a modification of the LB litmus test in which both threads contain guards:

```
r2 := x;             ||  r3 := y;
if (r2 = 3) {        ||  if (r3 != 0) {
    y := 1;          ||      x := 3;
}                    ||  }
```

The model maintains orderings between the reads and writes in both threads, on the grounds that we cannot know if the writes should execute without seeing the value of the read.

Analysis of the merging rules can tell us what modifications would possibly remove these edges: if the left thread contained a write of 3 to $x$ or the right thread a non-zero write to $y$, then merging would remove the reads entirely. As the whole program does not contain either of these extra writes, then the behaviour of this program as a segment of a larger program is similar to its behaviour alone: the reorderings are not permitted.

This model uses two ideas that we build on in the development of the Modular Relaxed Dependencies model in Chapter 3:

- Event relatedness is determined by analysis, not by speculation (as in the inter-execution `jf` edges of WeakestMO or the re-verification of Promising). These relatedness edges are what we refer to as dependencies.
- By analysis of a sub-program, such as the modified LB program above, we should be able to determine what a potential context would need to contain in order to modify that behaviour, and for all contexts which do not modify it, the model's representation of the sub-program should appear within its representation of the whole program. This is what we take to mean modularity.

# Chapter 3

# Modular Relaxed Dependencies

The MRD model, first introduced in Paviotti et al. (2020), uses event structures to represent the maximum set of possible executions without needing pre-justification for potential execution paths, similar to the "Bubbly" model of Sewell and Pichon-Pharabod, but then derives relations across this structure non-destructively to produce the full set of permitted executions within a single denotation. The model is shown in publication to be implementable on both x86 and ARMv8 architectures, and to retain sequential consistency in the absence of any data races.

Here we give the definitions which construct and reason about these examples, and subsequently explain how the MRD model handles the earlier programs in which the JMM and Promising gave incorrect results. The model has been submitted as a draft proposal to the International Standards Organization (Batty et al. (2019)). The version presented here differs from the published model in the order of operations performed in dependency calculation and in the presentation of the forwarding operations, discussed further in Section 3.2.

## 3.1 Labelled Event Structures

The MRD model computes relations over event structures, similar to the Weakest and WeakestMO models. It begins with a *labelled event structure*, which describes the memory accesses performed by the program without any additional information on causality or synchronisation.

The labelled event structure used by MRD is represented as:

$$(E, \sqsubseteq, \#, \lambda)$$

Where:

- $E$ is the set of all events in the structure
- $\sqsubseteq$ is the program order relation, which relates events originating from statements placed syntactically in order
- $\#$ is the conflict relation, which relates reads of different values at the same access and their descendants
- $\lambda$ is a labelling function from events to descriptions of the memory access they represent

An event's label can denote a read from global $x$ of value $v$ as $\mathrm{R}\ x\ v$, a write to a global as $\mathrm{W}\ x\ v$, an entry point to a locked section as L, or an exit point from a locked section as U. We use a shorthand syntax $(e : \mathrm{W}\ x\ v)$ to denote that event $e$ has the label $\mathrm{W}\ x\ v$:

$$(e : l) \triangleq \lambda(e) = l$$

We also derive *preserved program order*, written as $\leq$, from event labels and the $\sqsubseteq$ relation. Two events are in preserved program order if they are $\sqsubseteq$-related and either:

1. Both are accesses to the same location, or
2. At least one event is a lock or unlock.

If either of these hold for events $e_1$ and $e_2$, where $e_1 \sqsubseteq e_2$, then $e_1 \leq e_2$.

$$e_1 \leq e_2 \Leftrightarrow e_1 \sqsubseteq e_2 \wedge (((e_1 : \_\ x\ \_) \wedge (e_2 : \_\ x\ \_)) \vee \lambda(e_1) \in \{\mathrm{L}, \mathrm{U}\} \vee \lambda(e_2) \in \{L, \mathrm{U}\})$$

To keep our representation of the labelling function $\lambda$ tidy, we overload the union operator $\cup$ to combine functions over disjoint sets:

$$\text{For functions } F_1 : A \to C \text{ and } F_2 : B \to C$$
$$F_1 \cup F_2 : A \cup B \to C \triangleq \lambda\ x.\text{if } x \in A \text{ then } F_1(x) \text{ else } F_2(x)$$

We also use the functional assignment syntax $F[a \mapsto b]$:

$$F[a \mapsto b] \triangleq \lambda x.\text{if } x = a \text{ then } b \text{ else } F(x)$$

### 3.1.1 Operations on Labelled Event Structures

We interpret programs into event structures recursively, by computing a smaller sub-structure for a sub-program and expanding or combining these sub-structures. As a result, we need three operations over event structures to represent three composition operations: appending new events, placing a second structure in conflict with the first, and placing a second structure in parallel with the first.

The $\bullet$ operation appends an event $e$ to the top of an event structure. This places it before all existing events in program order, and adds its label to $\lambda$.

$$(e : \mathrm{W}\ x\ v) \bullet (E, \sqsubseteq, \#, \lambda) = (E', \sqsubseteq', \#, \lambda') \text{ where}$$
$$E' = E \cup \{e\} \qquad \sqsubseteq' = \sqsubseteq \cup \{(e, e') \mid e' \in E\} \qquad \lambda' = \lambda[e \mapsto \mathrm{W}\ x\ v]$$

The $+$ operation places two event structures in conflict. This takes the union of their respective event sets and relations, adding no new program order edges, and adds conflict edges from all events in one program to all events in the other.

$$(E_1, \sqsubseteq_1, \#_1, \lambda_1) + (E_2, \sqsubseteq_2, \#_2, \lambda_2) = (E', \sqsubseteq', \#', \lambda') \text{ where}$$
$$E' = E_1 \cup E_2 \qquad \sqsubseteq' = \sqsubseteq_1 \cup \sqsubseteq_2 \qquad \#' = \#_1 \cup \#_2 \cup (E_1 \times E_2) \qquad \lambda' = \lambda_1 \cup \lambda_2$$

The $+$ operation extends to more than two structures as the sum $\Sigma$ via repeated application.

$$\Sigma_1^n (E_n, \sqsubseteq_n, \#_n, \lambda_n) = (E_1, \sqsubseteq_1, \#_1, \lambda_1) + (E_2, \sqsubseteq_2, \#_2, \lambda_2) + ... + (E_n, \sqsubseteq_n, \#_n, \lambda_n)$$

Finally the $\times$ operation places two event structures side-by-side but not in conflict. This takes the union of all events and relations of the two structures, adding no new edges.

$$(E_1, \sqsubseteq_1, \#_1, \lambda_1) \times (E_2, \sqsubseteq_2, \#_2, \lambda_2) = (E', \sqsubseteq', \#' \lambda') \text{ where}$$
$$E' = E_1 \cup E_2 \qquad \sqsubseteq' = \sqsubseteq_1 \cup \sqsubseteq_2 \qquad \#' = \#_1 \cup \#_2 \qquad \lambda' = \lambda_1 \cup \lambda_2$$

### 3.1.2 Interpreting Programs as Event Structures

We now construct a function from program text to event structures.

Our interpretation function is written using our earlier structure composition functions, meaning it must recursively translate program fragments into structures before combining them. To do this, it uses a continuation-passing style. The continuation is invoked to store the interpretation of the rest of the program, while the statement being converted is translated into an appropriate event and appended. To ensure termination, we also include a step index variable, usually denoted as $n$, which decreases by 1 whenever a `while` loop is interpreted and immediately halts further interpretation at 0.

We begin with the rule for interpreting a write statement:

$$[\![\texttt{x := r1}]\!]_{n\ \rho\ \kappa} = (e : \mathrm{W}\ x\ \rho(r1)) \bullet \kappa(\rho)$$

The parameter $\rho$ represents the environment of local variables, which we update as needed and pass to the continuation $\kappa$. This continuation calls the semantic interpretation function recursively for the rest of the program, as shown in the rule for interpreting two consecutive statements:

$$[\![P_1; P_2]\!]_{n\ \rho\ \kappa} = [\![P_1]\!]_{n\ \rho\ (\lambda\rho'.[\![P_2]\!]_{n\ \rho'\ \kappa})}$$

Whenever we attempt to interpret a read, we avoid making any assumptions about what value we will observe. Instead, we fork our interpretation function and let each branch assume one observed value, then place all resulting structures in conflict.

$$[\![\texttt{r1 := x}]\!]_{n\ \rho\ \kappa} = \Sigma_{v \in V}(e_v : \mathrm{R}\ x\ v) \bullet \kappa(\rho[r_1 \mapsto v])$$

For control flow statements, we restrict our input language to require all `if` statements to only use local registers in the guard condition. This means that when interpreting an `if` statement, we can use the local environment $\rho$ to evaluate the guard without additional forking.

$$[\![\texttt{if}(B)\{P_1\}\{P_2\}]\!]_{n\ \rho\ \kappa} = \begin{cases} [\![P_1]\!]_{n\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \top \\ [\![P_2]\!]_{n\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \bot \end{cases}$$

We give the full semantic interpretation function in Fig. 32, and illustrate the expansion of this function over a small sample program in the proceeding.

Note that when we perform any parallel composition, the result is not $([\![P_1]\!]_{n\ \rho\ \emptyset}) \times ([\![P_2]\!]_{n\ \rho\ \emptyset})$ as one may expect. Instead, we add a pair of L and U events immediately before the fork and after the join. This has no impact on the inter-thread orderings of events between $P_1$ and $P_2$,

$$\begin{aligned}
[\![P]\!]_{0\ \rho\ \kappa} &= \underline{\emptyset} & & \\
[\![\texttt{skip}]\!]_{n\ \rho\ \kappa} &= \kappa(\rho) & [\![P_1\,;\ P_2]\!]_{n\ \rho\ \kappa} &= [\![P_1]\!]_{n\ \rho\ (\lambda\rho.[\![P_2]\!]_{n\ \rho\ \kappa})} \\
[\![\texttt{r := x}]\!]_{n\ \rho\ \kappa} &= \Sigma_{v\in V}(\text{R}\ x\ v \bullet \kappa(\rho[r\mapsto v])) & [\![\text{L}]\!]_{n\ \rho\ \kappa} &= \text{L}\bullet\kappa(\rho) \\
[\![\texttt{x :=}\ M]\!]_{n\ \rho\ \kappa} &= (\text{W}\ x\ eval(M,\rho))\bullet\kappa(\rho) & [\![\text{U}]\!]_{n\ \rho\ \kappa} &= \text{U}\bullet\kappa(\rho)
\end{aligned}$$

$$\begin{aligned}
[\![P_1 \parallel P_2]\!]_{n\ \rho\ \kappa} &= [\![\text{L};\text{U}]\!]_{n\ \rho\ \kappa'} \\
&\qquad \text{where } \kappa' = (\lambda\rho.([\![P_1]\!]_{n\ \rho\ \emptyset}) \times ([\![P_2]\!]_{n\ \rho\ \emptyset}) \star ([\![\text{L}\,;\ \text{U}]\!]_{n\ \rho\ \kappa})) \\
[\![\textbf{if}\texttt{(}B\texttt{)\{}P_1\texttt{\}\{}P_2\texttt{\}}]\!]_{n\ \rho\ \kappa} &= \begin{cases} [\![P_1]\!]_{n\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \top \\ [\![P_2]\!]_{n\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \bot \end{cases} \\
[\![\textbf{while}\texttt{(}B\texttt{)\{}P\texttt{\}}]\!]_{n\ \rho\ \kappa} &= \begin{cases} [\![P\,;\textbf{while}\texttt{(}B\texttt{)\{}P\texttt{\}}]\!]_{(n-1)\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \top \\ [\![\texttt{skip}]\!]_{n\ \rho\ \kappa} & \text{if } [\![B]\!]_\rho = \bot \end{cases}
\end{aligned}$$

Figure 32: Semantic interpretation function in full

but it does forbid any optimisations from lifting events out of a spawned thread and executing them in the parent thread.

We begin with a short program in which one thread copies the value of $x$ into $y$ and another copies $y$ into $x$. Let $\rho_0$ be an initial register environment which maps all registers to 0, and $M$ be the coherent event structure we want to create. As there are no loops in our sample program, we will keep the step index at 1 throughout.

$$M = \left[\!\!\left[ \begin{array}{ccc} & \texttt{x := 0;} & \\ & \texttt{y := 0;} & \\ \texttt{r1 := x;} & & \texttt{r2 := y;} \\ \texttt{y := r1;} & \parallel & \texttt{x := r2;} \end{array} \right]\!\!\right]_{1\ \rho_0\ (\lambda x.\emptyset)}$$

We begin by separating the program into the initial write and everything else:

$$M = [\![\texttt{x := 0}]\!]_{1\ \rho_0\ c_1}$$

Which leaves us with continuation $c_1$:

$$c_1(\rho) = \left[\!\!\left[ \begin{array}{ccc} & \texttt{y := 0;} & \\ \texttt{r1 := x;} & & \texttt{r2 := y;} \\ \texttt{y := r1;} & \parallel & \texttt{x := r2;} \end{array} \right]\!\!\right]_{1\ \rho\ \emptyset}$$

Interpreting the first write gives us the event W $x$ 0 and leaves the environment unchanged, and asserts that W $x$ 0 should be $\sqsubseteq$-before everything output by the continuation:

$$M = \boxed{\mathrm{W}\ x\ 0}$$

$$\bullet$$

$$\left[\!\!\left[ \begin{array}{lcl} & \texttt{y := 0;} & \\ \texttt{r1 := x;} & & \texttt{r2 := y;} \\ \texttt{y := r1;} & || & \texttt{x := r2;} \end{array} \right]\!\!\right]_{1\ \rho\ (\lambda x.\emptyset)}$$

We do the same again to peel off the second write, remembering to add the $\sqsubseteq$ edge from the first write:

$$\begin{tikzcd} M = & \boxed{\mathrm{W}\ x\ 0} \arrow[d] \\ & \boxed{\mathrm{W}\ y\ 0} \end{tikzcd}$$

$$\bullet$$

$$\left[\!\!\left[ \begin{array}{lcl} \texttt{r1 := x;} & & \texttt{r2 := y;} \\ \texttt{y := r1;} & || & \texttt{x := r2;} \end{array} \right]\!\!\right]_{1\ \rho\ (\lambda x.\emptyset)}$$

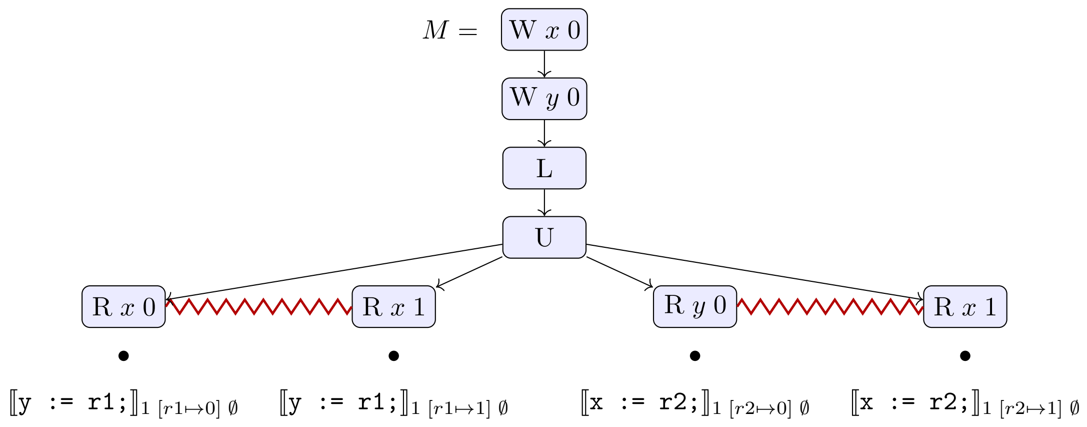
We can now separate out the two threads, recalling that we need to add a lock/unlock pair to avoid optimisation across thread boundaries and that placing two event structures in parallel does not add any relations between the two:

$$\begin{tikzcd} M = & \boxed{\mathrm{W}\ x\ 0} \arrow[d] \\ & \boxed{\mathrm{W}\ y\ 0} \arrow[d] \\ & \boxed{\mathrm{L}} \arrow[d] \\ & \boxed{\mathrm{U}} \end{tikzcd}$$

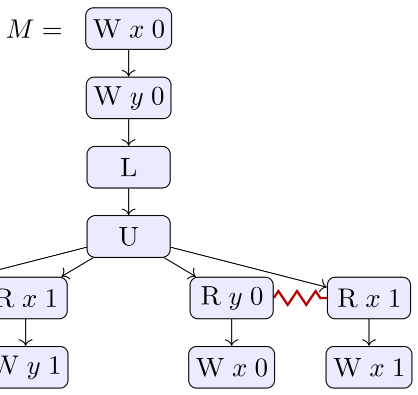
$$\bullet$$

$$\left[\!\!\left[ \begin{array}{l} \texttt{r1 := x;} \\ \texttt{y := r1;} \end{array} \right]\!\!\right]_{1\ \rho\ (\lambda x.\emptyset)} \qquad \left[\!\!\left[ \begin{array}{l} \texttt{r2 := y;} \\ \texttt{x := r2;} \end{array} \right]\!\!\right]_{1\ \rho\ (\lambda x.\emptyset)}$$

We now interpret the reads at the top of each thread. This means we fork the interpretation function, proceeding down one branch updating the register value to 0 and down the other updating it to 1.


$M =$
W $x$ 0
W $y$ 0
L
U
R $x$ 0
R $x$ 1
R $y$ 0
R $x$ 1
$[\![$y := r1;$]\!]_{1\ [r1 \mapsto 0]\ \emptyset}$
$[\![$y := r1;$]\!]_{1\ [r1 \mapsto 1]\ \emptyset}$
$[\![$x := r2;$]\!]_{1\ [r2 \mapsto 0]\ \emptyset}$
$[\![$x := r2;$]\!]_{1\ [r2 \mapsto 1]\ \emptyset}$

We use the updated environments to interpret the final writes, completing the structure:



Every path from the root node to a leaf node in this structure represents an execution of a thread. Combining a path to a leaf on the left hand side with a path to a leaf on the right hand side gives us an execution of the entire program. It doesn't tell us that this execution is *possible*, because we have entirely abstracted away the shared memory environment. These executions could observe values that were never actually written, or values that could not have been written in time to be observed.

Removing impossible executions from this set requires us to answer two questions:

1. Which write is each read observing a value from?

2. Which pairs of events definitely cannot be reordered?

We now annotate the sets of events representing executions with relations which represent these properties.

### 3.1.3 Executions

MRD's executions are augmented configurations described by the tuple:

$$X = (E, \textsc{rf}, \textsc{dp}, \textsc{lk})$$

Where:

- $E$ is a configuration, meaning a conflict-free, $\sqsubseteq$-maximal set of events.
- RF is the *reads-from* relation from writes to reads, which must only relate writes of value $v$ to location $x$ to reads of value $v$ from location $x$.
- DP is the *semantic dependency* relation, explained in detail in Section 3.2.
- LK the *lock order* relation, which is a total order over all lock/unlock events. It may not contradict program order within a thread, but is otherwise unrestricted.

As the RF and LK relations may be declared arbitrarily, provided they meet their respective constraints, and the semantic dependency relation is calculated over the whole structure, executions are determined by analysis after completion of the structure and not constructed piecewise during interpretation.

To be considered a valid behaviour of the program, a potential execution must be *complete* and *coherent*. Complete executions have a corresponding write for every read, while coherent executions cannot contain cycles caused by $\leq$,LK,DP and RF, and cannot contain RF edges which contradict program and lock order. Coherent executions must also choose a DP relation calculated using the rules for justification, given in Section 3.2, and the rules for freezing, given in Section 3.3.1.

**Definition 1** (Complete Execution)**.** *An execution $X$ is complete if and only if for all $(r : R\ x\ v)$ in $X$ there exists some $(w : W\ x\ v)$ such that $w \xrightarrow{RF}_X r$.*

**Definition 2** (Coherent Execution)**.** *An execution $X$ is coherent if and only if:*

$$(\leq \cup LK_X \cup DP_X \cup RF_X)^* \text{ is acyclic and, for } \sqsubseteq_{LK} = (\sqsubseteq \cup LK_X)^*$$

$$(w : W\ x\ v) \xrightarrow{RF}_X (r : R\ x\ v) \implies \forall (e : R\ x\ v') \in E.\ v = v' \vee \neg(w \sqsubseteq_{LK} e \sqsubseteq_{LK} r)$$

$$(w : W\ x\ v) \xrightarrow{RF}_X (r : R\ x\ v) \implies \forall (e : W\ x\ v') \in E.\ \neg(w \sqsubseteq_{LK} e \sqsubseteq_{LK} r)$$

$$(w : W\ x\ v) \in E \implies \exists DP_w \in freeze(w).\ DP_w \subseteq DP_X$$

## 3.2 Justification and Dependency

Justification relates potentially empty sets of events to a single write event:

$$E \vdash w \text{ – The set } E \text{ justifies the write } w$$

We call set $E$ a *justifying set* for $w$.

Intuitively, a set of events justify a write if the write cannot occur without the every event in the justifying set occurring first. Justifications represent partial calculations of the semantic dependency relation, and are converted into dependencies by the *freezing* operation. Evaluations of whole programs must convert all remaining justifications into dependencies and propagate this into executions in order to give a set of behaviours for the program. Fences and lock operations freeze justifications for all events after them in program order.

Justification is calculated for a particular write $w$ by beginning with the set of all read events $\sqsubseteq$-before $w$, then repeatedly removing events which are irrelevant to both data and control flow. We do this in two steps, the first being *forwarding* and the second *coproduct.* We now give the full definitions for both operations, beginning with forwarding.

### 3.2.1 Forwarding

```
1: r1 := x;          1: r1 := x;
2: r2 := x;    →     2: r2 := r1;
3: y := r2;          3: y := r2;
```

As discussed in section 2.2.1, a read may be elided during execution if it is sequentially equivalent to copying a previously observed value.

Both line 1 and line 2 in the leftmost program above are before line 3 in program order, but the optimised form on the right makes it clear that line 2 need not generate a read operation during execution. In any execution where lines 1 and 2 appear to observe the same value, we therefore assume that the memory access at line 2 may have been replaced by copying the value stored locally. This is known as *load forwarding.*

We denote a load forwarding operation via the $\xrightarrow{LF}$ arrow, where $r_1 \xrightarrow{LF} r_2$ indicates that the read $r_2$ is actually taking a copy of the local value returned by read $r_1$:

$$(r_1 : \mathrm{R}\ x\ v) \xrightarrow{LF} (r_2 : \mathrm{R}\ x\ v) \text{ if } \nexists e.\ r_1 \leq e \leq r_2$$

A read may be also elided if it is fetching a value written immediately before it:

```
1: x := 1;          1: x := 1;
2: r1 := x;    →    2: r1 := 1;
3: y := r1;         3: y := r1;
```

If we observe the value 1 at line 2, we may be observing the result of a constant propagation instead of a load operation. This is known as *store forwarding*.

The $\xrightarrow{SF}$ arrow describes store forwarding, where $w \xrightarrow{SF} r$ denotes that the read $r$ is observing a cached or propagated value created by write $w$:

$$(w : \mathrm{W}\ x\ v) \xrightarrow{SF} (r : \mathrm{R}\ x\ v) \text{ if } \nexists e.\ w \leq e \leq r$$

These two combine into the general forwarding case $\xrightarrow{F}$, which takes their transitive closure.

**Definition 3** (Forwarding)**.** *If* $r_1 \xrightarrow{LF} r_2$ *then* $r_1 \xrightarrow{F} r_2$
*If* $w \xrightarrow{SF} r$ *then* $w \xrightarrow{F} r$
*If* $e_1 \xrightarrow{F} e_2$ *and* $e_2 \xrightarrow{F} e_3$ *then* $e_1 \xrightarrow{F} e_3$

For any given write $w$, the set of reads which occur $\sqsubseteq$-before it is written $R(w)$, while the set of all possible forwardings applied to $R(w)$ is written $F(w)$. If there is a lock or unlock event $\sqsubseteq$-before $w$, then all events $\sqsubseteq$-before it are placed in a “forbidden” set and ignored, as we do not allow optimisations across locks or unlocks.

**Definition 4** (Forwarding Sets)**.**

$$\begin{aligned}
forbid(w) &\triangleq \{e \mid \exists e'.\ \lambda(e') \in \{L, U\} \wedge e \sqsubseteq e' \sqsubseteq w\} \\
R(w) &\triangleq \{r \mid (r : R\ x\ v) \sqsubseteq w \wedge r \notin forbid(w)\} \\
F(w) &\triangleq \{S \mid e \in R \setminus S \Rightarrow \exists e'.\ e' \xrightarrow{F} e \wedge e' \in S \wedge e' \notin forbid(w)\}
\end{aligned}$$

### 3.2.2 Coproduct

The coproduct operation removes reads from a justifying set for $w$ if their value does not determine whether or not a write identical to $w$ executes. We discuss the calculation in stages, beginning with the most simple case and expanding the definition until it can handle the most complex case.

### Label Isomorphism

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

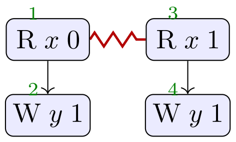


Figure 33: A program containing a write independent of a read, and its event structure.

If we take the two sub-structures just below the read, we can see that they are similar in appearance. They differ in the event identifiers, but each event in one has an equivalent in the other. Events 2 and 4 both have label W $y$ 1, meaning that any execution of this program will perform that write regardless of the result of the read. The coproduct operation is done per-read, removing a set of conflicting reads from the justifying sets of a set of conflicting writes.

For given read $r$ where $\{r\} \vdash w$, then if:

letting $\#^1 = \# \setminus (\#;\sqsubseteq)$

we have $R = \{r_v \mid r_v \#^1 r\}$

and for each $r_v \in R$ we can find $w_v \sim w$ after $r_v$

Then we can remove this read from the justifying set, giving us $\emptyset \vdash w$

### 3.2.3 Expanding Coproduct to Sets

```
1: r1 := x;
2: r2 := y;
3: z := r2;
```

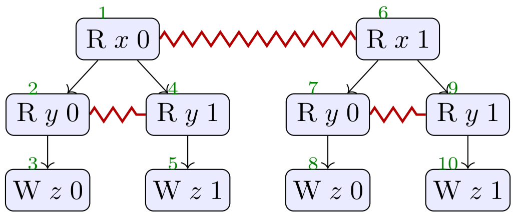


Figure 34: A program containing a write which is dependent on one read and independent of another, and its event structure.

Having coproduct defined for single reads is only useful if all of our justifying sets are singletons. In this program, the write on line 3 is affected by the value read on line 2, but not on line 1.

If we look at the initial justifying set for event 3, which in this case is the set of all prior reads, we find $\{(2 : \mathrm{R}\ y\ 0), (1 : \mathrm{R}\ x\ 0)\} \vdash 3$. For event 8, we have $\{(7 : \mathrm{R}\ y\ 0), (6 : \mathrm{R}\ x\ 1)\} \vdash 8$.

```
r1 := x;
r2 := y;
z := r2;
```

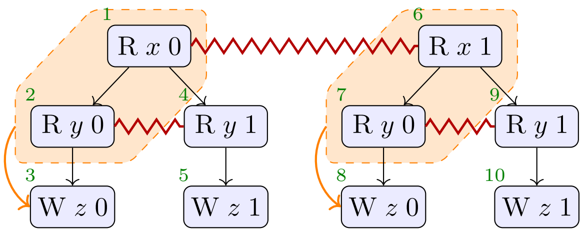


Figure 35: The only available justifying sets for events 3 and 8, highlighted in orange

If we truncate these sets at events 1 and 6 respectively, they are label isomorphic, containing a single read from $y$ which returns the value 0. If we run a coproduct operation for event 3 at event 1, it should succeed, because the rest of the justifying set for event 3 is equivalent to the rest of a justifying set for event 8.

For given read $r$ where $\mathbf{C} \cup \{r\} \vdash w$, then if:

letting $\#^1 = \# \setminus (\#; \sqsubseteq)$

we have $R = \{r_v \mid r_v \#^1 r\}$

and for each $r_v \in R$ we can find $w_v \sim w$ after $r_v$

and $\{\mathbf{C_v} \cup \mathbf{r_v}\} \vdash \mathbf{w_v}$ where $\mathbf{C_v} \sim \mathbf{C}$

Then we can remove this read from the justifying set, giving us $C \vdash w$

This label isomorphism is useful for spotting similarities, but not strong enough to determine when two sets are equivalent enough to remove a read, as we now illustrate.

### Order Preservation

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

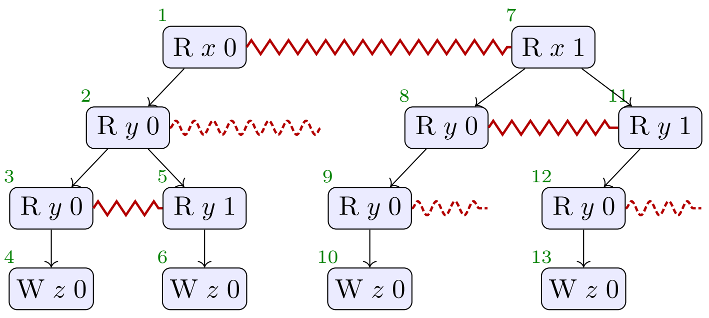
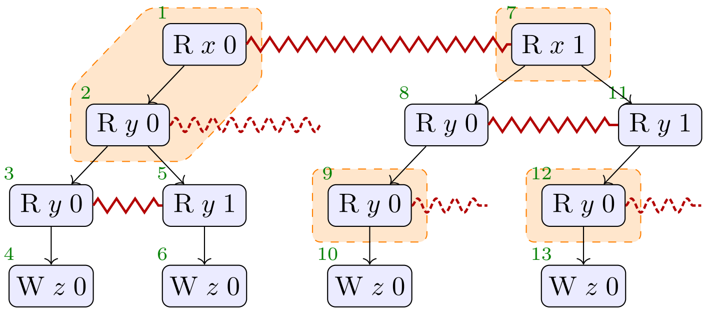


Figure 36: A program where the data dependency created depends on a prior control dependency, and part of its event structure

In the program shown in Fig. 36, though both branches of control flow result in copying a value seen at $y$, one branch reads twice and keeps the first while the other branch reads twice and keeps the second. We have elided the paths leading to a write of 1 on its event structure, indicated by the dashed arrows. Our above guidelines would allow us to remove one pair of reads from the justifying sets for each write:



The justifying sets are clearly label isomorphic up to the top-level read:

- $\{(1 : \mathrm{R}\ x\ 0), (2 : \mathrm{R}\ y\ 0)\} \vdash (4 : \mathrm{W}\ z\ 0)$
- $\{(1 : \mathrm{R}\ x\ 0), (2 : \mathrm{R}\ y\ 0)\} \vdash (6 : \mathrm{W}\ z\ 0)$
- $\{(7 : \mathrm{R}\ x\ 1), (9 : \mathrm{R}\ y\ 0)\} \vdash (10 : \mathrm{W}\ z\ 0)$
- $\{(7 : \mathrm{R}\ x\ 1), (12 : \mathrm{R}\ y\ 0)\} \vdash (13 : \mathrm{W}\ z\ 0)$

However, we shouldn't be able to remove it. Doing so would, in general, risk a style of coherence violation where two threads disagree on modification order - if two different values are observed at $y$, then the first value and the second value should not be swapped. Ignoring the condition on

$x$ and executing a single branch regardless of outcome is effectively swapping the order of these two reads, therefore potentially contradicting modification order. As a result, when determining the similarity of substructures beneath a read, we use a *label and* $\leq$*-preserving bijection* in place of label similarity.

For given read $r$ where $C \cup \{r\} \vdash w$, then if:

letting $\#^1 = \# \setminus (\#;\sqsubseteq)$

we have $R = \{r_v \mid r_v \#^1 r\}$

and for each $r_v \in R$ we can find $w_v \sim w$ after $r_v$

and $\{C_v \cup r_v\} \vdash w_v$

**if there is a label and $\leq$-preserving bijection $\approx$ such that** $C_v \approx C$

Then we can remove this read from the justifying set, giving us $C \vdash w$

### Upwards Closure

There is one final context in which an event cannot be removed from a justifying set, which can occur despite this event not impacting the write directly, which is when we've used another write in a store forwarding operation. We need to be aware of the potential ordering constraints of that write, because the validity of our justifying set is now sensitive to when it occurs.

```
1: r1 := z;                        ||
2: if (r1 = 1) {x := 1;}           ||   5: r3 := y;
3: r2 := x;                        ||   6: if (r3 = 1) {z := 1};
4: if (r2 = 1) {y := 1};           ||
```

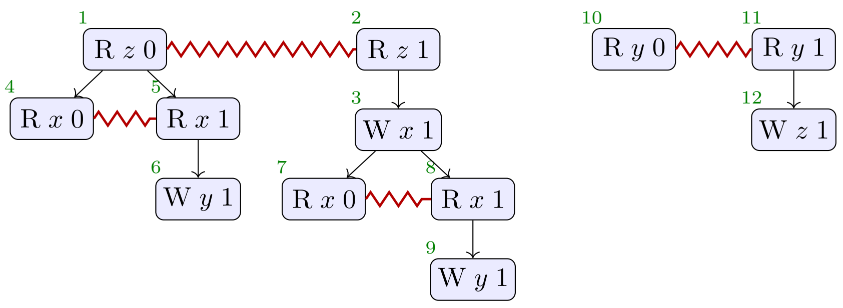


Figure 37: A program with a pair of hidden non-reorderable statements.

In the program in Fig. 37, we may be tempted by analysis of the event structure to declare any write of 1 to $y$ to be independent of the value read from $z$. However, looking at the program text, it would be highly counterintuitive to consider them unrelated. The guard on line 4 may be guaranteed to pass if the guard on line 2 passes, but that guard requires the read on line 1 to have returned.

We first step through the structure using the rules declared so far, determining the justifying sets for all writes of 1 to $y$. Initially, we have:

- $\{1, 5\} \vdash 6$
- $\{2, 8\} \vdash 9$

Since events 5 and 8 are label isomorphic, this implies that we should be able to remove events 1 and 2 from these justifications. However, these cases aren't as similar as their justifying sets make them appear, since event 8 may be forwarded away and event 5 may not.

To avoid these errors, we introduce *upward closure.* The upward closure of a justifying set $C$ is defined as the set of conflict-free extensions of $C$ such that if some event $e_2$ in the structure appears $\leq$-before some event $e_1$ in $C$, then $e_2$ must appear in the extension.

**Definition 5** (Upwards Closure)**.**

$$\uparrow\!C \triangleq \{S \mid C \subseteq S \wedge (\# \cap (S \times S) = \emptyset) \wedge \forall e_1, e_2.\ e_2 \in S \wedge e_1 \leq e_2 \Rightarrow e_1 \in S\}$$

When determining if two justifying sets are similar enough for coproduct, we require that the upwards closures of both are equivalent in both labelling and ordering.

### Full Definition

**Definition 6** (Coproduct)**.** *For given read $r$ where $C \cup \{r\} \vdash w$, then if:*

*letting $\#^1 = \# \setminus (\#; \sqsubseteq)$*

*we have $R = \{r_v \mid r_v \#^1 r\}$*

*and for each $r_v \in R$ we can find $w_v \sim w$ after $r_v$*

*and $\{C_v \cup r_v\} \vdash w_v$ where $C_v \sim C$*

*for each $C_v$ there is some $D_v \in \uparrow\!C_v$*

*and a label and $\leq$-preserving bijection $\approx$ such that $D_v \approx D$*

*Then we can remove this read from the justifying set, giving us $D \vdash w$.*

*The set of all such $D$ is given by $\sum_r w$.*

We now need to elaborate on the justification-up-to function, recursively applying this definition to every read in the initial justifying set for a write.

### Justification Up-To

As events within a single execution are totally ordered by program order, we run the justification-up-to function over a chain. Give an chain $C = (E, \sqsubseteq)$ of events $E$ totally ordered by $\sqsubseteq$, let the notation $e \bullet C$ be a constructor which adds a single $\sqsubseteq$-maximal event $e$:

$$e \bullet C \triangleq (\{e\} \cup E, (\{e\} \times E) \cup \sqsubseteq)$$

Justification-up-to can then be recursively defined over this constructor:

**Definition 7** (Justification-up-to)**.**

$$\begin{aligned} \overset{\emptyset}{\vdash w} &= F(w) \\ \overset{e \bullet C}{\vdash w} &= \begin{cases} \overset{C}{\vdash w} \ \cup \sum_e w & \textit{if } (e : R\ x\ v) \\ \overset{C}{\vdash w} & \textit{otherwise} \end{cases} \end{aligned}$$

The function evaluates as follows:

1. Remove the top-level event at each call until the set is empty, at which point it returns $F(w)$.
2. Return to the site of the previous call. If the top-level event is a read, attempt to coproduct at it using all established justifying sets so far and return the result. If it is not, return the existing justifying-up-to sets.
3. Repeat step 2 until the complete chain has been reassembled, then finally return all generated sets.

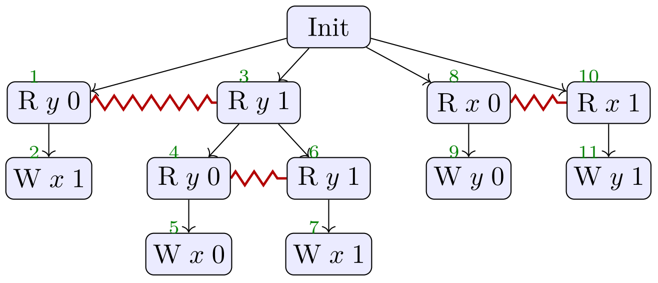
This produces the complete set of potential justification edges for $w$. In the absence of freeze operations, the total set of justifications for an event $w$ is given by $\overset{\{e \mid e \sqsubseteq w\}}{\vdash w}$.

We now run a few examples using the justification calculation directly. While the freeze operation is necessary to produce a final denotation, for these examples it suffices to use the intuition that freezing converts one justifying set per write into a dependency relation, removing all writes from the set to ensure that dependencies are only from read events to write events. We cover freezing more formally in Section 3.3.1.

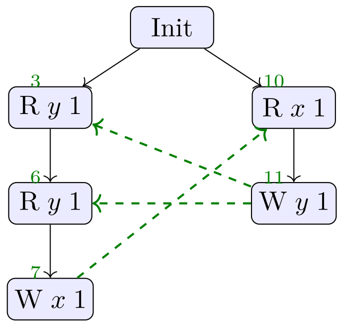
## 3.3 Worked Examples

**Avoiding The JMM's Optimisation Problem**

```
1: r1 := y;
if (r1 = 1) {
  2: r2 := y;      ||   5: r3 := x;
  3: x := r2;      ||   6: y := r3;
} else {
  4: x := 1;
}
```



The JMM example uses the fact that MRD removes repeated loads of the same value from the same location from semantic dependency, only taking the earliest such read as the "true" predecessor of later writes. This allows us to treat a read as having a predetermined value, or at least a value which isn't taken from shared memory, depending on the context in which it appears. Once again, we construct the event set and reads-from relation which corresponds to the permitted outcome $\texttt{r1} = \texttt{r2} = 1$ as shown in Fig. 38.



$X = \{3, 6, 7, 10, 11\}$

$\textsc{rf} = \{(7, 10), (11, 3), (11, 6)\}$

$\textsc{dp} = ?$

$\textsc{lk} = \emptyset$

Figure 38: Our target execution representing the optimisation performed by HotSpot

This execution would be forbidden by the combination of dependency edges $10 \xrightarrow{\textsc{dp}} 11$ and either $6 \xrightarrow{\textsc{dp}} 7$ or $3 \xrightarrow{\textsc{dp}} 7$.

We can see from the event structure that $10 \xrightarrow{\textsc{dp}} 11$, as there are no intervening events and each conflicting read leads to a write of a different value, so we focus on the justifying sets for event 7. The pre-justification $\{3,6\}$ has a load forwarding available – $6 \xrightarrow{LF} 3$. Our initial justifications for event 7 are therefore $\{3\} \vdash 7$ and $\{3,6\} \vdash 7$.

We then expand the justification-up-to function into

$$\begin{aligned}
\overset{\{0,3,6\}}{\vdash 7} &= \overset{\{3,6\}}{\vdash 7} \\
&= \overset{\{6\}}{\vdash 7} \cup \sum_{3} 7 \\
&= \overset{\emptyset}{\vdash 7} \cup \sum_{6} 7 \cup \sum_{3} 7 \\
&= \{\{3\},\{3,6\}\} \cup \emptyset \cup \sum_{3} 7
\end{aligned}$$

There is no event under event 4 isomorphic to event 7, thus $\sum_6 7 = \emptyset$ and we concern ourselves instead with $\sum_3 7$.

Our set of conflicting events is the set $\{1,3\}$, and event 2 below event 1 is label similar to event 7 below event 3. Following Definition 6 and choosing the smaller justification $\{3\}$ for event 7, this gives:

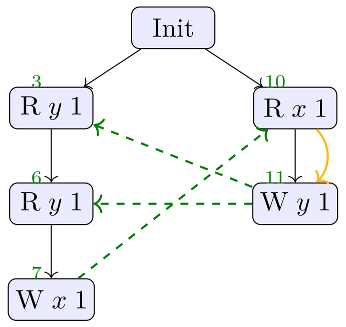
$$\begin{aligned}
R &= \{1,3\} \\
\mathbb{S} &= \{\{(1,2),(3,7)\}\} \\
C_0 &\vdash 2 \qquad C_1 = \{3\} \\
D_1 &\in \{\emptyset, \{6\}, \{0\}, \{0,6\}\} \\
\sum_{3} 7 &= \{D \mid C_0 \vdash 2 \wedge D_0 \in\uparrow (C_0 \setminus \{1\}) \wedge D_0 \sim D_1\}
\end{aligned}$$

If we can find $C_0$ such that its upward closure is label similar to some choice of $D_1$, then we can remove 3 from the justification for event 7.

To find $C_0$, we calculate the justifications for event 2, beginning with the pre-justification $\{1\}$. As the initialisation writes can create a store forwarding for event 1, the set $F(2) = \{\{0\},\{1\}\}$. We take the latter to be $C_0$ as it is a valid justification.

$$
\begin{aligned}
R &= \{1,3\} \\
\mathbb{S} &= \{\{(1,2),(3,7)\}\} \\
C_0 = \{1\} &\qquad C_1 = \{3\} \\
D_0 \in \{\emptyset,\{0\}\} &\qquad D_1 \in \{\emptyset,\{6\},\{0\},\{0,6\}\} \\
\sum_3 7 &= \{D \mid D_0 \sim D_1 \sim D\}
\end{aligned}
$$

This gives us $\emptyset \vdash 2$ and $\emptyset \vdash 7$, and additionally $\{0\} \vdash 2$ and $\{0\} \vdash 7$. Taking either of these breaks the $3 \xrightarrow{\text{DP}} 7$ edge and permits the execution. We illustrate this with the former, making event 7 independent.



$$
\begin{aligned}
X &= \{3,6,7,10,11\} \\
\text{RF} &= \{(7,10),(11,3),(11,6)\} \\
\text{DP} &= \{(10,11)\} \\
\text{LK} &= \emptyset
\end{aligned}
$$

**Avoiding Promising's OOTA Problem**

```
1: x := 2;            4: x := 1;
2: r1 := x;           5: r2 := x;
if (r1 != 2){    ||   6: r3 := y;
  3: y := 1;          if (r3 != 0){
}                       7: x := 3;
                      }
```

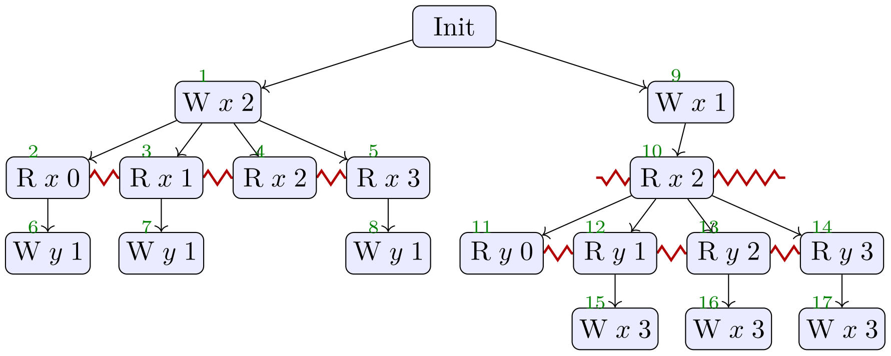
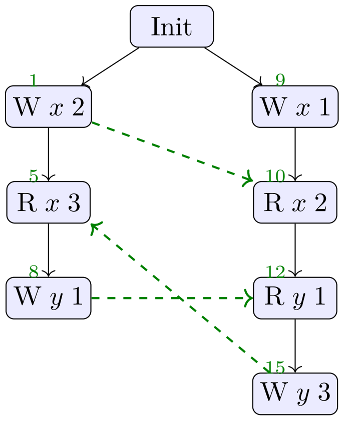


Figure 39: Program text and the truncated event structure for the OOTA7 program, where the unprinted structures in conflict with event 10 are isomorphic.

We begin with the event structure created by the program text, as shown in Fig. 3.3. The forbidden outcome $\texttt{r1} = 3 \wedge \texttt{r2} = 2 \wedge \texttt{r3} = 1$ corresponds to an execution containing events 1, 5, 8, 9, 10, 12, and 15. For this to be a complete execution, each read event in this set must also be in the RF relation. Drawing the only possible candidate results in the target execution in Fig. 3.3.



$X = \{1, 5, 8, 9, 10, 12, 15\}$

$\text{RF} = \{(1, 10), (8, 12), (15, 5)\}$

$\text{DP} = ?$

$\text{LK} = \emptyset$

This execution is impossible if we have dependencies from events 5 to 8 and from 12 to 15, as this would cause a cycle. We begin by calculating the justification of event 8.

The pre-justification $\{5\}$ has no forwardings available, so it becomes our only initial justifying set. The final set of possible justifications for event 8 is given by

$$\begin{aligned} \overset{\{0,1,5\}}{\vdash} 8 \quad = \quad \overset{\{1,5\}}{\vdash} 8 \quad = \quad \overset{\{5\}}{\vdash} 8 \\ = \overset{\emptyset}{\vdash} 8 \quad \cup \sum_{5} 8 \\ = \{\{5\}\} \cup \sum_{5} 8 \end{aligned}$$

As there is no isomorphic write under event 4, $\sum_5 8 = \emptyset$ and the only possible justifying set for event 8 is $\{5\}$.

The pre-justification for event 15 is the set $\{10, 12\}$, likewise with no available forwardings.

$$\begin{aligned} \overset{\{0,9,10,12\}}{\vdash} 15 \quad = \quad \overset{\{9,10,12\}}{\vdash} 8 \quad = \quad \overset{\{10,12\}}{\vdash} 8 \\ = \overset{\{10\}}{\vdash} 15 \quad \cup \sum_{12} 15 \\ = \overset{\emptyset}{\vdash} 15 \quad \cup \sum_{10} 15 \cup \sum_{12} 15 \\ = \{\{10,12\}\} \cup \sum_{10} 15 \cup \sum_{12} 15 \end{aligned}$$

We cannot perform a coproduct operation at event 12 as we cannot find an isomorphic write under event 11, thus $\sum_{12} 15 = \emptyset$, but we can coproduct at event 10 as all conflicting structures are isomorphic. This gives us $\sum_{10} 15 = \{\{12\}\}$, giving the final set of justifying sets $\{\{12\}, \{10, 12\}\}$.

When we freeze these justifications, we cannot avoid $5 \xrightarrow{\text{DP}} 8$ or $12 \xrightarrow{\text{DP}} 15$. This means execution $X$ will always be discarded and the behaviour is forbidden.

### 3.3.1 Freezing

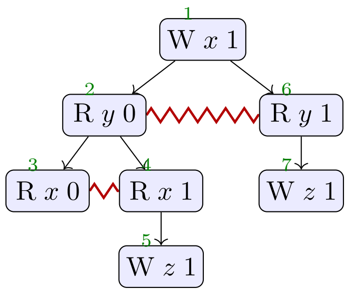
The freezing operation converts justifying sets, which denote hypothetical dependencies that may be modified by the addition of new events to the program, into immutable dependency edges. A freeze occurs at every point in the program across which optimisation should be forbidden, as these are the points at which we know that further operations on the event structure cannot alter justification edges. This includes the top of the structure when interpreting a program as a "whole program" which will not be further modified.

Freezing a write $w$ will give a set of potential dependency relations, and the coherence requirement of Section 3.1.3 enforces that any execution chooses a single dependency for $w$ from this set. The calculation $freeze(w)$ first takes the set of "freezing points" $FP$, which are locks and unlocks, which occur $\sqsubseteq$-before $w$. If this set is empty, then it calculates the justifications for $w$ up to the top of the program and converts each justifying set $C$ into a set of pairs $\{(e, w) \mid e \in C\}$, which can be treated as a fragment of a relation. If the set $FP$ is non-empty, it terminates the justification calculation at the $\sqsubseteq$-latest freezing point.

$$
\begin{aligned}
FP(w) &\triangleq \{e \mid \lambda(e) \in \{\mathrm{L}, \mathrm{U}\} \wedge e \sqsubseteq w\} \\
S(w) &\triangleq \begin{cases} \{e \mid e \sqsubseteq w\} \vdash w & \text{if } FP = \emptyset \\ \{e \mid f \sqsubseteq e \sqsubseteq w\} \vdash w \quad \text{where } f \text{ is } \sqsubseteq\text{-maximal in } FP & \text{otherwise} \end{cases} \\
freeze(w) &\triangleq \{\{(e, w) \mid e \in S\} \mid S \in S(w)\}
\end{aligned}
$$

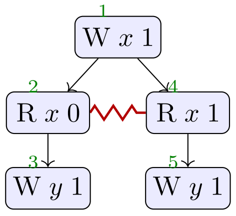
### 3.3.2 Updates and Rationale

In the original version of MRD, both forwarding and coproduct were done during the initial creation of the structure. Each execution had its own justification relation and every appended event would modify the relation for each execution, whether by adding a read to the set, successfully performing a coproduct operation, or causing a forwarding operation. The reason for the modification is the *forward-after-coproduct* bug, illustrated via the structure:



Placing events into justifying sets for event 5 as they are appended first creates the justification $\{4\} \vdash 5$, at which point we cannot coproduct due to the empty structure below event 3, then $\{2, 4\} \vdash 5$. If we try to coproduct now, noting the similarity between events 5 and 7, the operation fails due to the absence of any read from $x$ before event 7. When event 1 is appended, we are left with justifications $\{1,2\} \vdash 5$ and $\{6\} \vdash 7$, which freeze into $2 \xrightarrow{\text{DP}} 5$ and $6 \xrightarrow{\text{DP}} 7$, ultimately creating two singleton dependency edges from directly conflicting reads to label similar writes.

The version presented here also makes forwarding operations optional, to avoid the addition of spurious dependency edges during upward closure in structures such as this:



If we cannot avoid using the $1 \xrightarrow{F} 4$ edge, then the only initial justification for event 5 is $\{1\} \vdash 5$ and we cannot coproduct to remove event 2 from $\{2\} \vdash 3$ without adding event 1 via upwards closure. This does not correspond intuitively to any optimisation of the program, as there should not be a dependency edge which corresponds to a store-and-copy operation replacing a subsequently ignored read.

Finally, for representational simplicity, the description of justification and preserved program order as elements in a third-layer tuple has been replaced by descriptions of their derivations, as they can be inferred from other edges constructed during interpretation. This brings the structure closer to that used by the symbolic variant.

## 3.4 Advantages Over Trace-Based Models

The problem of weak memory concurrency arose initially due to a divergence between how we think about program execution when writing a semantics and how we think about program execution when we create compilers and hardware. The traditional output of a semantic interpretation function $[\![P]\!]$ is an object representing only memory state, or even a single value in a purely functional language. The output itself does not care for the structure inside the program which creates it – this exists only as the sequence of derivation rules which we use as a bridge between the program text and the output.

This causes these models to view the program text as an immutable, railroaded execution regardless of what we would call semantic dependency. Our reasoning system forces us into a chain of pre- and post-conditions, which in turns forces our interpretation of the program to be fully linear. When concentrating on implementing these programs, however, it is more efficient to view them as a minimal set of such dependency edges. The programmer can be assumed to have no particular intention about the ordering of statements unless otherwise indicated.

When we execute a program on multiple threads, however, the change in program state over time becomes critically important to its output. If we concern ourselves only with the output, any modifications to the change in program state must be propagated to the reasoning system. If we represent behaviour over time *as* the output of the semantics, then any tweaks to the ordering of statements can be represented as modifications to the output and leave the process of reasoning about the program can remain minimally disturbed. We illustrate this in Chapter 5, where we propagate the statement ordering information from MRD into a program logic which functions as a slight weakening of a standard Hoare logic.

Trace-based models which begin at a sequential interpretation of the program and make a series of step-by-step modifications are essentially approaching the problem from the opposite angle. MRD begins with zero ordering, allowing any hypothetical memory state to be visible at any point in the interpretation of the program, and then imposes the minimal possible ordering and discards any impossible executions. The most notable difference between these approaches is that the ordering of statements which MRD represents as semantic dependency cannot be modified by parallel composition. If two events are judged to be non-reorderable in MRD, then while parallel composition may make those events observable in a complete execution it cannot cause them to lose their ordering. If each event needs a witness within an execution before its orders can be calculated, as in trace-based models, then its orderings are only calculable given the parallel composition of a particular thread.

The difficulty of denotational semantics which represent state over time, however, is that they do not enjoy the particular property of equal denotations representing equivalent outputs. Equality is too tight a relation for a behaviour-over-time structure, so in its place we introduce a new relation over these structures.

# Chapter 4

# A Refinement Relation For MRD Structures

While the dependency calculation can reason about optimisations in terms of data and control flow, it cannot automatically determine every optimisation a compiler writer, or even a programmer themselves, may want to apply. Dependency analysis uses a very constrained idea of "correctness" with respect to program transformations, treating every possible access as being significant to the program. When considering a program as a function from input to output however, certain accesses may be irrelevant, or even detrimental, to its correctness.

Consider two implementations of a stack, one of which tracks its own size and one of which does not: if the length variable is never used in the rest of the program, either implementation may be used interchangeably as long as all other updates are identical. Alternatively, if an implementation accidentally updates the length variable twice, it should be permissible to remove the extra write operation on the grounds that, while a concurrent process could detect the second write, it would never be certain which implementation was running if it only detected a single write. The behaviours of the one-write implementation would be identical to those of the two-write implementation in every case other than the detection of the second write.

To capture the idea of removing irrelevant or erroneous behaviours while keeping all others, we introduce *refinement*. We begin in Section 4.2 by giving a non-modular definition of refinement, which we call *observational refinement*. Then, in Section 4.3, we make this definition modular with respect to program composition, establishing a definition of refinement which still holds when both programs are placed in arbitrary contexts. This would ordinarily require a proof that the refinement still holds for the program fragments under all possible local environments, so in Section 4.5 we introduce a notation for abstracting away this environment and proving that the refinement relation must hold regardless of local state. Finally, in Section 4.7 we give an overview of how refinement reflects some of the program transformations described in the Java Causality Test Cases of Pugh (2004), and does not reflect others.

## 4.1 Preliminaries

Throughout, whenever a program $P_1$ has a superset of the behaviours of program $P_2$, we describe $P_1$ as the *source program* and $P_2$ as the *target program.* This reflects the optimisation-based motivation for refinement: we want to claim that the target program refines the source progra.

We take as a running example the sequence of program transformations described by Ševčík and Aspinall (2008) in the discussion of the aforementioned Hotspot optimisation, where source program $T_1$ is optimised into target program $T_2$, and then source $T_2$ into target $T_2$:

```
     T3                      T2               T1
-------------------------------------------------------
r1 := y;
if (r1 = 1){
    r2 := y;             r1 := y;         x := 1;
    x := r2;     ⟶      x := 1;    ⟶    r1 := y;
} else {
    x := 1;
}
```

The programs $T_1$ and $T_2$ are indistinguishable in any context, but the event structures of $[\![T_1]\!]_{n\ \rho\ \kappa}$ and $[\![T_2]\!]_{n\ \rho\ \kappa}$, shown in Fig. 40, are different.

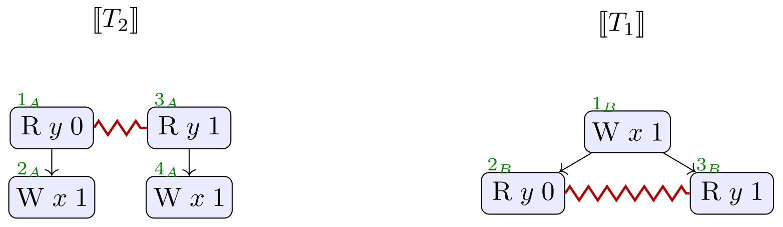


Figure 40: Event structures for equivalent programs $T_1$ and $T_2$.

## 4.2 Observational Refinement

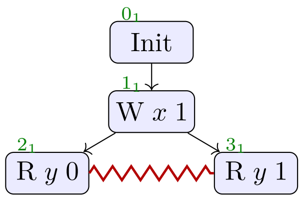
We can say that one program emulates another if it produces the same outputs while running. An output can be modelled as a write to a specific location, whether this is the terminal, a file handler, or a user interface element. By defining a particular location as observable, we can define a behaviour as an ordered set of writes to that observable location.

As MRD does not represent memory as an object, the state of a location over time can be understood as the set of writes from which we could draw an RF edge, and the order in which older writes become invisible. A read cannot observe a write $w_1$ if there is some other write $w_2$ to the same location such that, by the time the read happens, $w_2$ must have happened after $w_1$, overwriting it. Writes can only be ordered with respect to each other by $\leq$ and LK edges, so we wrap these into the derived *happens-before* relation $\xrightarrow{\text{HB}} = (\leq \cup \xrightarrow{\text{LK}})^*$.

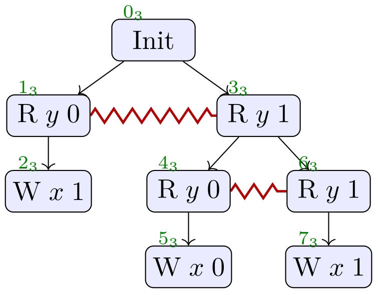
If all *observable writes* in the source program have equivalents with the same labels in the target program, and those equivalents follow the same $\xrightarrow{hb}$ order, then we can claim that the two programs have the same behaviour with respect to the chosen definition of "observable".

**Definition 8** (Observational Refinement)**.** *For all complete programs $P_1$ and $P_2$, step counter values $n$, register environments $\rho$ and continuations $\kappa$, we say $P_1$* is an observational refinement of *$P_2$, which we write $[\![P_1]\!]_{n\ \rho\ \kappa} \preccurlyeq_{\text{OBS}} [\![P_2]\!]_{n\ \rho\ \kappa}$, if and only if for every complete execution $X_1 \in [\![P_1]\!]_{n\ \rho\ \kappa}$, there exists some execution $X_2 \in [\![P_2]\!]_{n\ \rho\ \kappa}$ and label-preserving bijection over observable events $\sim$ such that $E_1|_{\text{OBS}} \sim E_2|_{\text{OBS}}$ and for all $e_1 \in E_1|_{\text{OBS}}$ and $e_2 \in E_2|_{\text{OBS}}$, if $e_1 \sim e_2$ then $\{e \mid e \xrightarrow{HB_{X_2}} e_2 \wedge e \in \textsc{Obs}\} \precsim \{e \mid e \xrightarrow{HB_{X_1}} e_1 \wedge e \in \textsc{Obs}\}$ and $\{e \mid e_2 \xrightarrow{HB_{X_2}} e \wedge e \in \textsc{Obs}\} \precsim \{e \mid e_1 \xrightarrow{HB_{X_1}} e \wedge e \in \textsc{Obs}\}$*

We can show that $Init; T_1 \preccurlyeq_{\text{OBS}} Init; T_3$ by assuming all writes are observable.



$[\![Init; T_1]\!]$



$[\![Init; T_3]\!]$

There are only two maximal conflict-free event sets in $T_1$, those being $X_0 = \{0_1, 1_1, 2_1\}$ and $X_1 = \{0_1, 1_1, 3_1\}$. There is no event which writes a 1 to $y$ so we are unable to complete $X_1$, leaving us with a single complete execution $X_0$ in which $0_1 \xrightarrow{\text{HB}_{X_0}} 1_1$.

Looking at executions of $T_3$ for something similar to $X_0$, we can isolate $X'_0 = \{0_3, 1_3, 2_3\}$ where $0_1 \sim 0_3$ and $1_1 \sim 2_3$. In $X'_0$ we similarly have the initialisation event before the write as

$0_3 \xrightarrow{\text{HB}_{X'_0}} 2_3$. This fulfils the final requirement of Definition 13, giving us $T_1 \preccurlyeq_{\text{Obs}} T_2$.

We can also show the inverse – that $\mathit{Init}; T_3 \preccurlyeq_{\text{Obs}} \mathit{Init}; T_1$. The execution $X'_0$ is the only complete one, thus the same reasoning holds.

Observational refinement provides a formal basis to work from, but it can only be used to reason about whole programs. In this case, to realistically allow a program fragment like $T_3$ to be transformed into $T_1$, we would need to show observational refinement for every possible input memory state. What we really want is to validate that the transformation will preserve observational refinement in all contexts. This is the goal of the full refinement relation, which we write as $T_1 \preccurlyeq T_3$.

## 4.3 Modular Refinement

With our definition of a permissable program transformation established, we now explain how the modular refinement relation, which extends the idea of a permissable program transformation to a permissable program fragment transformation, is constructed.

Showing $P_1 \preccurlyeq P_2$ is done in three steps: we first construct an *embedding*, which relates events from an execution of $P_1$ to an execution of $P_2$. This process is described in Section 4.3.2. In Section 4.3.3 we then use these embeddings, one per execution pair, to establish that each execution in $P_1$ *simulates* an execution of $P_2$. Finally, for each simulation pair, we show a property called *common prefix preservation*, described in Section 4.3.4.

However, if our goal is for these relations to continue to hold in arbitrary contexts, we first need to ensure that our initial and final thread-local environments are the same beween the source and target program. We can ensure the initial environments are the same by contructing both event structures with the same environment parameter, but the structure itself does not describe any changes to that parameter, as those are handled entirely by the interpretation function. To expose these updates, we present a slight modification to the semantics which makes the final register environments explicit.

### 4.3.1 Environment Tracking

If two programs finish executing with differing register environments, it is trivial to construct a context which produces unequal sets of observable events by simply printing the contents of the registers. As the current semantics makes no record of actions on registers at all, we define a small extension to track them.

The extended semantic interpretation function $[\![P]\!]^T_{n\ \rho\ \kappa}$ is identical to $[\![P]\!]_{n\ \rho\ \kappa}$ in all cases except appending reads, and produces objects identical to those of the original denotational semantics except for the addition of a new component inside executions and an extended label signature. Reads in $[\![P]\!]^T_{n\ \rho\ \kappa}$ have labels which track the target register of the read. The new reads are ordered by the *register order* $<_r$, which is program order restricted to reads which store a value in the same register.

$$(r_1 : \mathrm{R}\ \mathtt{r}_i\ x\ v_1) <_r (r_2 : \mathrm{R}\ \mathtt{r}_i\ y\ v_2) \Leftarrow r_1 \sqsubseteq r_2$$

Executions in $[\![P]\!]^T_{n\ \rho\ \kappa}$ are 5-tuples:

$$(E, \mathrm{LK}, \mathrm{RF}, \mathrm{DP}, \rho)$$

Where the extra component $\rho$ is an environment represented as a function from registers to their contents. In addition to the completeness and coherence requirements of Section 3.1.3, these executions have one additional constraint: the environment $\rho$ must map each register $i$ to the value read into it at the $<_r$-latest read.

$$\forall i.\ \rho_X(i) = v \Leftrightarrow \exists (r : \mathrm{R}\ \mathtt{r}_i\ x\ v.\ \forall e \in E_X.\ \neg(r <_r e)$$

Once a complete denotation has been produced, $\rho$ indicates the final register state after a given execution.

The extra label component on reads is ignored for the purposes of label isomorphism, to avoid changing the behaviour of coproduct. It is still the case that $\mathrm{R}\ \mathtt{r}_{r1}\ x\ 1 \sim \mathrm{R}\ \mathtt{r}_{r2}\ x\ 1$, and any such pair as valid in a label-preserving bijection.

Returning to the example, the target $T_1$ and source $T_3$ actually have slightly different final register environments for otherwise similar executions. Adding the following context to both programs:

$$C[P] = \texttt{r2 := 0; \_; z := r1; z := r2;}$$

Gives:

| $C[T_3]$ | $C[T_1]$ |
|---|---|
| `r2 := 0;` | |
| `r1 := y;` | |
| `if (r1 = 1){` | `r2 := 0;` |
| `    r2 := y;` | `x := 1;` |
| `    x := r2;` | `r1 := y;` |
| `} else {` | `z := r1;` |
| `    x := 1;` | `z := r2;` |
| `}` | |
| `z := r1;` | |
| `z := r2;` | |

The program $C[T_1]$ can perform the writes $\{(\mathrm{W}\ x\ 1), (\mathrm{W}\ z\ 1), (\mathrm{W}\ z\ 0)\}$ with the 1 written to $z$ first and the 0 written second. The program $C[T_3]$ cannot, as the value of 0 initially stored in $r_2$ is never overwritten.

In light of this, we amend $T_1$ to copy the value of $r_1$ into $r_2$ and name the result $T_1'$.

| $T_1$ | $\longrightarrow$ | $T_1'$ |
|---|---|---|
| `x := 1;` | | `x := 1;` |
| `r1 := y;` | | `r1 := y;` |
| | | `r2 := r1;` |

The event structures of $T_1$ and $T_1'$ are identical, as register-only operations generate no memory accesses.

### 4.3.2 Embeddings and Validity

When describing refinement, we want to relate the set of executions in the target program with the set of executions in the source program. This means establishing an execution-to-execution relation to form the basis of this set comparison, and an event-to-event relation across these executions. However, as discussed earlier, label isomorphism is too strong. Consider the following two programs:

```
r1 := x;     and     r1 := y;
                     r1 := x;
```

It is clear that in all contexts they will behave equivalently - the read from $y$ in the second program is immediately overwritten with a read from $x$. However, there is no event isomorphic to that read in the first program. When comparing executions, we want a notion of a *redundant read*, which we should be free to discard.

Similarly, compilers often want to remove a *redundant write*:

```
x := 1;     and     x := 1;
                    x := 1;
```

A context can distinguish these two programs by continuously writing to and reading from $x$ in parallel. However, the first program has strictly fewer behaviours than the second – the first program can change the value of $x$ once, while the second program may be seen to change it either once or twice.

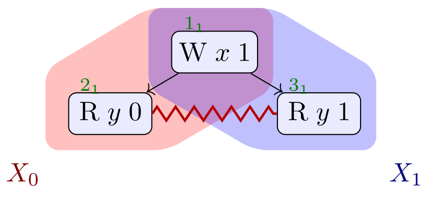
To capture redundant event transformations, we allow reads and writes in refinement to be related one-to-many from the target program to the source program. We ensure the redundancy of the missing events when we relate the executions themselves.

The event relation is formalised as a *validity* requirement for an embedding, the first component of a full definition for refinement.

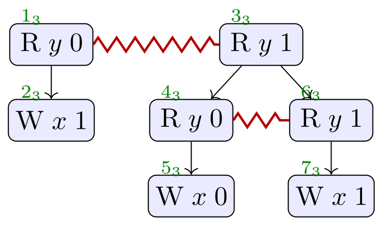
**Definition 9** (Embedding Validity)**.** *An embedding $\simeq$ is valid for two executions $X_1 \in [\![P_1]\!]_{n\ \rho\ \kappa}$ and $X_2 \in [\![P_2]\!]_{n\ \rho\ \kappa}$ if and only if:*

- *$\simeq$ is label-preserving: if $e_1 \simeq e_2$ then $e_1$ and $e_2$ are label isomorphic.*
- *$\simeq$ is symmetric.*
- *Every write in $E_1$ is related under $\simeq$ to at least one write in $E_2$, while every write in $E_2$ must relate to a single write in $E_1$.*
- *Every lock and unlock in $E_1$ is related under $\simeq$ to a unique lock or unlock in $E_2$, and vice versa.*
- *Every read in $E_1$ is related under $\simeq$ to at least one read in $E_2$.*

To return to our example, we established earlier that there are two executions of $T_1'$. We have execution $X_0$ which reads 0 at the final read from $y$, and execution $X_1$ which reads 1. We must now find an execution $X_0'$ of $T_3$ and embedding $\simeq_0$ such that $X_0 \simeq_0 X_0'$, and execution $X_1'$ and embedding $\simeq_1$ such that $X_1 \simeq_1 X_1'$.



$[\![T_1']\!]$



$[\![T_3]\!]$

Since there is only one execution of $T_3$ which reads a 0 from $y$, there is only one choice for $X'_0$, this being $\{1_3, 2_3\}$. There are two possible executions which read a 1 from $y$, but only one of them also writes a 1 to $x$. This means that there is also only one possible choice for $X'_1$, which is $\{3_3, 6_3, 7_3\}$. The choice of the embeddings $\simeq_0$ and $\simeq_1$ are shown as purple dotted lines in Figure 41.

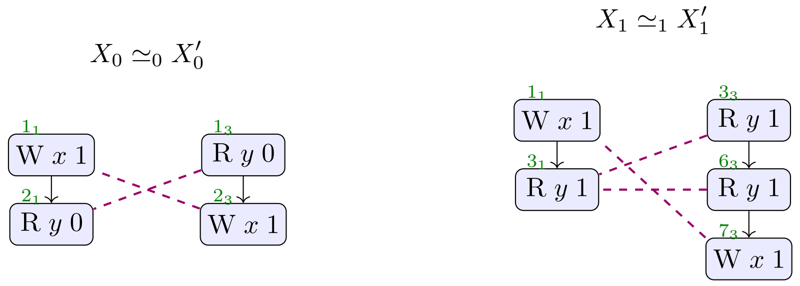


Figure 41: Embeddings between executions of $T'_1$ and $T_3$

### 4.3.3 Execution Simulation

Now that we have a notion of equivalence over events between executions, we can start to compare how they behave.

Recalling the definition for observational refinement, we at least require that HB orders observable writes in the target program more strongly than in the source program. This entails that whenever a pair of events are ordered by $\leq$ or LK in the source program, their equivalents under embedding in the target program are likewise ordered.

We should also consider dependencies, as these both create ordering edges and can influence program outcomes in sequential composition. We illustrate this with another pair of programs:

```
r1 := x;
r2 := y;
if (r1 = r2) {                 r1 := x;
    z := r1;         and       z := r1;
}
```

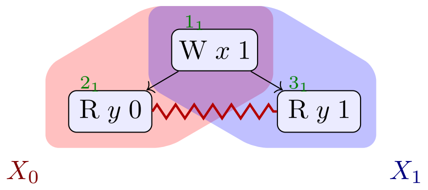
This is an attempt to remove a non-redundant read. In the source program, there is a dependency edge between the read from $y$ and the write to $z$. The target program has no read from $y$, and the only dependency for the write is a read from $x$. Reflecting this, we have a justification edge from both of the two reads of the source program, but only a single justification edge from one read in the second. To forbid this transformation, we therefore require equivalent justification edges for equivalent events. As justification edges are frozen into dependency edges, we similarly restrict dependency.

Combining these, and using the notation $R^?$ to denote the reflexive closure of relation $R$, we arrive at a definition of *execution simulation*.

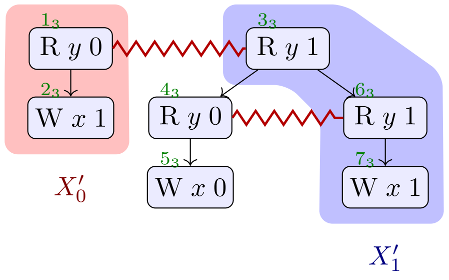
**Definition 10** (Execution Simulation)**.** *For two executions $X_1 \in [\![P_1]\!]^T_{n\ \rho\ \kappa}$ and $X_2 \in [\![P_2]\!]^T_{n\ \rho\ \kappa'}$, we say $X_1$* simulates *$X_2$ under valid embedding $\simeq$, written $X_1 \preccurlyeq_\simeq X_2$, if the following holds for all $e_1 \in E_1$ and $e_2 \in E_2$ whenever $e_1 \simeq e_2$:*

- $LK_2|^?_{e_2} \subsetneq LK_1|^?_{e_1}$
- $DP_2|_{e_2} \simeq DP_1|_{e_1}$
- $\leq_2 |^?_{e_2} \simeq \leq_1 |^?_{e_1}$
- *For every $e_1$ in $E_1$ and $C_1$ such that $C_1 \vdash e_1$, for $e_2 \simeq e_1$ there exists $C_2$ such that $C_2 \vdash e_2$ and $C_1 \simeq C_2$*
- $\rho_1 = \rho_2$

All executions in $P_1$ must simulate at least one execution in $P_2$ for $P_1 \preccurlyeq P_2$ to hold. Returning to our JMM example, this means that every execution of $T_1'$ must simulate at least one execution of $T_3$. Having already named $X_0$ and $X_1$, chosen executions $X_0'$ and $X_1'$ of $T_3$ and created embeddings $\simeq_0$ and $\simeq_1$, we must show $X_0 \preccurlyeq_{\simeq_0} X_0'$ and $X_1 \preccurlyeq_{\simeq_1} X_1'$.



$[\![T_1']\!]$



$[\![T_3]\!]$

Execution $X_0$ contains no $\leq$ or justification edges, thus $X_0'$ must likewise contain no such edges. This means that event $2_3$ must be independent of $1_3$, which holds: prior to any forwardings we have $\{1_3\} \vdash 2_3$ and $\{3_3, 6_3\} \vdash 7_3$, then after forwarding calculations the latter is simplified to $\{3_3\} \vdash 7_3$, and after coproduct events $2_3$ and $7_3$ are independent.

The same is true for execution $X_1$; there are no $\leq$ or justification edges. In $X_1'$ however, we have that $3_3 \leq 6_3$. Since the embedding is relating both of these events to $3_1$, we have to show there exists an event $e$ such that $e \leq^? 3_1$ where $e \simeq 3_3$ to match $6_3$ and $3_1 \leq^? e$ where $e \simeq 6_3$ to match $3_3$. Thanks to the reflexive closure, we can just use $3_1$ for both.

Relating two programs execution-to-execution using simulation alone is unfortunately still too weak. If we want refinement to be closed under the application of an arbitrary context, it must continue to hold after adding events to the bottom of the structure. Under the existing definitions, this is not always the case. We illustrate and rectify this by adding the property of *common prefix preservation*.

### 4.3.4 Common Prefix Preservation

Any two maximal conflict-free sets of events in the same structure must satisfy the following property: either the sets are the same, or there is at least one point in the structure at which these two sets enter into conflict with each other by each containing a different conflicting event. In the labelled event structures used by MRD, this can only occur at a read. All events before the diverging read event (or events) in program order must therefore be shared between both executions, and these shared events form a *common prefix* between the two executions. For executions of multi-threaded programs it is possible to have two such points, as program order is no longer a total order, but the executions will still share a single common set of events which can be referred to as a single prefix.

Common prefix preservation requires that whenever two executions in the target program share a common prefix in the event structure, the two executions in the source program they respectively simulate must also share a common prefix up to the same point.

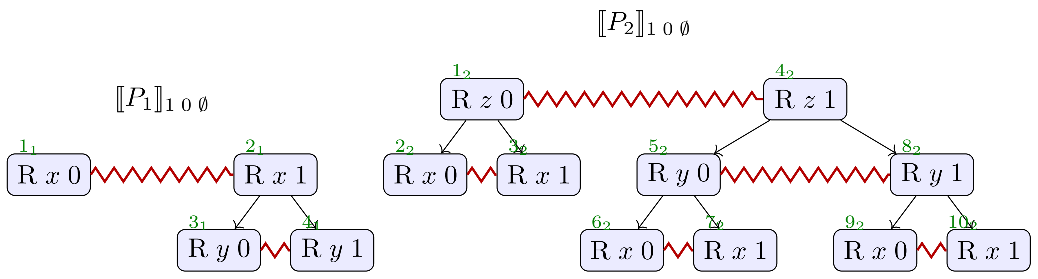
In this example, the contents of register $r_2$ are influenced by a read from $x$ in $P_1$ and a read from $z$ in $P_2$.

$P_1$:

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

$P_2$:

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



However, as reads are not given dependencies in MRD, the differences in control flow are not directly represented in the structures, and we can establish valid embeddings and execution simulations from $P_1$ to $P_2$.

Let $X_0 = \{1_1\}$, $X_1 = \{2_1, 3_1\}$, and $X_2 = \{2_1, 4_1\}$. Then:

1. $1_1 \simeq_0 2_2$ and $X'_0 = \{1_2, 2_2\}$
2. $2_1 \simeq_1 7_2$, $3_1 \simeq_1 5_2$, and $X'_1 = \{4_2, 5_2, 7_2\}$
3. $2_1 \simeq_2 10_2$, $4_1 \simeq_1 8_2$, and $X'_2 = \{4_2, 8_2, 10_2\}$
4. Refinement follows trivially from embedding validity in all cases, as there are no $\leq$, LK or justification edges.

If we now append the code fragment `x := r2`, the resulting write will have a justification edge from a read of $x$ in $P_1$ but from a read of $z$ in $P_2$. This is shown in Fig. 42.

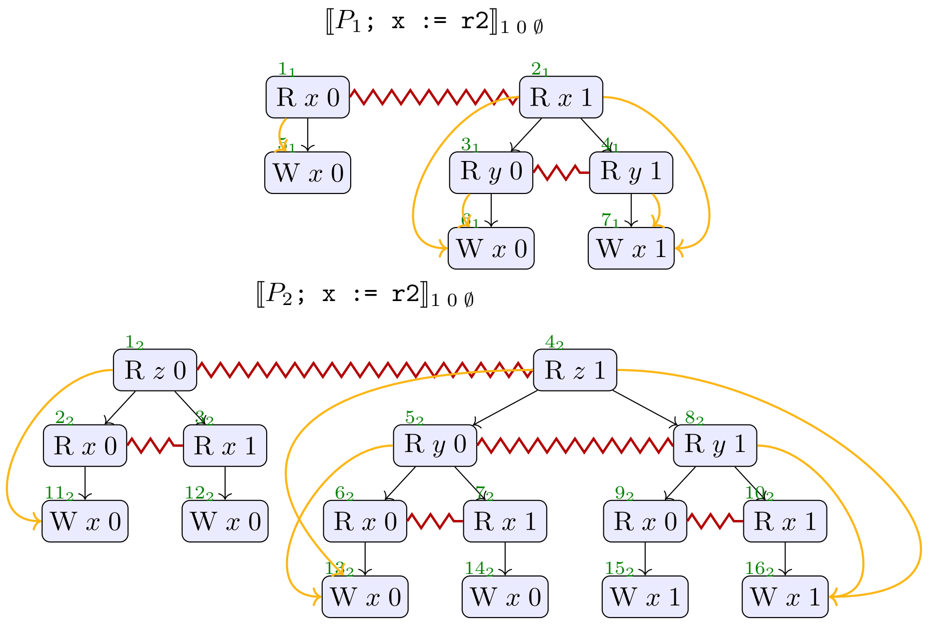


Figure 42: The denotations for $P_1$`; x := r2` and $P_2$`; x := r2`.

The execution $X_0$ is now $\{1_1, 5_1\}$ where $1_1 \vdash 5_1$, and has no equivalent in $P_2$`; x := r2`. All writes of 0 to $x$ in $P_2$`; x := r2` have a justification edge from some read of $z$, which has no equivalent in $P_1$`; x := r2`, and since executions of $P_1$`; x := r2` must have equivalent dependency or justification edges for refinement to hold we can no longer show $P_1$`; x := r2` $\preccurlyeq$ $P_2$`; x := r2` despite $P_1 \preccurlyeq P_2$. We forbid claiming refinements such as these by introducing common prefix preservation.

The set of read events which cause a pair of executions in the same event structure to diverge is referred to as the set of *least divergent points*, and defined as follows:

**Definition 11** (Least Divergent Points)**.** *The event pair $(d, d')$ is in $LDP(X_1, X_2)$ if and only if*

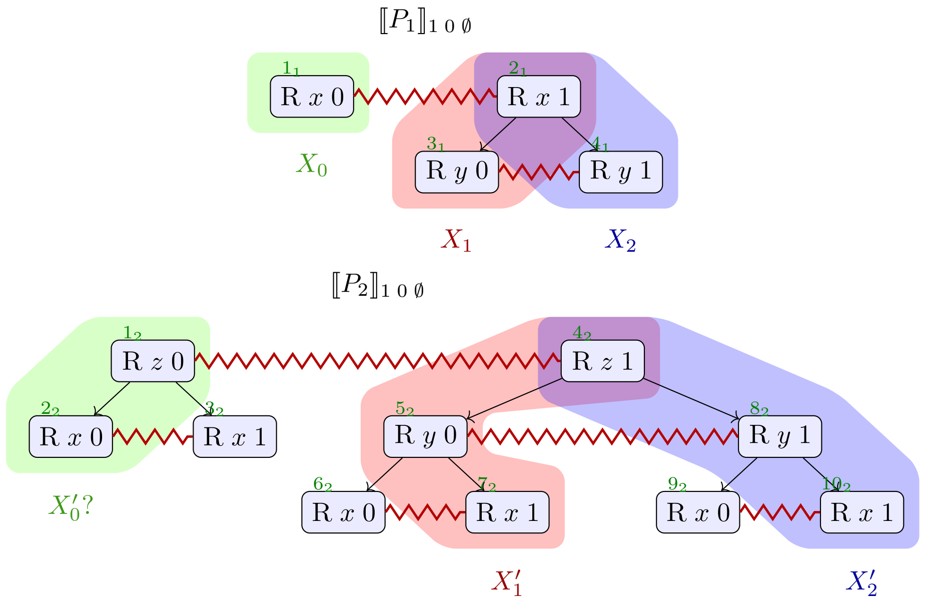
- *$d \# d'$ and there are no program order edges between $d$ and $d'$.*
- *$d \in X_1$ and $d \notin X_2$*
- *$d' \in X_2$ and $d' \notin X_1$*
- *For all $e \in X_1$, if $e \sqsubseteq d$ then $e \in X_2$ – the symmetric requirement for $e' \sqsubseteq d'$ is entailed.*

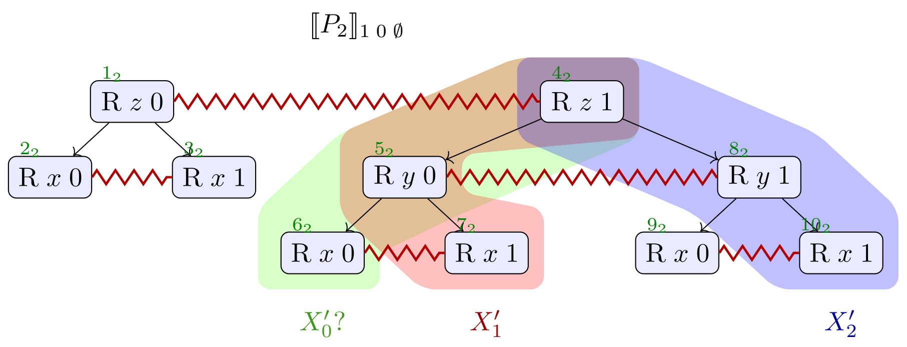
If $X_1$ and $X_2$ are identical, then $\mathrm{LDP}(X_1, X_2)$ is the empty set.

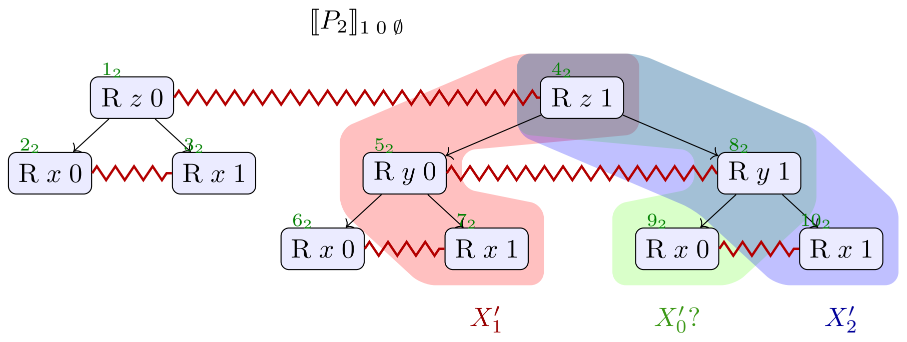
Returning to our programs from earlier, the executions $X_0$ and $X_1$ diverge at a read from $x$, and so do executions $X_0$ and $X_2$, while executions $X_1$ and $X_2$ diverge at a read from $y$. Common prefix preservation would require that, when finding the executions simulated by these, the simulated executions must likewise diverge at equivalent reads.

**Definition 12** (Common Prefix Preservation)**.** *A set of embeddings $\Gamma$ containing relations $\simeq_i$ is* common prefix preserving *if and only if:*
*For all $X_1, X_1', X_2, X_2'$,*
*if $X_1 \preccurlyeq_{\simeq_1} X_1'$ and $X_2 \preccurlyeq_{\simeq_2} X_2'$*
*then $LDP(X_1, X_2)(\simeq_1 \times \simeq_2)LDP(X_1', X_2')$*

It is no longer possible to show that $P_1$ refines $P_2$ in the previous example: as illustrated below, if we let $X_0'$ be $\{1_2, 2_2\}$ then it diverges from other executions at a read from $z$. If we let it be $\{4_2, 5_2, 6_2\}$ then it diverges from $X_1'$ at a read from $x$ as desired but diverges from $X_2'$ at a read from $y$. The inverse holds if we let $X_0'$ be $\{4_2, 8_2, 9_2\}$ – it now diverges from $X_2'$ at a read from $x$ but $X_1'$ at a read from $y$.


$[\![P_1]\!]_{1\ 0\ \emptyset}$
$1_1$
R $x$ 0
$2_1$
R $x$ 1
$3_1$
R $y$ 0
$4_1$
R $y$ 1
$X_0$
$X_1$
$X_2$
$[\![P_2]\!]_{1\ 0\ \emptyset}$
$1_2$
R $z$ 0
$2_2$
R $x$ 0
$3_2$
R $x$ 1
$4_2$
R $z$ 1
$5_2$
R $y$ 0
$6_2$
R $x$ 0
$7_2$
R $x$ 1
$8_2$
R $y$ 1
$9_2$
R $x$ 0
$10_2$
R $x$ 1
$X'_0$?
$X'_1$
$X'_2$



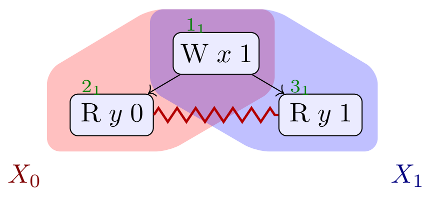


Our final definition of refinement includes common prefix preservation, preventing us from claiming refinement between these two programs.

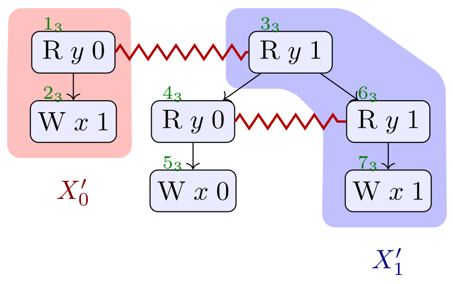
### 4.3.5 Full Definition

**Definition 13** (Modular Refinement). $[\![P_1]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \kappa}$ *if and only if there exists a common prefix preserving set of embeddings* $\Gamma$ *such that for all* $X_1 \in [\![P_1]\!]^T_{n\ \rho\ \kappa}$ *there exists* $X_2 \in [\![P_2]\!]^T_{n\ \rho\ \kappa}$ *and* $\simeq \in \Gamma$ *such that* $X_1 \preccurlyeq_{\simeq} X_2$.

In Section 4.3.2 we isolated executions $X_0$ and $X_1$ of $T_1'$ and executions $X_0'$ and $X_1'$ of $T_3$, and showed that $\simeq_0$ and $\simeq_1$ are valid embeddings for their respective pairs. We then showed, in Section 4.3.3, that $X_0 \preccurlyeq_{\simeq_0} X_0'$ and $X_1 \preccurlyeq_{\simeq_1} X_1'$, providing an equivalent in $T_3$ for each execution in $T_1'$. It remains to show that $\{\simeq_0, \simeq_1\}$ is common prefix preserving.



$[\![T_1']\!]$



$[\![T_3]\!]$

Whether $X_0$ or $X_1$ occurs at runtime ultimately depends on the value stored in `y`, as the point at which these two executions diverge is the read from $y$ at the end of the program, with $X_0$ containing $(2_1 : \mathrm{R}\ y\ 0)$ and $X_1$ containing $(3_1 : \mathrm{R}\ y\ 1)$. The LDP of $X_0$ and $X_1$ is therefore $\{(2_1, 3_1)\}$.

Given that $X_0$ simulates $X_0'$ and $X_1$ simulates $X_1'$, we want $X_0'$ and $X_1'$ to also diverge at a pair of equivalent events under $\simeq_0$ and $\simeq_1$ respectively. $X_0'$ diverges from $X_1'$ when $X_0'$ performs event $(1_3 : \mathrm{R}\ y\ 0)$ and $X_1'$ performs event $(3_3 : \mathrm{R}\ y\ 1)$, making their LDP $\{(1_3, 3_3)\}$. As $2_1 \simeq_0 1_3$ and $3_1 \simeq_1 3_3$, shown in Fig. 41, the embedding set is therefore common prefix preserving.

## 4.4 Properties of Refinement

In this section we show the required properties of refinement: that it is sound with respect to observational refinement, that it is closed under the addition of a new continuation, and finally that it is closed under context provided it holds over both programs for all register environments. In Section 4.5 we introduce a convenient shorthand for showing the required refinement across all register environments at once, effectively extending closure under context to a single input refinement proof and no longer requiring one proof per potential register environment.

### 4.4.1 Soundness

The goal of the modular refinement relation is to show observational refinement under all contexts, so it should first be shown that it entails observational refinement when applied to whole programs.

**Lemma 1** (Soundness)**.** *For all programs $P_1$ and $P_2$, step indeces $n$, and environments $\rho$, if $[\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$ then $[\![P_1]\!]_{n\ \rho\ \emptyset} \preccurlyeq_{OBS} [\![P_2]\!]_{n\ \rho\ \emptyset}$*

*Proof.* We begin with an arbitrary execution $X_1$ of $P_1$, then take $X_2$ and $\simeq$ such that $X_1 \preccurlyeq_{\simeq} X_2$. We note that, as $\simeq$ entails $\sim$, we can use $\simeq$ in place of $\sim$ when showing observational refinement. This leaves us with two goals:

1. $E_1|_{Obs} \simeq E_2|_{Obs}$

2. $\forall e_1 \in E_1|_{Obs}, e_2 \in E_2|_{Obs}.\ e_1 \simeq e_2 \Rightarrow$
   $\{e \mid e \xrightarrow{\text{HB}_{X_1}} e_1\} \supsetsim \{e \mid e \xrightarrow{\text{HB}_{X_2}} e_2\} \wedge$
   $\{e \mid e_1 \xrightarrow{\text{HB}_{X_1}} e\} \supsetsim \{e \mid e_2 \xrightarrow{\text{HB}_{X_2}} e\}$

The first component is trivially satisfied by $X_1 \preccurlyeq_{\simeq} X_2$ and Definition 9. For the second, we assume events $e_1 \in E_1|_{Obs}$ and $e_2 \in E_2|_{Obs}$ such that $e_1 \simeq e_2$. To establish $\text{HB}2|_{e_2} \subsetsim \text{HB}1|_{e_1}$, it suffices to show:

- Going forwards in happens-before, if there exists $e_2'$ such that $e_2 \xrightarrow{\text{HB}} e_2'$ then there must exist $e_1'$ such that $e_1' \simeq e_2'$ and $e_1 \xrightarrow{\text{HB}} e_1'$.
- Going backwards in happens-before, if there exists $e_2'$ such that $e_2' \xrightarrow{\text{HB}} e_2$ then there must exist $e_1'$ such that $e_1' \simeq e_2'$ and $e_1' \xrightarrow{\text{HB}} e_1$.

Their proofs are similar, so we describe only the first case in detail.

Assuming $e_2'$ such that $e_2 \xrightarrow{\text{HB}_{X_2}} e_2'$ holds, we know that one of the following must be true:

1. $e_2 = e_2'$
2. $e_2 \xrightarrow{\text{LK}_{X_2}} e_2'$
3. $e_2 \leq e_2'$
4. $\exists e.\ e_2 \xrightarrow{\text{LK}_{X_2}} e \wedge e \xrightarrow{\text{HB}_{X_2}} e_2'$
5. $\exists e.\ e_2 \leq_2 e \wedge e \xrightarrow{\text{HB}_{X_2}} e_2'$

Constructing a suitable $e_1'$ is trivial for the first three cases. In the first case we simply take $e_1' = e_1$. The next two cases arise from the clauses $\text{LK}_{X_1}|_{e_1} \supsetsim \text{LK}_{X_2}|e_2$ and $\leq_1 |_{e_1} \simeq \leq_2 |_{e_2}$ in Definition 10. The truth of the final two cases arises by transitive closure from the first three.

□

## 4.4.2 Closure under continuation

The first step to showing the modularity of our refinement relation is showing that it is closed under the addition of a continuation.

**Lemma 2** (Closure under continuation)**.** *For all $P_1$, $P_2$, $\rho$: If $[\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$ where $\emptyset$ returns an empty coherent event structure for any input, then for all continuations $\kappa_1$ and $\kappa_2$, if for all $\rho'$, $\kappa_1(\rho') \preccurlyeq \kappa_2(\rho')$ then $[\![P_1]\!]^T_{n\ \rho\ \kappa_1} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \kappa_2}$*

*Proof.* We begin with some execution $X_1$ of the full structure $[\![P_1]\!]^T_{n\ \rho\ \kappa_1}$, and split it into two sub-executions: one execution of $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$ and one execution of $\kappa_1(\rho_{P1})$ where $\rho_{P1}$ is the environment after the execution of $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$.

We name the first execution $X_{P1}$ of $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$, and the second $X_{\kappa 1}$ of $\kappa_1(\rho_{P1})$. There may be some edges from events in one to events in another, or differences in dependency edges, but we know that their event sets are identical.

By the definition of refinement, we must have some execution $X_{P2}$ of $[\![P_2]\!]^T_{n\ \rho\ \emptyset}$ and the embedding $\simeq_P$ such that $X_{P1} \preccurlyeq_{\simeq_P} X_{P2}$. Execution simulation mandates that $\rho_{P2} = \rho_{P1}$, which in turn means that the second half of $X_2$ covers some execution of $\kappa_2(\rho_{P1})$. Refinement of the continuations allows us to derive an execution $X_{\kappa 2}$ and embedding $\simeq_\kappa$ such that $X_{\kappa 1} \preccurlyeq_{\simeq_\kappa} X_{\kappa 2}$. We can now construct an execution $X_2$ of $[\![P_2]\!]^T_{n\ \rho\ \kappa_2}$ such that $E_2 = E_{P2} \cup E_{\kappa 2}$.

Constructing our executions in this way immediately gives us common prefix preservation. If we have $X_1'$ such that $X_1$ and $X_1'$ diverge in $P_1$, then the same will be true for $X_2'$ whenever $X_1' \preccurlyeq_{\simeq'} X_2'$ and likewise if the divergence is in $\kappa(\rho)$.

These constructions leave $X_1$ and $X_2$ underspecified. We have multiple choices for the contents of RF, LK and DP of each. The first two relations are completed on append and parallel composition operations, and each potential completion spawns a new execution. The freeze operation which generates DP edges also spawns new executions according to the contents of justification. We now show that any choice we can make for these relations in $X_1$ has an equivalent in $X_2$ under $\simeq$.

We initially construct the embedding $\simeq$ across these two executions as the union of $\simeq_\kappa$ and $\simeq_P$, which is trivially valid over $X_1$ and $X_2$.

**Preserved Program Order** The $\sqsubseteq$ relation over $E_1$ is trivially the union of $\sqsubseteq$ over $E_{P1}$ and $E_{\kappa 1}$, plus $E_{P1} \times E_{\kappa 1}$. As a restriction of $\sqsubseteq$, the $\leq$ relation is constructed similarly. We must now show that:

$$\forall e_1, e_1' \in E_1, e_2, e_2' \in E_2.\ e_2 \leq_2 e_2' \wedge e_1 \simeq e_2 \wedge e_1' \simeq e_2' \Rightarrow e_1 \leq_1 e_1'$$

If $e_2 \leq_2 e_2'$, then either:

1. $e_2 \leq_{P2} e_2'$

2. $e_2 \leq_{\kappa 2} e'_2$
3. $e_2 \in E_{P2} \wedge e'_2 \in E_{\kappa 2} \wedge loc(e_2) = loc(e'_2)$
4. $e_2 \in E_{P2} \wedge e'_2 \in E_{\kappa 2} \wedge \exists (l_2 : \mathrm{L}).\ e_2 \sqsubseteq l_2 \sqsubseteq e'_2$

Assuming we have $e_1 \simeq e_2$ and $e'_1 \simeq e'_2$, we must show that $e_1 \leq_1 e'_1$. In the first two cases, we have the required result by $X_{P1} \preccurlyeq_{\simeq_P} X_{P2}$ and $X_{\kappa 1} \preccurlyeq_{\simeq_\kappa} X_{\kappa 2}$.

In the third, we know that $e_1$ must be in $X_{P1}$ and $e'_1$ in $X_{\kappa 1}$ by our construction of $\simeq$. Therefore $e_1 \sqsubseteq e'_1$, and by the label preservation of $\simeq$ we must also have $e_1 \leq e'_1$.

In the fourth, we know that $l_2$ must have some equivalent $l_1$ by the validity of $\simeq_P$ and $\simeq_\kappa$. If $l_2$ is in $X_{P2}$ then by $X_{P1} \preccurlyeq_{\simeq_P} X_{P2}$ there must be $l_1 \in X_{P2}$ such that $l_1 \simeq_{P1} l_2$ and $e_1 \sqsubseteq l_1$, while $l_1 \sqsubseteq e'_1$ arises from our construction of $\sqsubseteq$. The equivalent holds if $l_2$ is in $X_{\kappa 2}$.

**Lock Order** The LK relation contains the restriction of $\sqsubseteq$ to locks and unlocks, plus an arbitrary completion in the case of parallel composition. The above argument also holds for executions without parallel composition.

Parallel composition cannot create an LK edge between an event in $E_{P2}$ and one in $E_{\kappa 2}$, because by definition they are in sequence. This means that any LK edges added by parallel composition must be between two events in $X_{P2}$ or two events in $X_{\kappa 2}$. As $X_{P1} \preccurlyeq_{\simeq_P} X_{P2}$ and $X_{\kappa 1} \preccurlyeq_{\simeq_\kappa} X_{\kappa 2}$, these edges must be matched in $X_1$.

**Dependency and Justification** We now concern ourselves with the DP and $\vdash$ relations. Our goal is to show that if $C_1 \vdash w_1$ and $w_1 \simeq w_2$ then there is some $C_2 \vdash w_2$ such that $C_1 \simeq C_2$. We know that if such $C_1$ exists then $w_1$ has not been frozen and thus $\text{DP}1|_{w_1}$ is empty, and equally that if $\text{DP}1|_{w_1}$ is not empty then no such $C_1$ exists. We begin by handling justification edges, and later show how this translates to dependency edges.

Taking arbitrary $C_1$ such that $C_1 \vdash_1 w_1$, either $w_1$ is in $P_1$ or $w_1$ is in $\kappa_1$. In the first case, the justification built by $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$ will be the same as the justification built by $[\![P_1]\!]^T_{n\ \rho\ \kappa_1}$ and our goal is met by $[\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$.

In the second case, there will be some justification $C_{\kappa 2}$ for $w_2$ built by $\kappa_1(\rho_{P1})$, and by $\kappa_1(\rho) \preccurlyeq \kappa_2(\rho)$ we can take $C_{\kappa 2} \vdash w_2$ such that $C_{\kappa 1} \simeq C_{\kappa 2}$. It remains to show that the interpretation of $P_1$ cannot modify this justifying set in a way that $P_2$ cannot match. We accomplish this by induction over the justification-up-to function, beginning at the $\sqsubseteq$-maximal event $e_1$ in $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$ and its equivalent $e_2$, where $e_1 \simeq e_2$, in $[\![P_2]\!]^T_{n\ \rho\ \emptyset}$.

We start by showing that for every $C_1 \in \overset{e_1 \bullet S_1}{\vdash w_1}$, where the set $S_1$ is all events between $e_1$ and $w_1$ in program order, there exists $C_2 \in \overset{e_2 \bullet S_2}{\vdash w_2}$ such that $C_1 \simeq C_2$. In the inductive step, we show that if, for all $C_1, E_1$ and $w_1$, $C_1 \in \overset{E_1}{\vdash w_1} \wedge w_1 \simeq w_2 \Rightarrow \exists E_2,\ C_2 \in \overset{E_2}{\vdash w_2}\ .\ C_1 \simeq C_2$ then for every $r_1 \simeq r_2$ and $D_1$, if $D_1 \in \overset{r_1 \bullet E_1}{\vdash w_1}$ then there exists some $D_2 \in \overset{r_2 \bullet E_2}{\vdash w_2}$ such that $D_1 \simeq D_2$.

**Base Case** Under the definition of the justification-up-to function, we know that $\overset{e_1 \bullet S_1}{\vdash w_1}$ runs all coproduct operations in $\kappa_1(\rho_1)$ over some initial justifying set in $F(w_1)$ calculated over $\kappa_1(\rho_1)$.

We can therefore describe the difference between $C_1$ and $C_{\kappa 1} \cup C_{P1}$ for some $C_{P1} \subseteq X_1$ in two stages: first as the set of events potentially removed from $C_{\kappa 1}$ by forwardings from events in $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$, and then by the set of coproduct operations whose validity is modified by those events.

We begin by constructing the difference. If a forwarding operation occurs in $C_1$, then either:

- The forwarding is represented fully in $C_{P1}$, and therefore occurs in $C_{P2}$ by $[\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$. As we are only calculating justification up to the end of $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$, this case is irrelevant.

- The forwarding is represented fully in $C_{\kappa 1}$, and therefore occurs in $C_{\kappa 2}$ by $[\![P_1]\!]^T_{n\ \rho\ \kappa_1} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \kappa_2}$. This does not create a difference between $C_1$ and $C_{\kappa 1}$.

- The forwarding occurs across the continuation boundary, thus some $e_{P1} \in C_{P1}$ removes some $e_{\kappa 1} \in C_{\kappa 1}$.

For a forwarding relation across the continuation boundary to hold, $e_{P1}$ must be maximal in $\leq_{P1}$ and $e_{\kappa 1}$ must be minimal in $\leq_{\kappa 1}$, and by the definition of refinement the equivalent must hold for $e_{P2}$ and $e_{\kappa 2}$. This means there can be no intervening accesses to the same variable between $e_{P2}$ and $e_{\kappa 2}$, and by the label isomorphism requirement it must likewise be possible for $e_{P2}$ to remove $e_{\kappa 2}$ from $C_2$.

Given that initially $C_{\kappa 1} \simeq C_{\kappa 2}$, if $C_{\kappa 1}$ loses equivalent events under embeddings to $C_{\kappa 2}$ then both lose equivalent pairs of coproduct operations – if normally $C_{\kappa 1}$ would coproduct at $r_1$ but $r_1$ has been removed, then $C_{\kappa 2}$ must also lose some $r_2 \simeq r_1$. All sets in $\overset{e_1 \bullet S_1}{\vdash w_1}$ must therefore have an embedding equivalent in $\overset{e_2 \bullet S_2}{\vdash w_2}$.

**Induction** Expanding our hypothesis, that $D_1 \in \overset{r_1 \bullet E_1}{\vdash w_1}$, into $D_1 \in \overset{E_1}{\vdash w_1} \vee D_1 \in \sum_{r_1}(w_1)$, we can trivially discharge the first case and focus on the case in which $D_1$ arises from a coproduct at $r_1$.

We know that there must be some $r_1' \# r_1$ and some execution $X_1'$ containing $r_1'$, and likewise that for $r_2' \# r_2$ there must be some $X_2'$ containing $r_2'$. Recalling our construction of $X_1$ and $X_2$ from $X_{P1}$ and $X_{P2}$, we can similarly assert that $X_1'$ and $X_2'$ contain some $X_{P1}'$ and $X_{P2}'$ respectively. By $[\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$ and CPP, as before, we have $\simeq'$ such that $X_{P1}' \preccurlyeq_{\simeq_P'} X_{P2}'$, $X_{\kappa 1}' \preccurlyeq_{\simeq_\kappa'} X_{\kappa 2}'$, $\simeq' = \simeq_P' \cup \simeq_\kappa'$, and $r_1' \simeq' r_2'$. If $r_1'$ were in $\kappa_1(\rho)$, then trivially $r_2'$ must not be in some $C_2'$ by $\kappa_1(\rho_1) \preccurlyeq \kappa_2(\rho_2)$ as the coproduct must have occurred in the respective continuations. We continue assuming that $r_1$ and $r_2$ are in $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$ and $[\![P_2]\!]^T_{n\ \rho\ \emptyset}$ respectively. If $\sum_{r_1}(w_1)$ is nonempty, there must be some $w_1' \in X_1'$ such that $w_1' \sim w_1$.

Splitting these executions into event sets prior to the reads and after the reads gives us the following general shape:

$$\begin{tikzcd}
 & T_1 \arrow[dl] \arrow[dr] & & & T_2 \arrow[dl] \arrow[dr] & \\
r_1 \arrow[d] \arrow[rr, squiggly, no head] & & r_1' \arrow[d] & r_2 \arrow[d] \arrow[rr, squiggly, no head] & & r_2' \arrow[d] \\
S_1 \arrow[d] & & S_1' \arrow[d] & S_2 \arrow[d] & & S_2' \arrow[d] \\
w_1 & & w_1' & w_2 & & w_2'
\end{tikzcd}$$

We want to show the impossibility of all cases where $w_1$ and $w_1'$ can be lifted, but $w_2$ and $w_2'$ cannot.

Lifting of $w_2$ and $w_2'$ fails if there is some event $e_2 \in C_2$ where $e_2' \notin X_2'$ (or the equivalent inverse, where $e_2' \in C_2'$ and $e_2 \notin X_2$). We now show that this is impossible if the lift of $w_1$ and $w_1'$ has succeeded for some $C_1$ and $C_1'$.

We know that there must be some $e_1 \in C_1$ such that $e_1 \simeq e_2$ by our inductive hypothesis. From our assumption that the lift succeeds, there must also be one $e_1' \in C_1'$ such that $e_1 \sim e_1'$. From $X_{P1}' \preccurlyeq_{\simeq_P'} X_{P2}'$ and $X_{\kappa 1}' \preccurlyeq_{\simeq_\kappa'} X_{\kappa 2}'$ there must be at least one $e_2' \in X_2'$ such that $e_1' \simeq' e_2'$.

Lifting can now only fail if there are two potential events in $C_2'$, both $e_2'$ and some $e_2''$, such that $e_2 \sim e_2'$ and $e_2 \sim e_2''$ but only one $e_2 \in X_2$. We split these into the following cases, assuming our events are organised such that $e_2' \leq e_2''$:

1. If $e_2'$ and $e_2''$ are both reads, then either there is some intervening write $w$ to the same location of a different value preventing a load forwarding shape or $e_2''$ was re-included in

an upwards closure operation.

2. If $e_2'$ and $e_2''$ are both writes, then either there is some $r' \in X_2'$ and $r'' \in X_2'$ such that $e_2' \leq r' \leq e_2'' \leq r''$, one such read exists and the other write was added in upwards closure, or neither read exists and both writes were added in upwards closure.

If a forwarding shape has been prevented, there must be some event $g_2$ such that $e_2' \leq g_2 \leq e_2''$. As we have established that lifting succeeds across $X_1$ and $X_1'$, there cannot be an event $g_1$ where $e_1' \leq g_1 \leq e_1'$ and $g_1 \simeq' g_2$ as this would require both $e_1' \leq g_1$ and $g_1 \leq e_1'$. This would make $\simeq'$ invalid, which contradicts our hypothesis.

In the upwards closure cases, because any bijection used in coproduct must be $\leq$-preserving we can assert that these upward closures must be matching some other execution which takes this shape. We split this into the read case and the write case.

In the read case, assuming $e_2'$ and $e_2''$ have label R $x$ $v$, there must be some execution containing R $x$ $v$ $\leq$ W $x$ $v'$ $\leq$ R $x$ $v$ to require the upward closure. However, due to the $\leq$ closure requirement in coproduct, this fails unless $X_2'$ also contains some W $x$ $v'$, placing us into the forward blocked case.

In the write case, the other execution must have the shape W $x$ $v$ $\leq$ R $x$ $v$ $\leq$ W $x$ $v'$ $\leq$ R $x$ $v'$. Once again, to coproduct this with $X_2'$ and include anything in the upwards closure we need all 4 events to be present in both executions, thus we must be in the forward blocked case.

**Freezing** If $C_1 = \emptyset$ then there must be some lock $L_1$ at which $C_1$ is frozen. We take $e_1$ to be the event immediately $\sqsubseteq$-after $L_1$, likewise $e_2$ immediately after lock $L_2$ at which $C_2$ is frozen, and perform our previous induction over $\overset{\{e' \mid e \sqsubseteq^? e'\}}{\vdash w_1}$ and $\overset{\{e' \mid e' \sqsubseteq^? e_2\}}{\vdash w_2}$. As all coproduct operations are optional, any coproduct that occurs $\sqsubseteq$-before the equivalent of $e$ in $X_2$ can be ignored. □

### 4.4.3 Closure under context

We now proceed to show that when one program refines another, this refinement continues to hold in an arbitrary context. A context $\mathbb{C}$ is a fragment of valid syntax containing a hole marked with _. When applied to another syntax fragment $P$, the result, denoted $\mathbb{C}[P]$, replaces all instances of holes [_] with the body of $P$.

We give contexts a more formal structure by defining them over each operator in the language as follows:

1. $\mathbb{C} = [\_]$

2. $\mathbb{C} = R \,||\, \mathbb{C}'$
3. $\mathbb{C} = \mathbb{C}' \,||\, R$
4. $\mathbb{C} = R;\ \mathbb{C}'$
5. $\mathbb{C} = \mathbb{C}';\ R$
6. $\mathbb{C} = \texttt{if}\ B\ \texttt{then}\ \mathbb{C}'\ \texttt{else}\ R$
7. $\mathbb{C} = \texttt{if}\ B\ \texttt{then}\ R\ \texttt{else}\ \mathbb{C}'$

Where $\mathbb{C}$ and $\mathbb{C}'$ represent arbitrary contexts, and $R$ represents an arbitrary program with no hole.

**Lemma 3** (Closure under context)**.** *For all $P_1$ and $P_2$, if $\forall \rho.\ [\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$ then for all $\mathbb{C}$, $\kappa$, $\kappa'$, if $\kappa \preccurlyeq \kappa'$ then $\forall \rho.\ [\![\mathbb{C}[P_1]]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![\mathbb{C}[P_2]]\!]^T_{n\ \rho\ \kappa'}$*

Where $\kappa \preccurlyeq \kappa'$ means $\forall \rho.\ \kappa(\rho) \preccurlyeq \kappa'(\rho)$.

*Proof.* We proceed by structural induction over our context $\mathbb{C}$ via the cases defined above.

In the base case, where $\mathbb{C} = [\_]$, refinement trivially continues to hold. It remains to show that, assuming $[\![\mathbb{C}'[P_1]]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![\mathbb{C}'[P_2]]\!]^T_{n\ \rho\ \kappa'}$ for all $\rho$, the same holds for $\mathbb{C}[P_1]$ and $\mathbb{C}[P_2]$ in the remainder.

$$\mathbb{C} = R;\ \mathbb{C}' \quad \vee \quad \mathbb{C} = \mathbb{C}';\ R$$

For the first, we unwrap the constructor to give the goal

$$[\![R;\ \mathbb{C}'[P_1]]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![R;\ \mathbb{C}'[P_2]]\!]^T_{n\ \rho\ \kappa'}$$

The inductive hypothesis gives us that $\forall \rho. [\![\mathbb{C}'[P_1]]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![\mathbb{C}'[P_2]]\!]^T_{n\ \rho\ \kappa'}$, therefore by definition $\lambda\rho.\ [\![\mathbb{C}]\!]^T_{n\ \rho\ \kappa} \preccurlyeq \lambda\rho.\ [\![\mathbb{C}'[P_2]]\!]^T_{n\ \rho\ \kappa'}$ As refinement is reflexive, for arbitrary $\rho$ we have that $[\![R]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![R]\!]^T_{n\ \rho\ \emptyset}$, therefore by Theorem 3 refinement holds.

The next case is similar, but the inductive hypothesis gives $[\![\mathbb{C}[P_1]]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![\mathbb{C}[P_2]]\!]^T_{n\ \rho\ \emptyset}$ and reflexivity gives $\lambda\rho.\ [\![R]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![R]\!]^T_{n\ \rho\ \kappa'}$.

$$\mathbb{C} = \texttt{if}\ B\ \texttt{then}\ \mathbb{C}'\ \texttt{else}\ R \quad \vee \quad \mathbb{C} = \texttt{if}\ B\ \texttt{then}\ R\ \texttt{else}\ \mathbb{C}'$$

In the conditional cases, since we evaluate $\mathbb{C}[P_1]$ and $\mathbb{C}[P_2]$ under the same environment, we can assume that either $B$ will evaluate to true in both structures or that it will evaluate to false in both structures. The resulting structures are both stated to be refinements in the inductive hypotheses or by reflexivity.

$$\mathbb{C} = R \,||\, \mathbb{C}' \quad \vee \quad \mathbb{C} = \mathbb{C}' \,||\, R$$

We now handle the final two cases, noting that they produce equivalent event structures and therefore have equivalent proofs. By the inductive hypothesis we have that for all $\kappa, \kappa'$ such that $\kappa \preccurlyeq \kappa'$, $[\![\mathbb{C}'[P_1]]\!]^T_{n\ \rho\ \kappa} \preccurlyeq [\![\mathbb{C}'[P_2]]\!]^T_{n\ \rho\ \kappa'}$. For convenience we set the variable $C_1$ to be equal to the syntax fragment obtained by $\mathbb{C}'[P_1]$, and likewise for $C_2$.

Per the definition of $[\![R \mid\mid C_1]\!]$, we must show that $[\![L;U]\!]^T_{n\ \rho\ \emptyset} \bullet ([\![R]\!]^T_{n\ \rho\ \emptyset} \times [\![C_1]\!]^T_{n\ \rho\ \kappa_1}) \star [\![L;U]\!] \preccurlyeq [\![L;U]\!]^T_{n\ \rho\ \emptyset} \bullet ([\![R]\!]^T_{n\ \rho\ \emptyset} \times [\![C_2]\!]^T_{n\ \rho\ \kappa_2}) \star [\![L;U]\!]$. From the result for $\mathbb{C} = R;\mathbb{C}'$ above we can remove the initial lock/unlock pair and reduce our goal to $([\![R]\!]^T_{n\ \rho\ \emptyset} \times [\![C_1]\!]^T_{n\ \rho\ \kappa_1}) \star [\![L;U]\!] \preccurlyeq ([\![R]\!]^T_{n\ \rho\ \emptyset} \times [\![C_2]\!]^T_{n\ \rho\ \kappa_2}) \star [\![L;U]\!]$.

We construct the embedding $\simeq$ by observing that, since we can assume $[\![C_1]\!]^T_{n\ \rho\ \kappa_1} \preccurlyeq [\![C_2]\!]^T_{n\ \rho\ \kappa_2}$, we can infer the existence of a valid embedding which relates events in $C_1$ to events in $C_2$ $\cdot$ We assume an execution $X_{C_1}$ of $C_1$, an embedding $\simeq'$ and an execution $X_{C_2}$ of $C_2$ such that $X_{C_1} \preccurlyeq_{\simeq'} X_{C_2}$. Extending $\simeq'$ with equality over events in $R$ creates a valid embedding for $\mathbb{C}[P_1]$ and $\mathbb{C}[P_2]$.

The first condition for refinement, the equality of the sets of observable events, arises trivially from the construction of $\simeq$.

We now take an execution $X_1$ of $R \mid\mid C_1$ created by combining $X_{C_1}$ with an arbitrary execution $X_R$ of $R$ as shown in the definition for parallel composition, and likewise construct $X_2$ using the same execution of $R$. It remains to show that $X_1 \preccurlyeq_{\simeq} X_2$.

The requirements that $\leq_1 \simeq \leq_2$ and $\text{DP}_1 \simeq \text{DP}_2$ behave similarly, as the parallel composition operator never adds new edges to either. The final relations are unions of the relations from $R$ and $C_1$ or $R$ and $C_2$, and given $\leq_R = \leq_R$ and $\leq_{C_1} \simeq \leq_{C_2}$ (and likewise for $\text{DP}$) this union cannot violate the required property.

Due to the rules for completions of $\text{LK}$ and $\text{RF}$, the operation potentially creates a class of executions in $C_2$ such that the above properties hold, differing only in the contents of $\text{LK}$ and $\text{RF}$. We first show that a class exists such that $\text{LK}_1 \gtrsim \text{LK}_2$, then that an execution in that class satisfies $\text{RF}_1 \gtrsim \text{RF}_2$.

In $\text{LK}_1$, we have $\text{LK}'_1 \cup \text{LK}_R$ plus a component $\text{LK}'$ to form a total order. Since $\text{LK}'_1 \gtrsim \text{LK}'_2$, it remains to show that there exists a completion of lock order in $C_2$ which is a subset of $\text{LK}'_1$ over the embedding. For every pair in the completion $\text{LK}'$, each lock/unlock event $L_R$ is placed either before or after some lock/unlock event $L_{C_1}$. We already have that if $C_2$ and $R$ contain locks an isomorphic event $L_{C_2}$ exists in $C_2$, and an equivalent $L_R$ in $R$. If every possible completion is generated by $R \times C_2$, then one exists where $L_{C_2}$ and $L_R$ are in the same order as $L_{C_1}$ and $L_R$. If $C_2$ does not contain locks, then the property is trivial. A similar strategy shows the preservation of $\textsc{rf}_1 \supsetapprox \textsc{rf}_2$.

The dependency $\textsc{dp}_{X_1}$ must be the union of $\textsc{dp}_{X_{C_1}}$ and $\textsc{dp}_{X_R}$, thus by $X_{C_1} \preccurlyeq_{\simeq} X_{C_2}$ and $X_R = X_R$ we can assert that $\textsc{dp}_{X_{C_1}} \cup \textsc{dp}_{X_R} \simeq \textsc{dp}_{X_{C_2}} \cup \textsc{dp}_{X_R}$. The justification is likewise a union of disjoint edges, thus whenever $D_1 \vdash e_1$ and $e_1 \simeq e_2$ then there must be some $D_2 \vdash e_2$ such that $D_2 \simeq D_1$.

Common prefix preservation is somewhat complicated here by the fact that for any $X_1'$ of $R||C_1$ there is more than one LDP of $(X_1, X_1')$. Constructing our embeddings via equality on $R$ gives us CPP for divergences in the left half of the composition, and re-using embeddings over $C_1$ and $C_2$ gives CPP for divergences on the right.

It remains to show that joining onto a lock/unlock pair does not impact refinement. Since we have shown that the freezing operation preserves refinement, it remains only to show that the extensions to the embedding and $\textsc{lk}$ are safe. We can extend $\simeq$ trivially by relating the added events to themselves, and since every lock or unlock in $C_1$ must have an equivalent event under the embedding in $C_2$ it is clear that the new $\textsc{lk}$ edges relating the added events to the existing ones will generate isomorphic edges.

This results in the proof of $(\llbracket R \rrbracket^T_{n\ \rho\ \emptyset} \times \llbracket C_1 \rrbracket^T_{n\ \rho\ \kappa_1}) \star \llbracket L;\ U \rrbracket \preccurlyeq (\llbracket R \rrbracket^T_{n\ \rho\ \emptyset} \times \llbracket C_2 \rrbracket^T_{n\ \rho\ \kappa_2}) \star \llbracket L;\ U \rrbracket$.

□

## 4.5 Symbolic Environment Event Structures

To ensure $P_1 \preccurlyeq P_2$ continues to hold in a context which places $P_1$ and $P_2$ after some arbitrary code, as shown earlier, we have to start with a proof of $\forall \rho.\ \llbracket P_1 \rrbracket^T_{n\ \rho\ \emptyset} \preccurlyeq \llbracket P_2 \rrbracket^T_{n\ \rho\ \emptyset}$. As this could be very many $\rho$s, it would be preferable to abstract away the concrete values contained in the register environment and have a single structure from which we can derive our initial refinement.

We can construct the symbolic register environment 𐤓 (the letter Resh, which is the predecessor to $\rho$ in the Phoenician script), which maps every register $r_i$ to a special value $I_i$ about which we know nothing. However, using 𐤓 under our existing definitions cannot result in a complete event structure, as it is impossible to properly construct $\textsc{rf}$ and $\textsc{dp}$ edges or branch on a value without knowing anything about the values in our program.

To get around this, we define *symbolic environment event structures*, or SE event structures, which can later be instantiated into coherent event structures for any given initial environment.

An SE event structure is a set of pairs

$$\{(E, \phi)\}$$

Where:

- The event structure $E$ is an ordinary labelled event structure as defined in Section 3.1
- The condition $\phi$ is a predicate over environments, representing which conditions must hold over some input environment $\rho$ for this structure to arise when $\rho$ is passed to the semantic interpretation as an initial environment. If the initial environment is never used in a branch statement, the set will be a singleton with the condition $\top$.

Negating a condition using $\phi^-$ permits only environments which do not satisfy it.

$$\phi^- \triangleq \lambda\rho.; \neg(\phi(\rho))$$

Events in SE event structures can be relabelled from symbolic to concrete values using *instantiation*, which takes a register environment $\rho$ and uses it to replace symbolic initial values with the appropriate concrete initial values.

$$\mathcal{I}(e, \rho) \triangleq \begin{cases} (e : \mathrm{W}\ x\ \rho(\mathtt{r}_i)) & \text{if } (e : \mathrm{W}\ x\ I_i) \\ e & \text{otherwise} \end{cases}$$

**Preliminary: Conditional Labelling Functions** We also declare a pair of new functions over writes called the *conditional labelling* functions. The first is the positive conditional labelling function:

$$(w : \mathrm{W}\ x\ I_i) \overset{?}{\leftarrow} \mathrm{W}\ x\ v \triangleq \lambda\rho.\ \rho(r_i) = v$$

Given a write of a symbolic value $(w : \mathrm{W}\ x\ I_i)$ and a target concrete write label $\mathrm{W}\ x\ v$, it returns the smallest condition $\phi$ such that if $\phi(\rho)$ then $\rho(r_i) = v$. In the event that $(w : \mathrm{W}\ x\ v')$ is a write of a concrete value, $w \overset{?}{\leftarrow} \mathrm{W}\ x\ v$ always returns $\lambda x.\ F$ unless $v = v'$, in which case it returns $\lambda x.\ T$.

The second is the negative conditional labelling function

$$(w : \mathrm{W}\ x\ I_i) \not\leftarrow \mathrm{W}\ x\ v \triangleq \lambda\rho.\ \rho(r_i) \neq v$$

This returns the smallest condition such that $\rho(r_i)$ does *not* return $v$. Similarly to the positive conditional labelling function, this will return false if given two writes of the same concrete value and true if given two writes of different concrete values.

### 4.5.1 Conditionals

The set of event structures in the interpretation of a program can only grow when we branch on the contents of the initial register environment. If the guard contains a reference to an initial register value, we continue once in the structure where it held and once in the structure where it did not, annotate each result with the branch condition, and then take the union of these two structures.

$$
\begin{gathered}
[\![\texttt{if } B \texttt{ then } P_1 \texttt{ else } P_2]\!]^T_{n\ \rho\ \kappa} = \\
\text{let } \phi' \text{ be the minimal condition such that } [\![B]\!]_\rho \text{ in} \\
\{(CES, \phi \wedge \phi') \mid (CES, \phi) \in [\![P_1]\!]^T_{n\ \rho\ \kappa}\} \cup \{(CES, \phi \wedge \phi'^-) \mid (CES, \phi) \in [\![P_2]\!]^T_{n\ \rho\ \kappa}\}
\end{gathered}
$$

If we are interpreting a guard with no reference to initial register values, the result is unchanged from the initial semantics. Note that at any stage we may safely remove all structures whose condition is unsatisfiable, since the number of potential environments which can satisfy a condition is monotonically decreasing as further clauses are added.

### 4.5.2 Executions

The executions in $S$ of an SE event structure contain an extra field, the *execution condition* $\phi$.

$$X = (E_X, \mathrm{LK}_X, \mathrm{RF}_X, \mathrm{DP}_X, \phi_X)$$

The new field $\phi_X$ of execution $X$ is similarly a predicate over environments, this time describing the conditions that must hold for $\mathrm{RF}_X$ and $\mathrm{DP}_X$ to be valid reads-from and dependency relations. If the contents of the initial register environment never appear in events touched by $\mathrm{RF}_X$ or $\mathrm{DP}_X$, this will be $\top$. If a symbolic variable does appear in one of these events, $\phi_X$ will record the maximal possible predicate that must hold over this variable for such a relation to appear in an instantiation. This is calculated immediately for $\mathrm{RF}_X$, and calculated by the justification-up-to and freeze operations for $\mathrm{DP}_X$.

**Definition 14** (Coherent Symbolic Environment Execution)**.** *An execution $X$ of a symbolic environment structure is coherent if and only if:*

$$(\leq \cup LK_X \cup DP_X \cup RF_X)^* \textit{ is acyclic and, for } \sqsubseteq_{LK} = (\sqsubseteq \cup LK_X)^*$$
$$(w : W\,x\,v_w) \xrightarrow{RF}_X (r : R\,x\,v_r) \Longrightarrow \forall (e : R\,x\,v') \in E.\ (\phi \Rightarrow v_w = v') \vee \neg(w \sqsubseteq_{LK} e \sqsubseteq_{LK} r)$$
$$(w : W\,x\,v_w) \xrightarrow{RF}_X (r : R\,x\,v_r) \Longrightarrow \forall (e : W\,x\,v') \in E.\ \neg(w \sqsubseteq_{LK} e \sqsubseteq_{LK} r)$$

*And, in addition:*

$$(w : W\,x\,v_w) \xrightarrow{RF}_X (r : R\,x\,v_r) \Longrightarrow (\phi_X \Rightarrow v_w = v_r)$$
$$(w : W\,x\,v) \in E \Longrightarrow \exists (DP_w, \varphi) \in freeze(w).\ DP_w \subseteq DP_X \wedge (\phi \Rightarrow \varphi)$$

### 4.5.3 Justification

As before, the justification calculation consists of a forwarding step followed by a coproduct step, and we describe the forwarding first.

#### Forwarding

When handling the unknown initial register values, a write event may or may not cause a forwarding to happen, depending on the actual value the register will contain when the semantics is given a concrete environment. We therefore introduce a new store forwarding rule, which adds a condition to the resulting justifying set.

$$(w : \mathrm{W}\,x\,v) \xrightarrow{SF}^{\alpha=v} (r : \mathrm{R}\,x\,\alpha) \text{ if } \nexists e.\ w \leq e \leq r$$

We keep these conditions through the definition of the forwarding relation:

**Definition 15** (Conditional Forwarding)**.** *If* $r_1 \xrightarrow{LF} r_2$ *then* $r_1 \xrightarrow{F}^{\top} r_2$
*If* $w \xrightarrow{SF}^{\varphi} r$ *then* $w \xrightarrow{F}^{\varphi} r$
*If* $e_1 \xrightarrow{F}^{\varphi_1} e_2$ *and* $e_2 \xrightarrow{F}^{\varphi_2} e_3$ *then* $e_1 \xrightarrow{F}^{\varphi_1 \wedge \varphi_2} e_3$

While the definition of the set $F(w)$ combines the requirements for all forwarding edges it uses:

$$\begin{aligned}
forbid(w) &\triangleq \{e \mid \exists e'.\ \lambda(e') \in \{\mathrm{L}, \mathrm{U}\} \wedge e \sqsubseteq e' \sqsubseteq w\} \\
R(w) &\triangleq \{r \mid (r : \mathrm{R}\ x\ v) \sqsubseteq w \wedge r \notin forbid(w)\} \\
F(w) &\triangleq \{(S, \varphi_S) \mid e \in R \setminus S \Rightarrow \exists e'.\ e' \overset{\varphi_e}{\xrightarrow{F}} e \wedge e' \in S \wedge e' \notin forbid(w) \wedge \varphi_S \Rightarrow \varphi_e\}
\end{aligned}$$

**Coproduct**

Coproduct over conditional edges can combine their respective conditions, and create new ones if the writes or justifications would only be isomorphic given particular initial environments.

Given $\vdash^C$ edges $((C_1, e_1), \phi_1)$ and $((C_2, e_2), \phi_2)$ where $e_1 \sim e_2$. For every potential $D_1$ and $D_2$ such that

1. $D_1 \in \uparrow(C_1)$
2. $D_2 \in \uparrow(C_2)$
3. there is no event $e \in D_1$ such that $r_1 \leq e$

we take the weakest condition $\phi$ such that $\phi(\rho) \Rightarrow \mathcal{I}(D_1, \rho) \sim \mathcal{I}(D_2, \rho)$.

We now have $((D_1, e_1), \phi_1 \wedge \phi) \in \vdash^C$ and $((D_2, e_2), \phi_2 \wedge \phi) \in \vdash^C$.

This is passed through the justification-up-to calculation, with the initial justifying sets and conditions being those returned by conditional forwarding above. This means that $\overset{S}{\vdash} w$ now returns sets of the form $\{((C, w), \varphi)\}$.

## 4.5.4 Freezing

Freezing conditional justification edges transfers the condition into whichever execution contains the new dependency edge. As before, we take each write which has a justification edge and create a set of dependency edges from a single chosen justifying set. We now also take the corresponding condition and add it to the execution.

$$
\begin{aligned}
FP(w) &\triangleq \{e \mid \lambda(e) \in \{\mathrm{L}, \mathrm{U}\} \wedge e \sqsubseteq w\} \\
S(w) &\triangleq \begin{cases} \{e \mid e \sqsubseteq w\} \\ \qquad \vdash w & \text{if } FP = \emptyset \\ \{e \mid f \sqsubseteq e \sqsubseteq w\} \\ \qquad \vdash w \qquad \text{where } f \text{ is } \sqsubseteq\text{-maximal in } FP & \text{otherwise} \end{cases} \\
freeze(w) &\triangleq \{(\{(e, w) \mid e \in S\}, \varphi) \mid (S, \varphi) \in S(w)\}
\end{aligned}
$$

### 4.5.5 Semantic Interpretation

$$
\begin{aligned}
[\![P]\!]^T_{1\ \rho\ \kappa} &= \{(\emptyset, \top)\} \\
[\![\texttt{skip}]\!]^T_{n\ \rho\ \kappa} &= \kappa(\rho) \\
[\![\texttt{r := x}]\!]^T_{n\ \rho\ \kappa} &= \{(\Sigma_{v \in V} \mathrm{R}\ x\ v \bullet E, \phi | (E, \phi) \in \kappa(\rho[r \mapsto v]))\} \\
[\![\texttt{x := } M]\!]^T_{n\ \rho\ \kappa} &= \{((\mathrm{W}\ x\ [\![M]\!]_\rho) \bullet E, \phi) | (E, \phi) \in \kappa(\rho)\} \\
[\![P_1\texttt{;}\ P_2]\!]^T_{n\ \rho\ \kappa} &= [\![P_1]\!]^T_{n\ \rho\ \lambda\rho.[\![P_2]\!]^T_{n\ \rho\ \kappa}} \\
[\![P_1 \parallel P_2]\!]^T_{n\ \rho\ \kappa} &=
\end{aligned}
$$

$$
\begin{gathered}
\text{let } P_1 \times P_2 = \{(E_1 \times E_2, \phi_1 \wedge \phi_2) | (P_1, \phi_1) \in\ [\![P_1]\!]^T_{n\ \rho\ \emptyset},\ (P_2, \phi_2) \in\ [\![P_2]\!]^T_{n\ \rho\ \emptyset}\} \\
\text{and } P_1 \star P_2 = \{(E_1 \star E_2, \phi_1 \wedge \phi_2) | (P_1, \phi_1) \in\ [\![P_1]\!]^T_{n\ \rho\ \emptyset},\ (P_2, \phi_2) \in\ [\![P_2]\!]^T_{n\ \rho\ \kappa}\} \\
\text{in } (P_1 \times P_2) \star [\![\mathrm{L}\texttt{;}\ \mathrm{U}]\!]^T_{n\ \rho\ \kappa}
\end{gathered}
$$

$$
\begin{gathered}
[\![\texttt{if } B \texttt{ then } P_1 \texttt{ else } P_2]\!]^T_{n\ \rho\ \kappa} = \\
\text{let } \phi' \text{ be the minimal condition such that } B \text{ in} \\
\{(E, \phi \wedge \phi') | (E, \phi) \in [\![P_1]\!]^T_{n\ \rho\ \kappa}\} \cup \{(E, \phi \wedge \phi'^-) | (E, \phi) \in [\![P_2]\!]^T_{n\ \rho\ \kappa}\} \\
[\![\textbf{while}\texttt{(}B\texttt{)\{}P\texttt{\}}]\!]^T_{n\ \rho\ \kappa} = \\
\text{let } \phi' \text{ be the minimal condition such that } B \text{ in} \\
\{(E, \phi \wedge \phi') | (E, \phi) \in [\![P\,\texttt{;}\,\textbf{while}\texttt{(}B\texttt{)\{}P\texttt{\}}]\!]^T_{(n-1)\ \rho\ \kappa}\} \cup \{(E, \phi \wedge \phi'^-) | (E, \phi) \in [\![\texttt{skip}]\!]^T_{n\ \rho\ \kappa}\}
\end{gathered}
$$

$$
\begin{aligned}
[\![\mathrm{L}]\!]^T_{n\ \rho\ \kappa} &= \{(\mathrm{L} \bullet E_1, \phi) | (E_1, \phi) \in \kappa(\rho)\} \\
[\![\mathrm{U}]\!]^T_{n\ \rho\ \kappa} &= \{(\mathrm{U} \bullet E_1, \phi) | (E_1, \phi) \in \kappa(\rho)\}
\end{aligned}
$$

Figure 43: The full semantic interpretation function for programs which begin with the symbolic environment 4.

### 4.5.6 Soundness

If our construction is correct, then any execution of a symbolic environment event structure under a given condition $\phi$ must appear in any coherent event structure created from the same program with initial environment $\rho$ if $\phi(\rho)$ holds. To formalise this, we begin by lifting instantiation to completed denotations.

To interpret $\mathcal{I}(\llbracket P \rrbracket^{T}_{n\ \text{\reflectbox{4}}}, \rho)$, we first define instantiation over its components as follows:

Over an execution $(X, \text{LK}_X, \text{RF}_X, \text{DP}_X, \rho_X, \phi)$, it returns the corresponding instantiated execution only if its condition is satisfied by $\rho$. Otherwise, it returns the empty set. It also overwrites any leftover initial register contents in $\rho_X$ with their contents in $\rho$.

$$\mathcal{I}((X, \text{LK}_X, \text{RF}_X, \text{DP}_X, \rho_X, \phi_X), \rho) \triangleq$$
$$\{(X, \text{LK}_X, \text{RF}_X, \text{DP}_X, \rho') \mid \phi_X(\rho) \land \rho' = \lambda r.\ \text{if } \rho_X(r) = I_i \text{ then } \rho(r) \text{ else } \rho_X(r)\}$$

Over a conditional justification relation $\vdash^C$, it returns all edges whose conditions are satisfied by $\rho$:

$$\mathcal{I}(\vdash^C, \rho) \triangleq \{(C, w) \mid ((C, w), \phi) \in \vdash^C \land \phi(\rho)\}$$

Over a labelled event structure $(E, \sqsubseteq, \#, \lambda)$, it changes the labelling function $\lambda$ to give the instantiated label to its events.

$$\mathcal{I}(E, \sqsubseteq, \#, \lambda) \triangleq (E, \sqsubseteq, \#, \mathcal{I}(\lambda, \rho))$$

$$\mathcal{I}(\lambda, \rho) \triangleq \lambda e.\ \text{if } \lambda(e) = \text{W } x\ I_i \text{ then W } x\ \rho(r_i) \text{ else } \lambda(x)$$

Finally, over a completed structure, we take the single $((E, S, \vdash^C, \leq), \phi) \in \llbracket P \rrbracket^{T}_{n\ \text{\reflectbox{4}}}$ where $\phi(\rho)$, noting that this will be unique as all conditions in the final structure are mutually exclusive, and instantiate each of its components.

$$\mathcal{I}(\llbracket P \rrbracket^{T}_{n\ \text{\reflectbox{4}}\ \kappa}, \rho) \triangleq (\mathcal{I}(E, \rho), \mathcal{I}(S, \rho), \mathcal{I}(\vdash^C, \rho), \leq)$$

We can now phrase our soundness lemma as follows:

**Lemma 4** (Soundness). *If* $\mathcal{I}(\llbracket P \rrbracket^{T}_{n\ \text{\reflectbox{4}}\ \emptyset}, \rho) = (E, S, \vdash, \leq)$, *then* $\llbracket P \rrbracket_{n\ \rho\ \emptyset} = (E, S, \vdash, \leq)$

*Proof.* We proceed by induction over the structure of $P$.

Trivially, we can always stablish that $\mathcal{I}(\emptyset, \rho) = \emptyset$, as there are no events or relations to instantiate. The empty execution will have the final environment $\rho_X = \rho$, and its instantiation will have the environment $\mathcal{I}(\text{4}, \rho) = \rho$.

We now proceed rule-by-rule. In all cases we first name the pre-instantiation structure $((E', S', \vdash^C, \leq'), \phi)$ where $\phi(\rho)$.

**Read append**

$$\mathcal{I}(\kappa'(\rho'), \rho) = \kappa(\rho) \Rightarrow \mathcal{I}(\llbracket \texttt{ri := x} \rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket \texttt{ri := x} \rrbracket_{n\ \rho\ \kappa}$$

There is no difference in the read append rule, therefore this follows from the inductive hypothesis.

**Write Append**

$$\mathcal{I}(\kappa'(\rho'), \rho) = \kappa(\rho) \Rightarrow \mathcal{I}(\llbracket \texttt{x := ri} \rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket \texttt{x := ri} \rrbracket_{n\ \rho\ \kappa}$$

These structures are equivalent unless $\rho'(r_i) \neq \rho(r_i)$, which can only occur if $\rho'(r_i) = I_i$. In this case, the only differences are in the labelling functions $\lambda$ and $\lambda'$, and in the execution sets $S$ and $S'$. Per the definition of instantiation, in every case where $\lambda'(e) = \mathrm{W}\ x\ I_i$ the instantiation $\mathcal{I}(\lambda', \rho)$ will replace this label with $\mathrm{W}\ x\ \rho(r_i)$. As the label $\mathrm{W}\ x\ I_i$ can only occur if no statement of the form `ri := x` has been interpreted, the value $v$ must come from the initial environment, therefore $\rho(r_i)$ must be $v$ for the event $e$ to have this label in $\llbracket P \rrbracket_{n\ \rho\ \kappa}$.

To show $\mathcal{I}(S', \rho) = S$ we expand the definition of $w \bullet S$:

$$(w : \mathrm{W}\ x\ I_i) \bullet S \triangleq \{(\{w\} \cup E, \mathrm{LK}_E, \mathrm{RF}_E \cup \mathrm{RF}, \mathrm{DP}_E, \phi_E \wedge \phi) \mid (E, \mathrm{LK}_E \mathrm{RF}_E, \mathrm{DP}_E, \phi_E) \in S\}$$

Where:

$$\mathrm{RF} \in \{\{(w, r)\} \mid (r : \mathrm{R}\ x\ v') \in E \wedge \nexists e.\ w \leq e \leq_E r\}$$

$$(w, (r : \mathrm{R}\ x\ v)) \in \mathrm{RF} \Leftrightarrow (\phi(\rho) \Rightarrow \rho(r_i) = v)$$

For all coherent executions with satisfiable conditions

We know that $\mathcal{I}(S', \rho)$ will remove all executions $X$ where $\phi_X(\rho) = \bot$, thus if there remains some $X$ where $(w', e') \in \mathrm{RF}_X$ then $\mathcal{I}(e', \rho) = (e : \mathrm{R}\ x\ v)$. By $\mathcal{I}(\kappa'(\rho'), \rho) = \kappa(\rho)$, there must also be some execution equivalent to $\mathcal{I}(E)$ containing $(e : \mathrm{R}\ x\ v)$ where $\rho(r_i) = v$, thus there is some execution containing all edges in $E$ plus each edge $w \xrightarrow{\mathrm{RF}} r$.

**Sequencing**

$$\begin{aligned}\mathcal{I}(\llbracket P_1\rrbracket^T_{n\ \rho'\ \emptyset}, \rho) = \llbracket P_1\rrbracket^T_{n\ \rho\ \emptyset} \wedge \\ \mathcal{I}(\lambda\rho_2.\ \llbracket P_2\rrbracket^T_{n\ \rho_2\ \kappa'}, \rho) = \lambda\rho_2.\ \llbracket P_1\rrbracket^T_{n\ \rho_2\ \kappa} \Rightarrow \\ \mathcal{I}(\llbracket P_1;P_2\rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket P_1;P_2\rrbracket^T_{n\ \rho\ \kappa}\end{aligned}$$

This rule is identical in both models, but we cannot guarantee that continuation $\lambda\rho_2.\ \llbracket P_2\rrbracket^T_{n\ \rho_2\ \kappa'}$ is called with the same environment $\rho_2$ as the continuation $\lambda\rho_2.\ \llbracket P_1\rrbracket^T_{n\ \rho_2\ \kappa}$ if $\rho' \neq \rho$.

Given two environments $\rho_1$ and $\rho_2$, let the notation $\rho_1 \leftarrow \rho_2$ be the *overwrite* of $\rho_1$ by $\rho_2$:

$$\rho_1 \leftarrow \rho_2 = \lambda r.\ \text{if } r \in \mathit{Dom}(\rho_2) \text{ then } \rho_2(r) \text{ else } \rho_1(r)$$

If $\rho_2 = \rho \leftarrow \rho_1$ and $\rho_2' = \mathsf{4} \leftarrow \rho_1$ then $\mathcal{I}(\llbracket P\rrbracket^T_{n\ \rho_2\ \kappa}, \rho) = \mathcal{I}(\llbracket P\rrbracket^T_{n\ \rho_2'\ \kappa})$. This follows from the definition of $\mathcal{I}$: any value constructed from $\rho(r_i)$ in the second structure will be $I_i$ in the first, which is then instantiated into $\rho(r_i)$.

Taking the environment $\rho_2$ to be the environment passed to the continuation after evaluating $\llbracket P_1\rrbracket^T_{n\ \rho\ \emptyset}$, we can assert from the second inductive hypothesis that $\mathcal{I}(\llbracket P_2\rrbracket^T_{n\ \rho_2\ \kappa'}, \rho) = \llbracket P_1\rrbracket^T_{n\ \rho_2\ \kappa}$. We can also assert that $\rho_2$ must be the overwrite of the $\rho$ given to $\llbracket P_1\rrbracket^T_{n\ \rho\ \kappa}$ with some $\rho_1$ constructed by $P_1$. By the equivalence of the other rules, the environment $\rho_2'$ given to $\llbracket P_2\rrbracket^T_{n\ \rho_2'\ \kappa'}$ must be $\mathsf{4} \leftarrow \rho_1$. This means that $\mathcal{I}(\llbracket P_2\rrbracket^T_{n\ \rho_2\ \kappa'}, \rho) = \mathcal{I}(\llbracket P_2\rrbracket^T_{n\ \rho_2'\ \kappa'}) = \llbracket P_2\rrbracket^T_{n\ \rho_2\ \kappa}$, and the result follows.

**Parallel Composition**

$$\begin{aligned}&\mathcal{I}(\llbracket P_1\rrbracket^T_{n\ \rho'\ \emptyset}, \rho) = \llbracket P_1\rrbracket_{n\ \rho\ \emptyset} \wedge \mathcal{I}(\llbracket P_2\rrbracket^T_{n\ \rho'\ \emptyset}, \rho) = \llbracket P_2\rrbracket_{n\ \rho\ \emptyset} \wedge \\ &\qquad\qquad \mathcal{I}(\kappa'(\rho'), \rho) = \kappa(\rho) \Rightarrow \quad \mathcal{I}(\llbracket P_1 || P_2\rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket P_1 || P_2\rrbracket^T_{n\ \rho\ \kappa}\end{aligned}$$

Naming the structures $((E_1', S_1', \_, \leq_1'), \phi_1$ of $\llbracket P_1\rrbracket^T_{n\ \rho'\ \emptyset}$ and $(E_1, S_1, \_, \leq_1)$ of $\llbracket P_1\rrbracket_{n\ \rho\ \emptyset}$ such that $\phi_1(\rho)$ and $\mathcal{I}((E_1', S_1', \_, \leq_1'), \rho) = (E_1, S_1, \_, \leq_1)$, and likewise for $P_2$, we must show:

- $\mathcal{I}(E_1' \times E_2', \rho) = E_1 \times E_2$
- $\mathcal{I}(S_1' \times S_2', \rho) = S_1 \times S_2$

The first follows from there being no difference in product rule over labelled event structures. For the second, we repeat the reasoning from the write append rule: the only possible difference is in the contents of RF, which is only completeable when $\phi(\rho)$.

**Conditionals**

$$\begin{aligned}
&\mathcal{I}(\llbracket P_1 \rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket P_1 \rrbracket_{n\ \rho\ \emptyset} \wedge \\
&\mathcal{I}(\llbracket P_2 \rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket P_2 \rrbracket_{n\ \rho\ \emptyset} \Rightarrow \\
&\qquad\qquad \mathcal{I}(\llbracket \text{if } B \text{ then } P_1 \text{ else } P_2 \rrbracket^T_{n\ \rho'\ \kappa'}, \rho) = \llbracket \text{if } B \text{ then } P_1 \text{ else } P_2 \rrbracket^T_{n\ \rho\ \kappa}
\end{aligned}$$

This gives two structures to instantiate, but only one will have a condition $\phi$ such that $\phi(\rho)$. If $\llbracket B \rrbracket$ evaluates to true under $\rho$, the structure will be in $\llbracket P_1 \rrbracket^T_{n\ \rho'\ \kappa'}$, otherwise it will be in $\llbracket P_2 \rrbracket^T_{n\ \rho'\ \kappa'}$. In each respective case, we invoke one of our inductive hypotheses. While loops are identical, and not discussed separately.

**Justifications** This leaves us with the differences between $\vdash^C$ and $\vdash$.

For a given conditional justification edge $((C', w'), \phi)$, we know that some $w$ must exist in $E$ and that $\mathcal{I}(\{e' \mid e' \sqsubseteq w'\}, \rho) = \{e \mid e \sqsubseteq w\}$ by $\mathcal{I}(E', \rho) = E$. It follows that $F(w')$ can only differ from $F(w)$ in edges which touch symbolic $I_i$-valued write labels, and that if $(S, \phi_S) \in F(w)$ for $\phi_S \neq \top$ then there must be some forwarding operation of the form $(e_1 : \mathrm{W}\ x\ I_i) \xrightarrow{LF} (e_2 : \mathrm{R}\ x\ v)$ where $\phi_S(\rho) \Rightarrow \rho(r_i) = v$. Discarding all edges where $\phi_S(\rho)$ does not hold, for all remaining edges $\lambda(e_1) = \mathcal{I}(\lambda')(e_1)$. From the forwarding rules on concrete valued writes, and $\leq = \leq'$, this means that $(e_1 : \mathrm{W}\ x\ v) \xrightarrow{LF} (e_2 : \mathrm{R}\ x\ v)$ in the concrete valued structure.

Knowing that $F(w') = \overset{w'}{\vdash} \emptyset$ and that $\mathcal{I}(\{e' \mid e' \sqsubseteq w'\}, \rho) = \{e \mid e \sqsubseteq w\}$, we can proceed by induction over justification-up-to, using $\mathcal{I}(F(w'), \rho) = F(w)$ as a base case.

$$\mathcal{I}(\overset{E'}{\vdash} w', \rho) = \overset{E}{\vdash} w \Rightarrow \mathcal{I}(\overset{e' \bullet E'}{\vdash} w', \rho) = \overset{e \bullet E}{\vdash} w$$

Expanding justification-up-to:

$$\overset{e \bullet E}{\vdash} w = \begin{cases} \overset{E}{\vdash} w \ \cup \sum_e w & \text{if } (e : \mathrm{R}\ x\ v) \\ \overset{E}{\vdash} w & \text{otherwise} \end{cases}$$

We know that $\mathcal{I}(e', \rho) = e$ by $\mathcal{I}(E', \rho) = E$, therefore for all $((C', w'), \phi_C)$ created by $\sum_{e'} w'$, either $\phi_C(\rho) = \bot$ or $(\mathcal{I}(C'), w)$ is created by $\sum_e w$. This means that there is some $D'_1 \in \uparrow C'_1$, $D'_2 \in \uparrow C'_2$ such that $\phi_C(\rho) \Rightarrow \mathcal{I}(D'_1, \rho) \sim \mathcal{I}(D'_2, \rho)$. We can retrieve $C_1 = \mathcal{I}(C'_1, \rho)$ and $C_1 \in \overset{E}{\vdash} w$ from our inductive hypothesis, likewise for $C_2$. As we have already filtered out all edges such that $\phi_C(\rho)$ does not hold, then we must be able to find $D_1 \in \uparrow C_1$ and $D_2 \in \uparrow C_2$ such that $D_1 \sim D_2$, thus $D_1 \in \overset{e \bullet E}{\vdash w}$, as required. $\square$

## 4.6 Closure under environment

We now show that the symbolic environment $\mathsf{4}$ provides a useful abstraction of the register environment modulo refinement. First, we need to properly establish what refinement over symbolic environment structures means.

**Definition 16** (Input-Agnostic Refinement)**.** *One set of symbolic environment event structures $\mathbb{E}_1 = [\![P]\!]^T_{n\ \mathsf{4}\ \kappa}$ is a refinement of another $\mathbb{E}_2 = [\![P]\!]^T_{n\ \mathsf{4}\ \kappa}$ if and only if:*
*For every $((E_1, S_1, \vdash_1, \leq_1), \phi_1) \in \mathbb{E}_1$, there exists $((E_2, S_2, \vdash_2, \leq_2), \phi_2) \in \mathbb{E}_2$ such that $\phi_1 \Rightarrow \phi_2$ and CPP preserving embedding set $\Gamma$ such that:*

*$\forall X_1 \in S_1.\ \exists X_2 \in S_2$ and valid $\simeq \in \Gamma$ such that*

*$\phi_{X_1} \Rightarrow \phi_{X_2}$ and $X_1 \preccurlyeq_{\simeq} X_2$.*

Now we can extend this into a closure under environment proof, showing that input-agnostic refinement is sufficient to show refinement under an arbitrary input.

**Theorem 1.**

$$[\![P_1]\!]^T_{n\ \mathsf{4}\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \mathsf{4}\ \emptyset} \Rightarrow \forall \rho.\ [\![P_1]\!]^T_{n\ \rho\ \emptyset} \preccurlyeq [\![P_2]\!]^T_{n\ \rho\ \emptyset}$$

*Proof.* Given our arbitrary starting $\rho$, we know that there is a unique $((E_1, S_1, \vdash_1, \leq_1), \phi_1) \in [\![P_1]\!]^T_{n\ \mathsf{4}\ \emptyset}$ such that $\phi_1(\rho)$, and by Lemma 4 that $\mathcal{I}((E_1, S_1, \vdash_1, \leq_1), \rho)$ is $[\![P_1]\!]^T_{n\ \rho\ \emptyset}$. From refinement we can now find $((E_2, S_2, \vdash_2, \leq_2), \phi_2) \in [\![P_2]\!]^T_{n\ \mathsf{4}\ \emptyset}$. As $\phi_1 \Rightarrow \phi_2$ we know that $\phi_2(\rho)$, and that this must therefore be $[\![P_2]\!]^T_{n\ \rho\ \emptyset}$.

Taking an arbitrary $X_1$ from $\mathcal{I}(S_1, \rho)$, our goal is to find $X_2$ in $\mathcal{I}(S_2, \rho)$ and embedding $\simeq$ such that $\mathcal{I}(X_1, \rho) \preccurlyeq_{\simeq} \mathcal{I}(X_2, \rho)$. We know that $\phi_{X_1}(\rho)$ must hold, because we took it from $\mathcal{I}(S_1, \rho)$, and as there must be some $X_2$ such that $\phi_{X_1} \Rightarrow \phi_{X_2}$ we can deduce that such $X_2$ must be in $\mathcal{I}(S_2, \rho)$. Finally, as $X_1 \preccurlyeq_{\simeq} X_2$ and the definition of execution simulation is unchanged between structures, it must also be the case that $\mathcal{I}(X_1, \rho) \preccurlyeq_{\simeq} \mathcal{I}(X_2, \rho)$. $\square$

## 4.7 Refinement in the Java Causality Test Cases

We have run the calculation of refinement over all explicitly described optimisations in the Java causality test cases. Not all test cases are described in terms of optimisation chains, so we ignore the test cases which only give an allowed or forbidden behaviour and do not state which optimisation could cause this behaviour. For several of the remaining cases an expected failure occurs - optimisations based on global value range analysis are inherently unsound in arbitrary contexts, and therefore modular refinement does not apply. In these cases we instead appeal to the weaker property of *whole program* refinement. Whole program refinement uses the usual calculations for refinement, but without the need to find valid embeddings and equivalent executions for those executions of the target program which are not *complete*.

## 4.7.1 Test Cases 2 and 3

The first example of interest is test case 2:

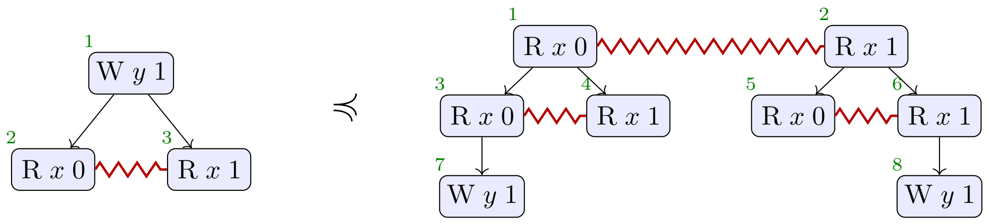
```
y := 1;      ||   r3 = y;
r1 := x;     ||   x = r3;
r2 := r1;    ||
```

$\preccurlyeq$

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

In this test case, a redundant read from $x$ is removed and replaced by a copy operation, and the subsequently redundant branch is eliminated.

Thanks to the modularity of refinement, we can restrict ourselves to only analysing the left-hand threads, as those are where the optimisation is performed. This leaves us with the following event structures:



For the target structure on the right we have two executions: the $\{1, 2\}$ execution, which we name $X_0$ after the read value, and the $\{1, 3\}$ execution, which we name $X_1$. As the write is independent, these are two executions in which a read from $x$ and store to $y$ both happen in any order, which diverge at the read. We can find similar executions in the structure on the right, those being $\{1, 3, 7\}$ and $\{2, 6, 8\}$ respectively. Note that the forwarding and coproduct rules also keep the reads and write unordered in these executions. As the definition of simluation allows us to discard reads from the source execution, we can say that $X_0$ simulates $\{1, 3, 7\}$ and $X_1$ simulates $\{2, 6, 8\}$, validating the optimisation.

Our explanation for why this case is allowed, however, differs from that of Pugh given in Pugh (2004), who argues that the optimisation should be permitted because all sequentially consistent executions of the source program may only observe 0 at both reads. The refinement relation determines that the optimisation should *always* be permitted, even if the reads could potentially observe different values, as the behaviours of the target program are always possible in the source program.

In the next case, test case 3, an additional thread is added to invalidate the original argument that both reads must be of 0. The optimisation is still permitted, even though there are complete executions of the source program which are missing from the target program.

```
y := 1;    ||  r3 := y;  ||  x := 2;
r1 := x;   ||  x := r3;  ||
r2 := r1;  ||            ||
```

$\preccurlyeq$

```
r1 := x;          ||  r3 := y;  ||  x := 2;
r2 := x;          ||  x := r3;  ||
if (r1 = r2) {    ||            ||
    y := 1;       ||            ||
}                 ||            ||
```

In this case, while a sequentially consistent execution could observe two different values at $x$, the original thread is still allowed to optimise away the branch. As the only difference is the addition of a third thread, the calculation of refinement is unchanged. This illustrates the advantages of a modular relation for program refinement – we can derive an explanation of why the optimisation occurs which will be consistent in all cases.

## 4.7.2 Test Case 6

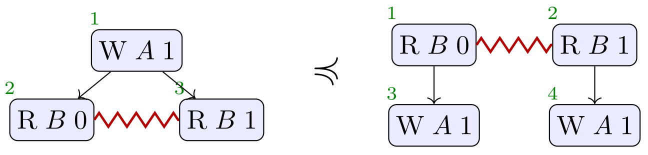
```
r1 := A;         ||  A := 1;
if (r1 = 1) {    ||  r2 := B;
    B := 1;      ||
}                ||
```

$\preccurlyeq$

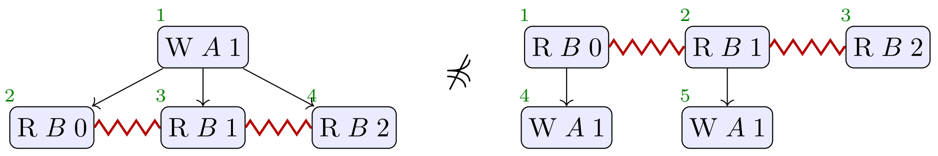
```
r1 := A;         ||  r2 := B;
if (r1 = 1) {    ||  if (r2 = 1) {
    B := 1;      ||      A := 1;
}                ||  }
                 ||  if (r2 = 0) {
                 ||      A := 1;
                 ||  }
```

In this test case, the right-hand thread is allowed to remove the branch and hoist the store above the load. This is justified by whole-program analysis: the only possible values stored to $B$ are 0 and 1, so the write must always occur.

If we fold this analysis into the run of MRD and interpret the programs with the value range $\{0, 1\}$, refinement holds in the same way as test cases 2 and 3: the program behaves as an unordered read and write.



However, if given an expanded value range (for example, $\{0, 1, 2\}$), we get a different result.



In this case, the target program can write 1 to $A$ and read 2 from $B$ in the same execution, while the source program cannot. If the program could ever run with this value range, then modular refinement does not hold.

This highlights a flaw in the modularity argument: the value range restriction is an implicit assumption required by the MRD model to produce any output at all, but value range restriction is an inherently non-modular style of analysis. While we have a modularity proof, it relies on this implicit restriction, as it assumes the context will be evaluated with the same value range as the programs. For the relation to be "truly" modular, we would need to run MRD with a value range indicative of the entire range of the integer type being used, which would produce an unusably large model for even 8-bit values. We later address this failing with the introduction of Symbolic MRD in Section 6.

### 4.7.3 Test Case 7

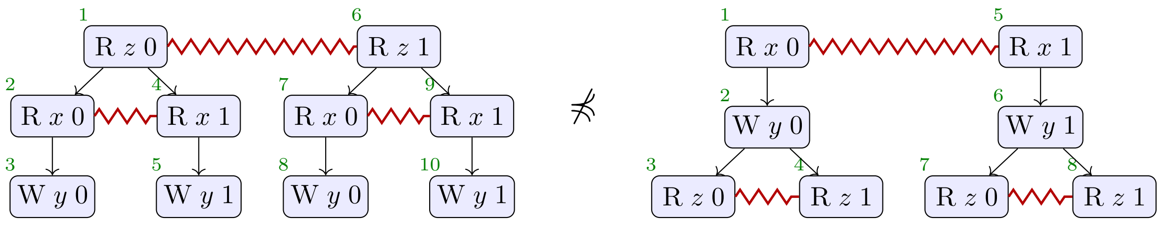
```
r1 := z;  ||  r3 := y;
r2 := x;  ||  z := r3;
y := r2;  ||  x := 1;
```

$\preccurlyeq$

```
r2 := x;  ||  x := 1;
y := r2;  ||  r3 := y;
r1 := z;  ||  z := r3;
```

In this case, both threads reorder some unrelated accesses. The thread on the left swaps the order of the reads and performs the write a line earlier, while the thread on the right moves the write to $x$ to the top.

We can no longer show refinement here, due to a flaw in the definition of least divergent points. Looking at the event structures for the left-hand threads, we can see that the source and target program diverge at different reads.



In the source program, there is one execution which reads 0 from $x$ and 0 from $z$, which we call $X_0$. There is also one execution which reads 1 from $x$ and 1 from $z$, which we call $X_1$. These executions diverge at the read from $x$, but any equivalent executions in the other thread diverge at the read from $z$. Therefore, we cannot find a common prefix preserving simulation relation.

The key observation here is that, without the register-tracking additions, the following two programs produce identical denotations:

```
r1 := x;
r2 := y;
```

$\approx$

```
r1 := x;
r1 := y;
```

The reads in the program on the left are legitimately unrelated, and can be reordered freely. In the program on the right, reordering the reads will change which is still live at the end of the fragment. Appending any write of the contents of $r_1$ after each program, the program on the left will result in a dependency from a read of $z$ while the program on the right in a dependency from a read of $x$.

If the program order relation in Definition 11 used to determine the least divergent points of two executions were replaced by the register order relation $<_r$, this test case would pass. $X_1$ and $X_2$ have two least divergent points under $<_r$, those being a read from $x$ under $<_{r_1}$ and a read from $z$ under $<_{r_2}$. However, it is not clear at this stage if this change to the definition will cause unsoundness, thus it is not presented as canonical at this time.

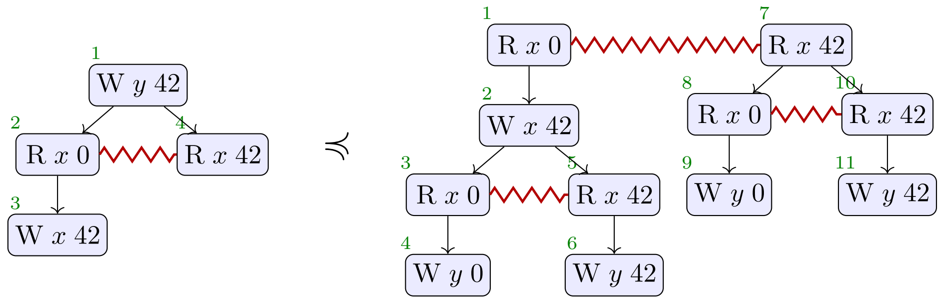
### 4.7.4 Test Case 17

```
y := 42;                    ||  r2 := y;
r3 := x;                    ||  x := r2;
if (r3 != 42) {             ||
    x = 42;                 ||
}                           ||
r1 := 42;                   ||
```

$\preccurlyeq$

```
r3 := x;                    ||  r2 := y;
if (r3 != 42) {             ||  x := r2;
    x = 42;                 ||
}                           ||
r1 := x;                    ||
y := r1;                    ||
```

This optimisation removes a redundant read from $x$ into $r_1$, replacing it with the statement `r1 := 42`, simplifies the subsequent write to $y$ into a write of a constant, and reorders the newly-unrelated events.



We can separate the target program into two executions: $X_0$ reads 0 from $x$ and contains events $\{1,2,3\}$, with dependency edge $2 \xrightarrow{\text{DP}} 3$. $X_{42}$ reads 42 from $x$ and contains events $\{1,4\}$, with no dependency edges.

In the source program, we can find an appropriate $X'_0$ with the event set $\{1,2,3,4\}$ and $X'_{42}$ with event set $\{7,10,11\}$. They diverge at reads from $x$, just as the executions of the target program do, so we now rely on MRD's dependency analysis to minimise dependencies appropriately.

$X'_0$ contains two initial justifying sets for its writes, those being $\{1,5\} \vdash 6$ and $\{1\} \vdash 2$. The dependency for event 2 matches the one found in $X_0$ if we assume $2_{X_0} \simeq 1_{X'_0}$ and $1_{X_0} \simeq 2_{X'_0}$, so all that remains is to remove the dependency for event 6. The justifying set $\{1,5\} \vdash 6$ can be simplified to $\{1,2\} \vdash 6$ via forwarding, and further simplified into $\{2\} \vdash 6$ via coproduct. This lines up with the reasoning given for the optimisation: the forwarding indicates that the read event is being replaced by taking a copy of the previously written value, in this case 42, while the coproduct indicates that the write to $y$ can be simplified to a write of a constant.

In $X'_{42}$, there is an initial dependency $\{7,10\} \vdash 11$. This can likewise be forwarded to $\{7\} \vdash 11$, and the final read can be removed by coproduct, reflecting the same reasoning. The events 3 and 10 in $X'_0$ and $X'_{42}$ respectively have no equivalents under embedding, because embeddings are still valid if they do not cover every read in the source program.

### 4.7.5 Conclusions

From these cases, we have seen that refinement does permit many optimisations that are “truly” modular and do not require whole-program analysis, but fails to recognise changes to program text permitted by whole-program analysis or to consistently reorder unrelated reads. The latter issue has a potential fix via the use of register order, while reasoning about whole-program analysis passes are beyond the intended scope of the relation.

We also illustrate some difficulties with defining when an optimisation should or should not be permitted. Programs typically have correctness criteria relative to their intended function, which is not information available to a compiler. Determining when an access is genuinely irrelevant to correctness cannot be done automatically, only when it is a repeat of a previous access. It is not necessarily feasible to use the refinement relation shown here to guide the creation of optimisation passes, as it implicitly requires this information. Similarly, libraries are typically developed with minimal or no knowledge of the programs which will use them, thus optimisations like value range analysis which require whole-program information cannot be easily anticipated by the programmer.

What refinement can do, by treating every access as observable, is permit a much wider scope of optimisations than dependency analysis alone, and reflect those optimisations directly in the events of the structure. This allows for a more destructive representation of optimisation passes – events delcared irrelevant by one pass are removed from the structure entirely, and invisible to subsequent passes. This is closer to the program view used by optimisations in practice, and may have potential for more accurately reflecting compiler behaviour than the more information-intensive style of analysis used by the dependency calculation.

In the subsequent chapter we introduce an even more intention-directed style of analysis than refinement, by folding our weak memory orderings into a program logic and using this to reason about more detailed correctness criteria.

# Chapter 5

# An MRD Based Owicki-Gries Style Logic

While refinement allows us to show that two programs are equivalent if both are refinements of each other, it cannot tell us in clear terms what those programs *do*. A user with a background in weak memory models can gain an intuition of what a small program will do by analysing its denotation, but a clear gap remains between the denotations themselves and the kind of condition pairs or contracts used in traditional program verification.

In this section we aim to bridge that gap by extracting the orderings provided by a weak memory model and using them to inform a Hoare logic style program correctness proof.

The main contribution here is the construction of a program logic which can validate Hoare triples over a potentially nondeterministic semantics. The existing work of Doherty et al. (2019), which we use as a foundation, relies on the C11 happens-before relation as an evaluation order. The authors note that continuing progress in the field of weak memory models includes the removal of a thread-local total ordering relation such as happens-before, and that the size and complexity of the resulting models creates new challenges for the construction of program logics. The separation logics of CSL (Concurrent Separation Logic) and FSL (Fenced Separation Logic) of Doko and Vafeiadis (2016) target weak memory programs, but require release-acquire synchronisation to make assertions about observable values to avoid tackling nondeterministic execution order in the logics directly.

With this logic we chose to take a different approach. Instead of using modalities to describe the amount of knowledge we can reasonably have about memory state per line of syntax, as in FSL, we first define a nondeterministic operational semantics, and then build a corresponding program logic which does not associate pre- and postconditions to locations in syntax, but to potential stages of execution. While we describe this method using the dependency relation created MRD, the technique is applicable to any event structure-based memory model.

## 5.1 Hoare Logic and the Owicki-Gries Method

A proof of program correctness using Hoare logic attaches a pre- and postcondition to each statement and uses an operational semantics to validate that each statement does indeed maintain its postcondition if it executes in an environment which satisfies its precondition. These precondition-statement-postcondition triples are written $\{Q\}\ P\ \{R\}$, meaning "if program $P$ executes in an environment where $Q$ holds, then $R$ will hold for any environment which results".

These triples are verified using an operational semantics, which represents the program environment as some mathematical abstraction and calculates the impact a statement will have on its input program environment. The truth of the property $Q$ can then be established for an abstract program environment, the transformation on that environment corresponding to $P$ performed by the operational semantics, and the property $R$ checked over the result.

We can illustrate this with a toy eval/apply language in which all statements compute an expression over the natural numbers and store the result in the environment $\Gamma$, which we represent as a function from variables $Var$ to $\mathbb{N}$. The apply rule updates the environment $\Gamma$, while the eval rule evaluates expressions to numerical values.

$$\text{Apply}\,\frac{eval(expr, \Gamma) = v}{\Gamma \xrightarrow{x:=expr} \Gamma[x \mapsto v]}$$

$$eval(n : \mathbb{N}, \Gamma) = n \qquad eval(v : Var, \Gamma) = \Gamma(v)$$

$$\begin{aligned} eval(e_1 + e_2, \Gamma) &= eval(e_1, \Gamma) + eval(e_2, \Gamma) \\ eval(e_1 - e_2, \Gamma) &= eval(e_1, \Gamma) - eval(e_2, \Gamma) \\ eval(e_1 \times e_2, \Gamma) &= eval(e_1, \Gamma) \times eval(e_2, \Gamma) \end{aligned}$$

Now for any property of the values stored in variables, such as $\{x = y\}$, we can determine by substitution whether or not some environment $\Gamma$ satisfies that property. To create a Hoare triple around some statement $s$, we first assume that the precondition $Q$ holds over the environment before we execute that statement, and then show that property $R$ must hold after executing it.

$$\text{Step}\,\frac{\forall \Gamma.\ (Q(\Gamma) \wedge \Gamma \xrightarrow{s} \Gamma') \Rightarrow R(\Gamma')}{\{Q\}\ s\ \{R\}}$$

The logic extends from statements to whole programs using the sequential composition operator. If we begin with precondition $Q$ and execute program $P_1$ to gain postcondition $R$, then execute program $P_2$ with precondition $R$ to gain postcondition $S$, then $\{Q\}\ P_1; P_2\ \{S\}$.

$$\textsc{Sequence}\frac{\{Q\}\ P_1\ \{R\} \qquad \{R\}\ P_2\ \{S\}}{\{Q\}\ P_1; P_2\ \{S\}}$$

What it cannot do is reason about the parallel composition of programs. It is *not* the case that if $\{Q\}\ P_1\ \{R\}$ and $\{R\}\ P_2\ \{S\}$, then $\{Q\}\ P_1\ ||\ P_2\ \{S\}$, as this would be a valid result:

$$\{x = 0\}$$
```
x := 1;   ||   x := x + 1;
```
$$\{x = 2\}$$

Since it is entirely possible for the right-hand thread to execute first, resulting in both threads writing 1 to $x$, our postcondition does not always hold.

We cannot simply treat the two threads as separate programs with the same precondition either – if $\{Q\}\ P_1\ \{R\}$ and $\{Q\}\ P_2\ \{S\}$, then we cannot assume $\{Q\}\ P_1\ ||\ P_2\ \{R \wedge S\}$, as this can result in unsatisfiable postconditions:

$$\{x = 0\}$$
```
x := 1;   ||   x := 2;
```
$$\{x = 1 \wedge x = 2\}$$

The Owicki-Gries method safely establishes a rule for parallel composition in a program logic which forbids unsatisfiable and insufficient postconditions. It uses the $\{Q\}\ P_1\ ||\ P_2\ \{R \wedge S\}$ approach, but only allows the rule to be used if the proofs of $\{Q\}\ P_1\ \{R\}$ and $\{Q\}\ P_2\ \{S\}$ satisfy a condition called *interference freedom*.

Two proofs over two statements, $\{Q_1\}\ s_1\ \{R_1\}$ and $\{Q_2\}\ s_2\ \{R_2\}$, are interference-free if they cannot cause each other's preconditions or postconditions to become false. This means that all of the following must hold:

1. $\{Q_1 \wedge Q_2\}s_1\{Q_2\}$
2. $\{Q_1 \wedge Q_2\}s_2\{Q_1\}$
3. $\{Q_1 \wedge R_2\}s_1\{R_2\}$
4. $\{Q_2 \wedge R_1\}s_2\{R_1\}$

In the example above, both statements have precondition $\{x = 0\}$, but neither statement preserves that condition:

$$\begin{array}{lcl}
\{x = 0 \wedge x = 0\} & & \{x = 0 \wedge x = 0\} \\
\texttt{x := 1;} & || & \texttt{x := 2;} \\
\{x = 0\} & & \{x = 0\}
\end{array}$$

The proof must be rewritten with preconditions and postconditions that are weaker, but always preserved:

$$\begin{array}{c}
\{x = 0\} \\
\begin{array}{lcl}
\{x = 0 \vee x = 2\} & & \{x = 0 \vee x = 1\} \\
\texttt{x := 1;} & || & \texttt{x := 2;} \\
\{x = 1 \vee x = 2\} & & \{x = 1 \vee x = 2\}
\end{array} \\
\{x = 1 \vee x = 2\}
\end{array}$$

This extends to whole programs pairwise: to compose program $P_1$ with $P_2$, each individual precondition-statement-postcondition sequence in $P_1$ must not interfere with any such sequence in $P_2$ and vice-versa.

**Definition 17** (Full Interference Freedom)**.** *$\{Q\}\ P_1\ \{R\}$ and $\{Q\}\ P_2\ \{S\}$ are interference-free if, for any sub-program $\{Q_i\}\ P_i\ \{R_i\}$ of $P_1$ and sub-program $\{Q_j\}\ P_j\ \{R_j\}$ of $P_2$, $\{Q_i \wedge Q_j\}\ P_i\{Q_j\}$ and $\{Q_i \wedge R_j\}\ P_i\{R_j\}$, and vice versa for $P_j$.*

This method allows us to perform Hoare logic verifications of multi-threaded programs, but the logic still composes verifications of individual program fragments using sequential composition. According to the sequential composition rule, every statement assumes that all preceeding statements have finished executing and that all subsequent statements have yet to execute. As we have shown, this is no longer a reasonable abstraction for a concurrent program. Lahav and Vafeiadis (2015) show that even with auxiliary variables, standard Owicki-Gries reasoning is unsuitable for weak memory contexts. The method has been extended to programs with *release/acquire synchronisation* (Dalvandi et al. (2020)), but has yet to be applied to fully relaxed (i.e. unsynchronised) memory accesses. Our task is now to replace the sequential composition rule with one that better reflects the realities of concurrent program execution.

### 5.1.1 A Semicolon-Free Hoare Logic

The first step is to separate out the syntactic program order operator, the semicolon, from the actual execution order. We can do this using the program counter. If a statement executes at program counter $i$ and finishes with program counter $j$, then we can think of its precondition as being the condition associated with PC value $i$ and its postcondition as being the condition associated with PC value $j$. We can write this as $\{Q\}_i\ P\{R\}_j$. If a statement can execute with two potential PC values, it needs two such triples. If it can execute with three, it needs three triples, and so on.

We can now represent a program as being an unordered set of statements, provided we use an operational semantics that correctly increments the program counter. We can ensure this for our toy semantics by adding a program counter to our environment and pushing the interpretation of the semicolon down into it.

$$\textsc{Apply PC}\,\frac{eval(expr, \Gamma) = v}{(\Gamma, i) \xrightarrow{x:=expr} (\Gamma[x \mapsto v], i+1)}$$

$$\textsc{Operational Sequence}\,\frac{(\Gamma_1, i) \xrightarrow{P_1} (\Gamma_2, j) \qquad (\Gamma_2, j) \xrightarrow{P_2} (\Gamma_3, k)}{(\Gamma_1, i) \xrightarrow{P_1;P_2} (\Gamma_3, k)}$$

We can use this to introduce a simple parallel composition rule, which interrupts one thread and allows the other to continue while incrementing the shared program counter. This includes an assumption of sequential consistency, but is a first step towards a weak memory capable operational semantics.

$$\textsc{Operational Parallel}\,\frac{\begin{array}{c}(\Gamma_1, i) \xrightarrow{P_1} (\Gamma_2, j) \qquad (\Gamma_2, j) \xrightarrow{P_2} (\Gamma_3, k)\\ (\Gamma_3, k) \xrightarrow{P_1'||P_2'} (\Gamma_4, n)\end{array}}{(\Gamma_1, i) \xrightarrow{(P_1;P_1')||(P_2;P_2')} (\Gamma_4, n)}$$

We also introduce a small skip rule to handle the case where $P_1'$ and $P_2'$ are now empty.

$$\textsc{Terminate Parallel}\,\frac{}{(\Gamma, i) \xrightarrow{\emptyset||\emptyset} (\Gamma, i)}$$

Now our *operational semantics* is locked to program order, but our logic no longer needs to concern itself with how the semicolon operator works. In fact, the logic no longer needs to understand how statement ordering of any kind works.

Instead of associating conditions with statements, we can associate them with program counter values. As long as a statement is *reachable* by program counter value $n$, then it may take the condition associated with value $n$ as a precondition and the condition associated with value $n+1$ as a postcondition. We choose to represent reachability using the transition relation. If a statement $s$ inside program $P$ appears inside a trace of $P$ in the operational semantics with counter value $i$ then $i$ appears in the set $PC_P(s)$.

$$i \in PC_P(s) \Leftrightarrow (\Gamma_0, 0) \xrightarrow{P} (\Gamma_1, n) \wedge \exists P_{pre}, P_{post}.\ (\Gamma_0, 0) \xrightarrow{P_{pre}} (\Gamma, i) \xrightarrow{s} (\Gamma', i+1) \xrightarrow{P_{post}} (\Gamma_1, n)$$

We choose to map counter values to conditions by establishing an invariant as a function, taking counter values and returning conditions. Taking invariant $I$ to have type $PC \rightarrow env \rightarrow pred$, calling $I(n)$ for some PC value $n$ gives a predicate over environments, and $I(n)(\Gamma)$ for some environment $\Gamma$ returns a predicate. We can then say that a statement *preserves* an invariant if it allows a proof of $I(n+1)$ given precondition $I(n)$, for all counter values $n$ at which the statement may execute.

$$\textsc{IStep}\ \frac{\forall n \in PC_P(s).\ (I(n)(\Gamma) \wedge (\Gamma, n) \xrightarrow{s} (\Gamma', n+1)) \Rightarrow I(n+1)(\Gamma')}{\{I\}\ P\ \{I\}}$$

We can also express this rule more simply as a repeated application of the Step rule:

$$\textsc{IStep}\ \frac{\forall n \in PC_P(s).\ \{I(n)\}s\{I(n+1)\}}{\{I\}\ P\ \{I\}}$$

Finally, we can change our representation of the program text from an ordered sequence to an unordered set. Our sequencing rule now takes the union of the executed statements, allowing it to represent both sequential and parallel composition.

$$\textsc{ISequence}\ \frac{\{I\}\ S_1\ \{I\} \qquad \{I\}S_2\{I\}}{\{I\}S_1 \cup S_2\{I\}}$$

For a purely sequential program $P$, the set $PC_P(s)$ for any $s$ will contain only a single value.

| | |
|---|---|
| `x := 1;` | $PC_P(x := 1) = 0$ |
| `x := 2;` | $PC_P(x := 2) = 1$ |
| `x := 3;` | $PC_P(x := 3) = 2$ |

We can then construct an invariant $I$ which corresponds to a standard Hoare logic proof of our target postcondition.

$$I(0) = \top \quad I(1) = \{x = 1\} \quad I(2) = \{x = 2\} \quad I(3) = \{x = 3\}$$

The only novelty here is that we are free to construct the final proof in any order we choose. We could, for instance, show that the first and last lines preserve the invariant, and then add the middle line.

$$\textsc{ISequence}\,\dfrac{\textsc{IStep}\,\dfrac{\{\top\}\texttt{x := 1}\{x=1\}}{\{I\}x := 1\{I\}} \quad \textsc{IStep}\,\dfrac{\{x=2\}\texttt{x := 3}\{x=3\}}{\{I\}x := 3\{I\}}}{\{I\}x := 1 \cup x := 3\{I\}}$$

As long as all statements preserve the invariant $I$, then we can treat $I(0)$ as our precondition and $I(3)$ as our postcondition and derive a proof equivalent to $\{I(0)\}P\{I(3)\}$.

However, we can now represent non-linear execution sequences in our operational semantics and propagate the results directly into the program logic.

```
x := 1;   ||   x := 3;
x: = 2;
```

$$PC_{P'}(x := 1) = \{0,1\}$$
$$PC_{P'}(x := 2) = \{1,2\}$$
$$PC_{P'}(x := 3) = \{0,1,2\}$$

In this version of $P$, the PC value 1 indicates that either `x := 1` or `x := 3` may have executed. Likewise, at PC value 2, any two statements may have executed except for the combination of `x := 2` and `x := 3`. As a result, our invariant needs to be much weaker:

$$I(0) = \top \quad I(1) = \{x = 1 \vee x = 3\} \quad I(2) = \{x = 1 \vee x = 2 \vee x = 3\}$$
$$I(3) = \{x = 2 \vee x = 3\}$$

And, as each statement has multiple PC values, we must prove more Hoare triples. The statement requiring the most sub-proofs to show invariant preservation is `x := 3`:

$$\textsc{IStep}\,\dfrac{\{I(0)\}\texttt{x := 3}\{I(1)\} \quad \{I(1)\}\texttt{x := 3}\{I(2)\} \quad \{I(2)\}\texttt{x := 3}\{I(3)\}}{\{I\}\texttt{x := 3}\{I\}}$$

However, having effectively enumerated all possible execution paths, we do not need to show interference freedom over $I$. This is because we have effectively completed *all* of the following proofs:

| | | |
|---|---|---|
| $\{I(0)\}$ | $\{I(0)\}$ | $\{I(0)\}$ |
| `x := 1` | `x := 1` | `x := 3` |
| $\{I(1)\}$ | $\{I(1)\}$ | $\{I(1)\}$ |
| `x := 2` | `x := 3` | `x := 1` |
| $\{I(2)\}$ | $\{I(2)\}$ | $\{I(2)\}$ |
| `x := 3` | `x := 2` | `x := 2` |
| $\{I(3)\}$ | $\{I(3)\}$ | $\{I(3)\}$ |

Therefore, using the assumption of sequential consistency that we baked into the logic via the use of a shared environment and program counter, we have covered every possible execution of the program.

For a program which uses top-level parallelism, this is an inferior approach to the Owicki-Gries method. Determining the invariant is more complex, and the proof burden is comparable. What we use this method for, however, is to represent non-linear execution of a single thread under a weak memory model.

As we have seen previously, weak memory models often depict statements as *partially* ordered. For sections of the program which are effectively sequential, as we haveshown, the indexed invariant adds no proof burden overhead. For particular statement pairs which may occur in any order, using the proper index into our invariant allows us to create a minimal set of pre- and post-conditions for each statement and avoid any new interference freedom proofs.

In the following section we introduce this new index, which we call *program futures*, and show that it correctly represents the ordering guarantees of MRD.

## 5.2 An Operational Semantics With MRD

While a program counter can tell us how many statements have executed, it cannot tell us what those statements are. It could tell us that, for instance, 3 lines of code have executed, but not which 3 lines. Something like a whole execution trace could accomplish the opposite extreme: it could tell us precisely which statements happened in which order, even those whose order of execution is irrelevant to a correctness argument.

Program futures represent a midpoint between these extremes. A program future contains the events that will execute in future, as well as the partial order over those events indicating which event pairs must stay in order. We use these futures to guide our operational semantics in a style similar to the "Bubbly" model discussed in Section 2.4.3, as a "to-do" list. The main difference is that we first run the entire MRD model to extract individual futures, and then execute these futures in the operational semantics, with no interaction between the execution and modelling of the program.

### 5.2.1 Program Futures

We write the program futures as a pair $(E, \preceq)$ containing an event set $E$ and relation $\preceq$. An event $e$ is *available* in future $f$ if it is maximal in $\preceq$, and an available event can be removed from a future with the $\triangleright$ operator. This can be thought of as executing $e$, and is used as such in the operational semantics.

$$e \triangleright (E, \preceq) = (E', \preceq|_{E'}) \qquad E' = E \setminus \{e\}$$

The $\triangleright$ operator can be lifted to a set of futures $F$, where it not only executes $e$ in all futures in $F$ where it is available, but also removes from the set any futures where $e$ is unavailable.

$$e \triangleright F = \{e \triangleright f \mid f \in F \land a \text{ available in } f\}$$

To continue the comparison with the "Bubbly" model, this is similar to the execution of $e$ discarding all events in conflict with $e$. If we execute $e$, then we must discard all futures which do not contain $e$.

We convert an MRD execution into a future using the $\varphi$ function, which maintains the event set $E$ and takes the union of $\textsc{dp}$ and $\leq$ to be the ordering relation $\preceq$.

$$\varphi(E, \textsc{rf}, \textsc{lk}, \textsc{dp}) = (E, \textsc{dp} \cup \leq_{|E})$$

We do not represent $\textsc{rf}$ in the program futures, as the operational semantics maintains enough information to determine when a read may or may not be executed.

The initial future set for a program $P$ takes the MRD denotation $C = (L, S, \vdash, \leq)$ of $P$ and produces a future for each execution (though it is possible for multiple executions to produce the same future if they differ only in $\textsc{rf}$).

$$F_P = \bigcup_{X \in S} \{\varphi(X)\}$$

### 5.2.2 Abstract Machine and Transition Relation

While we can now guide our operational semantics and index our predicates using program futures, the inclusion of weakly ordered memory makes writing the predicates themselves far more challenging. Our predicates will require some representation of memory state, which is a feature that MRD intentionally avoids. It is difficult to define a semantics for a statement like $\{x = 1\}$ under weak memory concurrency – does this mean that a future read *may* observe this value, or that it can *only* observe this value? And, in either case, how can we maintain enough information in our operational state to check?

In Doherty et al. (2019), the authors define a representation of state which both tracks committed writes and determines whether or not these writes may be observed, based on *tagged action graphs*. These graphs are similar to event structures, where the term "tagged action" describes an object similar to an event, but here they are used to represent a memory state rather than the entire computation. We build upon this work, using tagged action graphs as our state representation. Our predicates will refer to properties of tagged action graphs where predicates over sequentially consistent programs would use variable-to-value maps.

A tagged action is a triple $(g, a, t)$ consisting of a unique identifier $g$, an action (referred to in MRD as a label) $a$, and a thread identifier $t$.

We write our tagged action graphs using the tuple

$$(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo})$$

Where:

- $D$ is a set of events
- $\mathsf{sb}$ is the *sequenced before* relation, such that $(e_1, e_2) \in \mathsf{sb}$ if $e_1$ and $e_2$ are in the same thread and $e_1$ executed first
- $\mathsf{rf}$ is the reads from relation, obeying the usual definition
- $\mathsf{mo}$ is the *modification order* relation, such that $(e_1, e_2) \in \mathsf{mo}$ if $e_1$ and $e_2$ are writes to the same variable and $e_1$ executed first

The combination of these relations allows us to define when the value of a comitted write may be observable to a read:

1. The write cannot have since been overwritten by another write to the same location in the same thread
2. The current thread cannot have read from a write which was committed more recently

We do this check in two stages. First, we list the *encountered writes* of a thread. The set of writes thread $t$ has encountered in state $\sigma$ is defined by the set $EW_\sigma(t)$, which contains all writes related to some action in $t$ by $\mathsf{rf}$ or $\mathsf{mo}$, and also those related to an action which is sequenced-before an action in $t$ (such as initialisations, which occur in thread 0 but are sequenced-before all other actions):

$$EW_\sigma(t) = \{w \in \mathsf{Wr} \cap D \mid \exists b \in D_\sigma.\ tid(b) = t \wedge (w, b) \in (\mathsf{rf} \cup \mathsf{mo})^*; ltsb^*\}$$

The set of *observable writes* for thread $t$ in state $\sigma$ is the set of all writes in $\sigma$ which are not $\mathsf{mo}$-before an encountered write, and is given by $OW_\sigma(t)$:

$$OW_\sigma(t) = \{w \in \mathsf{Wr} \cap D \mid \forall w' \in EW_\sigma(t).\ (w, w') \notin \mathsf{mo}_\sigma\}$$

$$\textsc{Read}\ \frac{\begin{array}{c} b = (g, a, t) \qquad g \notin \mathit{tags}(D) \qquad a = \text{i:R}\ x\ n \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \qquad w \in \mathit{OW}_\sigma(t) \qquad \mathit{loc}(w) = x \qquad \mathit{val}(w) = n \end{array}}{(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \overset{b}{\rightsquigarrow} (D \cup \{b\}, \mathsf{sb} +_D b, \mathsf{rf} \cup \{(w, b)\}, \mathsf{mo})}$$

$$\textsc{Write}\ \frac{\begin{array}{c} b = (g, a, t) \qquad g \notin \mathit{tags}(D) \qquad a = \text{i:W}\ x\ n \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \qquad w \in \mathit{OW}_\sigma(t) \qquad \mathit{loc}(w) = x \end{array}}{(D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \overset{b}{\rightsquigarrow} (D \cup \{b\}, \mathsf{sb} +_D b, \mathsf{rf}, \mathsf{mo}[w, b])}$$

Figure 44: Operational rules for adding reads and writes to a tagged action graph.

$$\frac{n = \mathit{eval}(e, \gamma(t)) \quad a = \text{i:W}\ x\ n}{(i\colon [x] := e, \gamma) \xrightarrow{a}_t (\textsf{skip}, \gamma)} \qquad\qquad \frac{a = \text{i:R}\ x\ n \quad \rho' = \gamma(t)[r := n]}{(i\colon r := [x], \rho) \xrightarrow{a}_t (\textsf{skip}, \gamma[t := \rho'])}$$

Figure 45: Thread-local operational semantics.

We give the update rules for the operational state in Fig. 44, using the notation $\sigma \overset{b}{\rightsquigarrow} \sigma'$ to denote state transition.

We also use a thread-local semantics to maintain a thread-local register environment, generate actions, and evaluate expressions. This links the program text to the tagged action graph.

## Future-Guided Transition Relation

With a semantics for individual statements in place, we now use program futures to decompose the program into valid sequences of operational steps.

We give this semantics over the tuple

$$(\overline{Q}, (\sigma, \gamma), F)$$

Where:

- $\overline{Q}$ is the program text of $Q$ represented as an unordered set of statements
- $(\sigma, \gamma)$ is a pair containing a tagged action graph $\sigma$ and a thread state $\gamma$
- $F$ is a set of futures

Whenever an event is available in a future, we extract its label, construct an action with it, and take a step in the memory semantics and thread-local semantics.

$$\textsc{Future-Step}\ \frac{\begin{array}{c} \overline{Q}(t) = \overline{C} \uplus \{i\colon\ s\} \qquad a \triangleright F \neq \emptyset \\ (i : s, \gamma) \xrightarrow{a}_t (\textsf{skip}, \gamma') \qquad \sigma \overset{(g,a,t)}{\rightsquigarrow} \sigma' \end{array}}{(\overline{Q}, (\sigma, \gamma), F) \longrightarrow (\overline{Q}[t := \overline{C}], (\sigma', \gamma'), a \triangleright F)}$$

### 5.2.3 Example

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

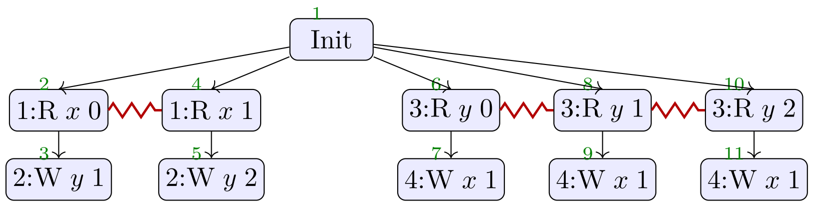


Figure 46: Load buffering with a semantic dependency

We give an example program and its event structure in Fig. 46.

Let $\Delta_S = \{(x, x) \mid x \in S\}$ be the diagonal of set $S$. We first derive our set of futures from this structure:

$$\begin{array}{ccc} \{2 \prec 3, 6, 7\} & \{2 \prec 3, 8, 9\} & \{2 \prec 3, 10, 11\} \\ \{4 \prec 5, 6, 7\} & \{4 \prec 5, 8, 9\} & \{4 \prec 5, 10, 11\} \end{array}$$

where $\{2 \prec 3, 6, 7\}$ represents the future $(\{2, 3, 6, 7\}, 2 \prec 3 \cup \Delta_{\{2,3,6,7\}})$. The atomic set of the program is

$$\overline{P} = \left\{ \begin{array}{l} 1 \mapsto \{\mathtt{1} : \mathtt{r1} := [\mathtt{x}], \mathtt{2} : [\mathtt{y}] := \mathtt{r1} + \mathtt{1}\}, \\ 2 \mapsto \{\mathtt{3} : \mathtt{r2} := [\mathtt{y}], \mathtt{4} : [\mathtt{x}] := \mathtt{1}\} \end{array} \right\}$$

The initial configuration is $(\sigma_0, \gamma_0)$, where $\sigma_0 = (\{(0_x, 0{:}\text{W }x\ 0, 0), (0_y, 0{:}\text{W }y\ 0, 0)\}, \emptyset, \emptyset, \emptyset)$ and $\gamma_0 = \{1 \mapsto \{r1 \mapsto 0\}, 2 \mapsto \{r2 \mapsto 0\}\}$, assuming the initialising thread has identifier 0.

To find out which events we can execute, we check our futures set. We cannot execute events 3 or 5, which both have pre-requisite events that have not yet been executed. These events are the only available events generated by line 2, so we cannot attempt to execute line 2. We can, however, execute any other line. Suppose we execute line 4, which corresponds to $\mathtt{4} : [\mathtt{x}] := \mathtt{1}$ in $\overline{P}(2)$. To use the transition relation $\longrightarrow$, we need to do the following: (1) determine the corresponding action, $a$ and new thread-local state using $\xrightarrow{a}_2$; (2) generate a tagged action $b = (g, a, 2)$ for a fresh tag $g$ and a new global state using $\overset{b}{\rightsquigarrow}$; and (3) check that $a$ is available in the current set of futures.

For (1), we can only create one action $a = 4{:}\text{W }x\ 1$ and the local state is unchanged. For (2), we generate a new global state using the WRITE rule in Fig. 44 (full details elided). For (3), we take our candidate futures $a \triangleright F$ by examining which MRD events have the label $4{:}\text{W }x\ 1$ — in this case events 7, 9, and 11. All futures can execute one of these events so the new future set contains all futures in $F$, each minus the set $\{7, 9, 11\}$.

$$\textsc{Tracked Read}\ \frac{\begin{array}{c} b = (g, a, t) \qquad g \notin tags(D) \qquad a = \text{i:R}\ x\ n \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \qquad w \in OW_\sigma(t) \qquad loc(w) = x \qquad val(w) = n \end{array}}{\hat{\sigma} \overset{a}{\rightsquigarrow}_t \hat{\sigma}[\mathsf{rf} \mapsto \mathsf{rf} \cup \{(w, a)\}, reads \mapsto reads \cup \{a\}]}$$

Figure 47: Transition relation with rf tracking

### 5.2.4 Equivalence between MRD and the operational semantics

We now establish equivalence of our operational semantics and the MRD denotational semantics.

We first record some more information in our operational state, in particular which writes are observed by reads. We name this the reads from relation rf, and write that $w\textsc{rf}r$ if a write $w$ stores the value observed by read $r$.

To disambiguate the standard semantics from the tracked-read semantics, we use $\hat{\sigma}$ to refer to states of the tracked read semantics.

This is the semantics that we prove equivalent to the MRD model. The proof of equivalence is separated into two parts: first we show that any execution produced by the operational semantics is a valid execution under the event structure semantics, and then we show that ay event-structure execution can be produced by the operational semantics.

**Theorem 2** (Soundness and Completeness)**.** *Every execution generated by the MRD model can be generated by the operational semantics, and every final state generated by the operational semantics corresponds to a complete execution of the MRD semantics.*

We define this correspondence between MRD executions and states of the operational semantics, using the notation $\sim$, as follows: given a bijection $\Lambda(e) = (g, \lambda(e), tid(e))$ which constructs a tagged action from an event, $(E, \textsc{rf}, \textsc{dp}, \textsc{rf}) \sim \hat{\sigma}$ iff $\Lambda(E) = \hat{\sigma}.D \wedge \Lambda(\textsc{rf}) = \hat{\sigma}.\mathsf{rf}$.

We establish these properties in Lemma 5 and Lemma 6, respectively.

#### All operational executions are event-structure executions

Here we show that every state reachable in the operational semantics corresponds to an execution of the MRD model. The existence of an execution with a corresponding set of events follows from the conversion of executions into program futures (via the $\varphi_X$ function) and the requirement that we follow a program future whenever we take an operational step. We then show that every rf relation built by the operational semantics is covered by an $\textsc{rf}$ relation in a corresponding execution in MRD.

**Lemma 5.** *Given some program $P$ such that*

$$(\overline{P}, (\hat{\sigma}_0, \gamma_0), F) \ \Rightarrow^* ((\lambda t.\ \emptyset), (\hat{\sigma}, \gamma), \{\emptyset\})$$

*over the tracked read semantics. There exists an execution* $(E, \textsc{lk}, \textsc{rf}, \textsc{dp})$ *in* $[\![Init; P]\!]_{n\ \rho_0\ \emptyset}$ *such that* $(E, \textsc{lk}, \textsc{rf}, \textsc{dp}) \sim \hat{\sigma}$.

*Proof.* Let $D = \hat{\sigma}.D$, $\mathsf{rf}_D = \hat{\sigma}.\mathsf{rf}$ and $f \in F$ be the future that we followed in the operational execution. By definition of $F$, we must have that $f = \varphi_X$ for some execution $X = (E, \textsc{lk}, \textsc{rf}, \textsc{dp})$ and $\Lambda(E) = D$.

We now show by contradiction that there must be some execution which has an $\textsc{rf}$ relation such that $\Lambda(\textsc{rf}) = \mathsf{rf}_D$. To arrive at a complete $\mathsf{rf}_D$ relation which does not correspond to any $\textsc{rf}$, there must either be an edge in $\mathsf{rf}_D$ which is absent from all $\textsc{rf}$ or an edge in $\textsc{rf}$ which is absent from $\mathsf{rf}_D$.

We know that MRD exhaustively generates $\textsc{rf}$ edges which do not create a cycle in $\textsc{rf} \cup \textsc{dp} \cup \leq$. There cannot be an edge in all possible $\textsc{rf}$ relations which is absent from $\mathsf{rf}_D$, as this would imply that $\mathsf{rf}_D$ is incomplete, i.e., there exists a read in $D$ not in the range of $\mathsf{rf}_D$. This is forbidden by the read rule: every read event in $D$ must have a corresponding edge in $\mathsf{rf}_D$.

If an edge is absent from all $\textsc{rf}$, it must always cause a $\textsc{rf} \cup \textsc{dp} \cup \leq$ cycle. Given that futures are ordered by $\textsc{dp} \cup \leq$ and the definition of availability within a future only allows us to execute events in this order, we can only generate $\mathsf{rf}_D$ edges from earlier writes to later reads. In order to create a cycle, however, we would need to relate a later write to an earlier read. This is forbidden by the availability requirement, thus there must be some $\mathsf{rf}$ which is equal to $\mathsf{rf}_D$. $\square$

### All event structure executions are represented by operational semantics executions

Here we show that, given an execution of the MRD model, we can find an operational state after stepping through the program which contains the same events with the same reads-from edges. We do this by establishing a total order over the events in the execution, and proving inductively that we are able to take operational steps in this order and that every such step results in an operational state corresponding to a restriction of the MRD execution.

In the following, we define the restriction of $X$ to event set $B$, given $X = (E, \textsc{lk}, \textsc{dp}, \textsc{rf})$ and $B \subseteq A$, as $X_{|B} = (B, \textsc{lk}_{|B}, \textsc{dp}_{|B}, \textsc{rf}_{|B})$.

**Lemma 6.** *Let* $X = (E, \textsc{lk}, \textsc{rf}, \textsc{dp})$ *be a complete execution of* $[\![Init; P]\!]_{n\ \rho_0\ \emptyset} = (L, S, \vdash, \leq)$ *and* $\xrightarrow{R}$ *be the relation* $\textsc{dp} \cup \textsc{rf} \cup \leq$. *Then both of the following hold:*

1. *There exists a relation* $\xrightarrow{R'}$ *disjoint from* $\xrightarrow{R}$ *such that* $\xrightarrow{R \cup R'}$ *is total and acyclic over* $E$, *giving the chain* $e_1 \xrightarrow{R \cup R'} e_2 \xrightarrow{R \cup R'} \cdots \xrightarrow{R \cup R'} e_k$.
2. *There exists an execution of the operational semantics* $\hat{\sigma}_0 \overset{\Lambda(e_1)}{\rightsquigarrow} \hat{\sigma}_1 \overset{\Lambda(e_2)}{\rightsquigarrow} \cdots \overset{\Lambda(e_k)}{\rightsquigarrow} \hat{\sigma}_k$ *over* $P$ *such that* $X_k \sim \hat{\sigma}_k$, *where* $X_k = X_{|\{e_1, e_2, \ldots e_k\}}$.

*Proof.* To begin, we capture ordering implicit in $R$ from reads to writes at the same location in the *operational from-reads* relation $\mathsf{ofr}$:

$$\mathsf{ofr} \triangleq \{(r : \mathrm{R}\ x\ v_1, w : \mathrm{W}\ x\ v_2) \mid \exists e.\ e \xrightarrow{\text{RF}} r \wedge e \xrightarrow{R \setminus \{(e,r)\} \cup \mathsf{ofr}}{}^{*}\ w\}$$

This means that if $w_1 \xrightarrow{R \cup \mathsf{ofr}}{}^{*} w_2$ and $w_1 \xrightarrow{\text{RF}} r$, then $(r, w_2) \in \mathsf{ofr}$, illustrated in Fig. 48. Intuitively, a read will always be $\mathsf{ofr}$-before any write to the same location which must occur after the write from which it read.

$$\begin{tikzcd} e \arrow[dd, "(R \setminus \{(e,r)\} \cup \mathsf{ofr})^*"'] \arrow[dr, "\mathsf{rf}"] & \\ & r \arrow[dl, dashed, "\mathsf{ofr}"] \\ w & \end{tikzcd}$$

Figure 48: Construction of an $\mathsf{ofr}$ edge.

We establish that $(R \cup \mathsf{ofr})^*$ must be acyclic. First define a partially constructed $\mathsf{ofr}$ as follows:

$$\mathsf{ofr}_n \triangleq \{(r : \mathrm{R}\ x\ v_1, w : \mathrm{W}\ x\ v_2) \mid \exists e.\ e \xrightarrow{\text{RF}} r \wedge e \xrightarrow{R \setminus \{(e,r)\} \cup \mathsf{ofr}_{n-1}}{}^{*}\ w\}$$
$$\mathsf{ofr}_0 \triangleq \{(r : \mathrm{R}\ x\ v_1, w : \mathrm{W}\ x\ v_2) \mid \exists e.\ e \xrightarrow{\text{RF}} r \wedge e \xrightarrow{R \setminus \{(e,r)\}}{}^{*}\ w\}$$

At $\mathsf{ofr}_0$, given $R^*$ is promised by MRD's acyclicity check to be acyclic, there cannot be an edge from $w$ to the witness of $e$ or from $r$ to $e$. This means the only way to cause a cycle between these three events is to create an edge from $w$ to $r$, which is the inverse of the edge we actually create.

The acyclicity of $(R \cup \mathsf{ofr}_n)^*$ follows from the same reasoning using the presumed acyclicity of $(R \cup \mathsf{ofr}_{n-1})^*$.

We now show by induction over $\xrightarrow{R \cup \mathsf{ofr} \cup R'}$ that we can perform operational steps in this order and obtain a progressive sequence of restrictions of an MRD execution.

**Base case** We take any arbitrary $\xrightarrow{R'}$ which makes $\xrightarrow{R \cup \mathsf{ofr} \cup R'}{}^{*}$ total and retains acyclicity, and take the set of $\mathsf{mo}$-minimal writes to be $E$. Each of these initialising writes will set a unique global variable to initial value as defined by Init. Let $e_1..e_m$ be these writes in the arbitrary order given by $R'$, as a set of writes to disjoint locations will be unordered by $R \cup \mathsf{ofr}$: The DP and $\mathsf{ofr}$ edges relate reads to writes, $\leq$ relates same-location accesses, and $\mathsf{rf}$ relates writes to reads.

Our state $\hat{\sigma}_0 = (W, \emptyset, \emptyset, \emptyset)$ where $W$ is the set of initialising writes. We let $\Lambda(\{e_1..e_m\}) = W$, as both of these sets contain one event or action respectively W $x$ 0 for each global variable $x$, and have $\Lambda(\emptyset) = \emptyset$ for the respective $\mathsf{rf}$ relations. Therefore $X_m \sim \hat{\sigma}_0$.

It remains to show that $X_{n-1} \sim \hat{\sigma}_{n-1} \Rightarrow X_n \sim \hat{\sigma}_n$ for $n \leq k$. There are 2 cases for the event label for $e_n$: a read event, or a write event.

**Inductive read case** In the case where $e_n$ is a read, we must use the semantic rule for appending a new read to state $\hat{\sigma}_{n-1}$, reprinted here:

$$\textsc{Tracked Read}\;\frac{\begin{array}{c} b = (g, a, t) \qquad g \notin \mathit{tags}(D) \qquad a = \text{i:R}\ x\ n \\ \sigma = (D, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}) \qquad w \in OW_\sigma(t) \qquad \mathit{loc}(w) = x \qquad \mathit{val}(w) = n \end{array}}{\hat{\sigma} \overset{a}{\rightsquigarrow}_t \hat{\sigma}[\mathsf{rf} \mapsto \mathsf{rf} \cup \{(w, a)\}, \mathit{reads} \mapsto \mathit{reads} \cup \{a\}]}$$

Maintaining the similarity requires an event $w$ to exist such that $(\Lambda(w), \Lambda(e_n)$ corresponds to the new event pair in $\mathsf{rf}_D$. This means that $\Lambda(w)$ must be in $OW_{\sigma_{n-1}}(\mathit{tid}(\Lambda(e_n)))$, and that $\mathit{loc}(w) = \mathit{loc}(e_n) \wedge \mathit{val}(w) = \mathit{val}(e_n)$. To maintain $\mathrm{RF} \sim \mathsf{rf}$, we must also have $(w, e_n) \in \mathrm{RF}$.

We now construct a witness for $w$. Given $X$ is complete, there must be some write $w'$ such that $w' \xrightarrow{\mathrm{RF}} e_n$. It must also be true that $w' \xrightarrow{R \cup \mathsf{ofr} \cup R'}{}^{*}\ e_n$, due to the inclusion of $\mathsf{rf}$ in $\xrightarrow{R}$. Therefore, $w'$ will be in $e_1..e_{n-1}$.

It remains to show that $w'$ will be in the observable writes set. We know that $\Lambda(w')$ will be in $OW_{\sigma_{n-1}}(\mathit{tid}(\Lambda(e_n)))$ if it is maximal in $\mathsf{mo}$, as this is strictly stronger than the original condition for inclusion into the set (that is, that $\Lambda(w')$ is maximal in $\mathsf{mo}$ restricted to the encountered events in the same thread).

**$\Lambda(w')$ may be maximal** Suppose $\Lambda(w')$ is not maximal and some event $w''$ exists in $X_{n-1}$ such that $(\Lambda(w'), \Lambda(w'')) \in \mathsf{mo}$. By the construction of $\mathsf{mo}$, it must also be true that $w' \xrightarrow{R \cup \mathsf{ofr} \cup R'}{}^{*}\ w''$.

If there is at least one $\xrightarrow{R'}$ edge connecting them, then by lemma 7 there exists another choice of $\xrightarrow{R'}$ such that $w''$ is added to $\hat{\sigma}_n$ before $w'$, which reverses their respective $\mathsf{mo}$ edge.

If there are no $\xrightarrow{R'}$ edges between them, then $w' \xrightarrow{R \cup \mathsf{ofr}} w''$. If $w' \xrightarrow{R \cup \mathsf{ofr}}{}^{*}\ w''$ and $w' \xrightarrow{\mathrm{RF}} e_n$, then we must have a from-reads edge $(e_n, w'') \in \mathsf{ofr}$. This is a contradiction, as it means that $w''$ cannot be in $e_1..e_n$ and therefore $\Lambda(w'')$ cannot be in $D$. By this and the above, whenever $w'$ is not maximal in $\mathsf{mo}$ we must be able to find a new $R'$ such that it becomes maximal. This gives us that $\Lambda(w')$ is in $OW_{\sigma_{n-1}}(\mathit{tid}(\Lambda(e_n)))$, so we use $w = w'$.

**Inductive write case** If $e_n$ is a non-initialising write, meaning there is some event $w \in X$ which is a write to the same variable as $e_n$, we must show that $w$ is in $OW_{\sigma_{n-1}}(\mathit{tid}(\Lambda(e_n)))$. This follows by the same reasoning as in the read case. □

**Lemma 7.** *If* $e_i \xrightarrow{R \cup R'}^* e_j \wedge e_i \not\xrightarrow{R}^* e_j$ *in the chain* $e_1..e_n$ *then there exists a valid choice of relation* $\xrightarrow{R''}$ *such that* $e_j \xrightarrow{R \cup R''}^* e_i$*, the union* $\xrightarrow{R \cup R''}$ *is acyclic, and there exists an event* $e'_n$ *in* $e_1..e_n$ *such that*

1. $e_1..e_n$ *under* $\xrightarrow{R \cup R'}$ *is equal to* $e_1..e'_n$ *under* $\xrightarrow{R \cup R''}$
2. $\sigma'_n$*, the result of running the operational semantics under* $\xrightarrow{R \cup R''}$*, is equal to* $\sigma_n$ *excepting relation edges between* $e_i$ *and* $e_j$

*Proof.* Begin by enforcing that $e_j \xrightarrow{R''} e_i$. A cycle in $\xrightarrow{R \cup R''}$ would have the shape $e_i \xrightarrow{R \cup R''} e_j$. As we can trivially prevent $e_i \xrightarrow{R''}^* e_j$ by construction, this relation edge must contain one or more event pairs $e$ and $e'$ such that $e \xrightarrow{R}^* e'$. This event pair can be moved $\xrightarrow{R''}$-before $e_j$ unless $e_j \xrightarrow{R}^* e$, and it can be moved $\xrightarrow{R''}$-after $e_i$ unless $e' \xrightarrow{R}^* e_i$. If both of these are true, then $e_i \xrightarrow{R}^* e_j$, which we know is untrue.

To find $e'_n$, choose any element which is not equal to $e_j$ from the set of events which are maximal in $\xrightarrow{R}$ and make it maximal in $\xrightarrow{R''}$. There must be some element in this set which is not $e_j$, as otherwise we would have $\forall e.e \xrightarrow{R} e_j$ and it would again be true that $e_i \xrightarrow{R}^* e_j$.

What remains is to create two arbitrary orderings $R''_1$ and $R''_2$ such that $e_1 \xrightarrow{R \cup R''_1} e_j$ and $e_j \xrightarrow{R \cup R''_2} e_n$ and both $\xrightarrow{R \cup R''_1}$ and $\xrightarrow{R \cup R''_2}$ are acyclic. Partition all remaining events into those which are and are not $\xrightarrow{R}$-before $e_j$, then construct $\xrightarrow{R''_1}$ over the first and $\xrightarrow{R''_2}$ over the second. These are individually trivial by the acyclicity of $\xrightarrow{R}$, and their union must be acyclic as their event sets are disjoint.

It is trivial to establish that $e_1..e_n = e_1..e_{n'}$ by the fact that all relation edges in $X$ are preserved in $\xrightarrow{R}$. To show that $\sigma_n$ is equivalent to $\sigma_{n'}$ excepting $(e_i, e_j)$ edges, it suffices to show that if $(e, e') \in \mathsf{mo}$ then $(e, e') \in \mathsf{mo}'$, as rf edges are constrained by $\xrightarrow{R}$. Events are only relocated around points $e_i$ and $e_j$, so any such edges would have the shape $(e, e_j)$ or $(e_i, e)$ in $\sigma_n$ and become $(e_j, e)$ or $(e, e_i)$ respectively. Focusing on the first case, any such reordering is only *required* if $e_j \xrightarrow{R}^* e$, as this forces $e$ to be placed $\xrightarrow{R \cup R''}^*$-after $e_j$ in our construction of $\xrightarrow{R''}$. This makes the initial mo edge impossible to observe. The same reasoning holds for the $(e_i, e)$ edge.

□

## 5.3 Program Logic

We can now return to our earlier formulation of indexed Hoare triples.

The notation $\{I\}_F\ \overline{Q}\ \{I'\}$ uses the set of predicates $I$ indexed by future set $F$ in conjunction with program text $Q$, represented as the unordered set of statements $\overline{Q}$. To establish a statement-by-statement derivation rule, we use the FUTURE-STEP execution rule:

$$\textsc{Future-Step}\ \frac{\overline{Q}(t) = \overline{C} \uplus \{i\colon\ s\} \qquad a \rhd F \neq \emptyset \qquad (i : s, \gamma) \xrightarrow{a}_t (\textsf{\textbf{skip}}, \gamma') \qquad \sigma \overset{(g,a,t)}{\rightsquigarrow} \sigma'}{(\overline{Q}, (\sigma, \gamma), F) \longrightarrow (\overline{Q}[t := \overline{C}], (\sigma', \gamma'), a \rhd F)}$$

This rule executes statement $s$ at line $i$, removing it from the set $\overline{Q}$, and updating the local state $\gamma$ and the tagged action graph $\sigma$ to $\gamma'$ and $\sigma'$ respectively. When invoked with initial future set $F$, it ensures that the action $a$ corresponding to statement $s$ is available in $F$, and returns $a \rhd F$ as the final part of the subsequent tuple.

To surround this with a Hoare triple, our precondition must be some invariant $I$ indexed into by the future set $F$, and the postcondition a (potentially different) invariant $I'$ indexed into by the future set $a \rhd F$. If the precondition $I(F)$ holds over the state pair $(\sigma, \gamma)$, then the postcondition $I'(a \rhd F)$ must hold over the resulting state pair $(\sigma', \gamma')$.

For convenience, we also wrap the $(\sigma, \gamma)$ pair in the *configuration* notation $\mathbb{C} = (\sigma, \gamma)$, as we no longer need to examine the individual components.

$$\frac{\forall \sigma, \gamma.\ (I(F)(\mathbb{C}) \wedge (\{i : s\}, \mathbb{C}, F) \longrightarrow (\emptyset, \mathbb{C}', a \rhd F)) \Rightarrow I'(a \rhd F)(\mathbb{C}')}{\{I\}_F\ \{i : s\}\ \{I'\}}$$

Since we would like a single invariant $I$ to hold for the entire program, we are chiefly interested in showing that $\{I\}_F\ \{i : s\}\ \{I\}$.

We now need to expand our logic to correctly combine pre- and postconditions for various statements into a correctness proof for a whole program. This needs to be slightly tighter than a traditional sequencing rule, as the operational rules are nondeterministic. A single execution step of one program might result in one of two or more possible sub-programs, all of which must be correct.

If we restrict ourselves to a single invariant, we can use the transitional step operator to construct a sequence-like rule by quantifying over all possible transitions which might result in termination. We know that an execution of a whole program can be described by the transitive closure of the relation given by the FUTURE-STEP rule. For program $P$ with initial future set $F$ and state $\mathbb{C}_0$, we know that state $\mathbb{C}$ is an acceptable outcome if we can show $(\overline{P}, \mathbb{C}_0, F) \longrightarrow^* (\emptyset, \mathbb{C}, \{\emptyset\})$. We also know that the only way to show this is to break $\overline{P}$ down into individual statements and repeatedly use the FUTURE-STEP rule to remove statements from the current working set, and that each invocation of this rule will update the future set accordingly.

Our rule for composing triples requires that if program $P$ can take a single transition step to program $Q$, and in doing so move from future set $F$ to future set $F'$, then program $Q$ must preserve the invariant under that future set and the transition must likewise preserve $I(F')$ given $I(F)$.

$$\frac{\begin{array}{l}\forall \mathbb{C}, \mathbb{C}_Q, \mathbb{C}^{\emptyset}, F_Q, Q. \\ ((\overline{P}, \mathbb{C}, F) \longrightarrow (\overline{Q}, \mathbb{C}_Q, F_Q) \longrightarrow^* (\emptyset, \mathbb{C}^{\emptyset}, \{\emptyset\})) \implies (I(F)(\mathbb{C}) \to I(F_Q)(\mathbb{C}_Q)) \wedge \{I\}_{F_Q}\overline{Q}\{I\}\end{array}}{\{I\}_F\ \overline{P}\{I\}}$$

By introducing as an axiom that the empty program preserves all invariants, we can use this as our *only* proof rule, doing away with the single-step rule entirely. If $P$ is a single statement, then $\overline{Q} = \emptyset$, and the rule simplifies to the single-step case above.

By partitioning our future sets and predicate sets into individual threads, we can use the Owicki-Gries method to create a modular correctness proof for concurrent programs. Each statement in each thread will have a family of triples associated with it: if statement $s$ in thread 1 is associated with action $a$, then for all possible $F$ such that $a$ is available, we will have a Hoare triple over $s$:

$$\forall F.\ a \triangleright F \neq \emptyset \Rightarrow \{I_1(F)\}s\{I_1(a \triangleright F)\}$$

The *global correctness constraint* therefore requires that all such triples in thread 1 are interference-free with all such triples in other threads, and we describe this in terms of the per-thread invariants.

**Definition 18** (Invariant Interference Freedom). *Invariants $I_{t_1}$ and $I_{t_2}$ are interference free if, for all $F_1$ such that $I_{t_1}(F_1)$ is defined and likewise $F_2$ for $I_{t_2}$, whenever we have $\{I\}_{F_1}\ \overline{P}\ \{I\}$ and $\{I\}_{F_2}\ \overline{Q}\ \{I\}$ then we also have $\{I_{t_1}(F_1) \wedge I_{t_2}\}_{F_2}\ \overline{P}\ \{I_{t_1}(F_1)\}$.*

We can now establish the correctness of a program through a series of smaller proof obligations.

**Lemma 8** (Owicki-Gries). *For each thread $t$, let $I_t$ be a future predicate corresponding to $t$. If $Init \Rightarrow X$, $X \Rightarrow \forall t.\ I_t(F_{|t})$ for initial future $F$ divided into thread futures $F_t$, and $\forall t.\ I_t(\emptyset) \Rightarrow Y$, then $\{X\}P\{Y\}$ holds provided both of the following hold.*

1. *For all threads $t$, $\{I_t\}_{F_{|t}}\overline{P_{|t}}\{I_t\}$* *(local correctness)*
2. *For all threads $t_1$, $t_2$ such that $t_1 \neq t_2$, $I_{t_1}$ and $I_{t_2}$ are interference-free.* *(global correctness)*

```
Init: x = 0 ∧ y = 0
                 ${[x = 0]_1 \wedge [x = 0]_2 \wedge [y = 0]_1 \wedge [y = 0]_2 \wedge r1 = r2 = 0}$
Thread 1                                 Thread 2
${[y = 0]_2 \wedge r2 = 0$                   ||  ${[x = 0]_1 \wedge r1 = 0}_G$
  $\wedge (\forall i.\ i \notin {0,1} \Rightarrow \neg[x \approx i]_1)}_F$   ||    ${r1 = 0 \vee (r1 = 1 \wedge (\forall i.\ i \notin {0,2} \Rightarrow \neg[y \approx i]_2))}_{G_4}$
1: r1 := [x]                             ||     3: r2 := [y]
${[y = 0]_2 \wedge r2 = 0 \wedge r1 \in {0,1}}_{F_1}$   ||    ${[x = 0]_1 \wedge r1 = 0}_{G_3}$
2: [y] := r1 + 1                         ||     4: [x] := 1
${r1 \neq 1 \vee r2 \neq 1}_{{\emptyset}}$                 ||  ${r1 \neq 1 \vee r2 \neq 1}_{{\emptyset}}$
                            ${r1 \neq 1 \vee r2 \neq 1}_{{\emptyset}}$
```

Figure 49: Proof outline for load buffering with semantic dependencies

## 5.3.1 Worked Examples

Because our predicates are written over tagged action graphs and not functions from locations to values, it is not immediately meaningful to write a predicate such as $[x = 1]$. Instead we take this statement to mean that any read of $x$ interpreted at this point in the program would observe the value 1, because a write of 1 to $x$ is mo-latest in the configuration.

$$[x = v]_t(\sigma, \gamma) \triangleq \exists(w : \mathrm{W}\ x\ v) \in \sigma.D\ .\ \forall(w' : \mathrm{W}\ x\ v').\ (w', w) \in \mathsf{mo} \vee v' \neq v$$

**LB+sdep**

The interleaved program text and future-indexed assertions for this example are given in Fig. 49.

To reduce the domain of our future predicate, we collapse the event-based futures used by the operational semantics into sets of label-based futures. An event future $F_E$ can be converted into a label future $F_L$ by applying the labelling function to all events in $F_E$. This makes the futures $\{3\}$ and $\{5\}$ equivalent, as both are instances of $\{2{:}\mathrm{W}\ y\ 2\}$, thus $I(\{3\}) = I(\{5\})$. This isn't always a valid step, as some events which share labels may not be related by $\preceq$ in the same way. Thus, the technique can only be used if the label-based representation describes exactly the futures generated by MRD, which is the case for our example.

We describe our initial set of label futures as $\{\{1^u \prec 2, c^v, 4\} \mid u \in \{0,1\} \wedge v \in \{0,1,2\}\}$. We verify that this is a valid step: all futures generated by MRD are described by these labels, and they do not describe any potential futures not generated by MRD. We partition this into $F = \{\{1^u \prec 2\} \mid u \in \{0,1\}\}$ and $G = \{\{3^v, 4\} \mid v \in \{0,1,2\}\}$ representing the future sets of threads 1 and 2, respectively. We let $F_1 = \{\{2\}\}$ be the future set after executing line 1, $G_3 = \{\{4\}\}$ be the future set after executing line 3 (reading some value for $y$), and $G_4 = \{\{3^v\} \mid v \in \{0,1,2\}\}$ be the future set after executing line 4.

Our future predicate $I$ must output a configuration predicate for every sub-future of the initial set. Our partitioning fully describes these sub-futures, so we now attach our assertions to these futures.

To simplify the visualisation, we interleave the future predicate components with the program to provide Hoare-style pre/post-assertions, and place the assertion above a line of code if that line of code is contained in the applied future. We use indentation to denote that an assertion applies to more than one future. In this example $G$ contains both lines 3 and 4, hence both lines are indented w.r.t. the first assertion in thread 2. We apply Lemma 8 to break down our proof burden into separate threads, and in the discussion below, we describe the local and global correctness checks.

For local correctness in thread 1, we must establish that $\{I\}_F\{1,2\}\{I\}_{\{\emptyset\}}$. According to future $F$, line 1 must be executed before line 2. This means we only need to verify that $\{I\}_F\{1\}\{I\}_{F_1}$ and $\{I\}_{F_1}\{2\}\{I\}_{\{\emptyset\}}$, which is identical to a standard Hoare logic proof and relatively uninteresting.

In thread 2, no order is imposed between lines 3 and 4. This means to establish local correctness we must check that:

- $\{I\}_G$ `3: r2 := y` $\{I\}_{G_3}$
- $\{I\}_G$ `4: x := 1` $\{I\}_{G_4}$
- $\{I\}_{G_3}$ `4: x := 1` $\{I\}_{\{\emptyset\}}$
- $\{I\}_{G_4}$ `3: r2 := y` $\{I\}_{\{\emptyset\}}$

The first three are trivial: line 3 modifies neither $x$ nor $r1$ and line 4 does not modify $r1$. For the final step, the first disjunct of $\{I\}_{G_4}$ is the same as the first disjunct of $\{I\}_{\{\emptyset\}}$, and the second disjunct ensures that we cannot observe $r2 = 1$ after executing line 3.

For global correctness, we must check that every assertion in thread 1 continues to hold after every line of thread 2, and vice versa. These checks are also straightforward, so omit a detailed discussion. The only noteworthy aspect is that for line 3 (and similarly, line 4), our precondition is the conjunction $\{I\}_{G_4} \wedge \{I\}_G$, as both $G$ and $G_3$ are subfutures corresponding to line 3.

**Partial Conditional**

```
Init: x = 0 ∧ y = 0
Thread 1
$\{[x = 0]_2 \wedge [y = 0]_2\}_F$
    $\{[x = 0]_2 \wedge [y \in \{0, 1\}]_2\}_{F_2}$
    1: x := 1
    $\{([x \approx 0]_2 \vee [x \approx 1]_2)$
    $\wedge [y = 0]_2\}_{F_1}$
    2: y := 1
$\{([x \approx 0]_2 \vee [x \approx 1]_2)$
$\wedge ([y \approx 0]_2 \vee [y \approx 1]_2)\}_{\{\emptyset\}}$

Thread 2
$\{I\}_G$
    $\{r1 = 0 \wedge r2 = n \wedge I\}_{G_{4^n}}$
    3:  r1 := x ;
    $\{r1 = m \wedge r2 = 0 \wedge I\}_{G_{3^m}}$
    4:  r2 := y ;
    $\{r1 = m \wedge r2 = n \wedge I\}_{G_{3^m,4^n}}$
        if r1 = 1 & r2 = 1 {
        $\{r1 = 1 \wedge r2 = 1 \wedge I\}_{G_{3^1,4^1}}$
            $\{r1 = 1 \wedge r2 = 1 \wedge [w = 0]_2 \wedge [z = 1]_2\}_{G_{3^1,4^1,6}}$
            $\{r1 = 0 \wedge r2 = 1 \wedge [w = 0]_2 \wedge [z = 1]_2\}_{G_{4^1,6}}$
            5:  w := 1 ;
            $\{r1 = 1 \wedge r2 = 1 \wedge [w = 1]_2 \wedge [z = 0]_2\}_{G_{3^1,4^1,5}}$
            $\{r1 = 1 \wedge r2 = 0 \wedge [w = 1]_2 \wedge [z = 0]_2\}_{G_{3^1,5}}$
            6: z := 1 ; }
        if r1 = 0 & r2 = 1 {
        $\{r1 = 0 \wedge r2 = 1 \wedge I\}_{G_{3^0,4^1}}$
            7: z := 1 }
        if r1 = 1 & r2 = 0
        $\{r1 = 1 \wedge r2 = 0 \wedge I\}_{G_{3^1,4^0}}$
            8: w := 1 ;
$\{[w = r1]_2 \wedge [z = r2]_2\}_{\{\emptyset\}}$

$\{[w = r1]_2 \wedge [z = r2]_2\}_{\{\emptyset\}}$
```

Figure 50: Partial conditional: $F = \{\{1, 2\}\}$ $G = \{\{3^1 \prec 5, 4^1 \prec 6\}, \{3^1 \prec 8, 4^0\}, \{3^0, 4^0\}, \{3^0, 4^1, 7\}\}$

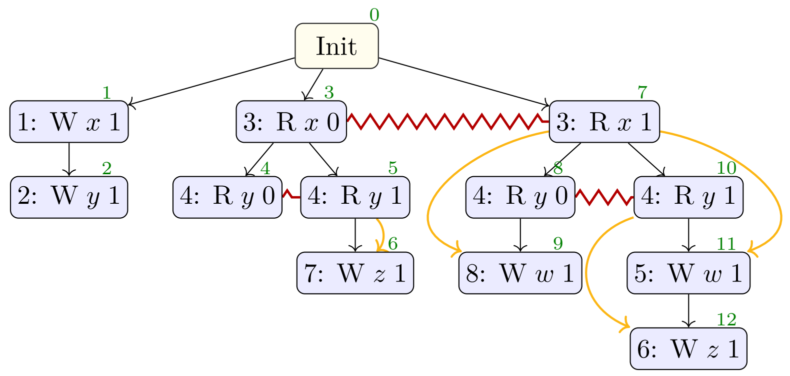


Figure 51: Event structure generated by the program in Fig. 50, including the minimal dependency edges for the writes in thread 2 (represented by the orange arrows).

Our next example is the partial conditional example, whose proof outline is given in Fig. 50.

The program contains no dependencies in Thread 1 and non-trivial dependencies in thread 2. In particular, in Thread 2 the dependencies are condition on the values read for $x$ and $y$.

- If $r1 = r2 = 1$, then we have dependencies between lines 3 and 8, and between lines 4 and 6, but lines 4/8 and 3/6 are mutually independent.
- If $r1 = 1$ and $r2 = 0$, there is a dependency between lines 3 and 8.
- If $r1 = 0$ and $r2 = 1$, there is a dependency between lines 4 and 7.

This prevents us from using our earlier tactic of using a line number to refer to any event generated by that line, since we now need both the line number and the value observed to know what our dependencies are. The futures must distinguish between events where line 3 observes the value 1, referred to as $3^1$, and events where it reads 2, referred to as $3^2$.

As before, we use the name of the future set in each assertion to track the events that have been executed. For instance

$$\{r1 = 1 \wedge r2 = 1 \wedge I\}_{G_{3^1,4^1}}$$

corresponds to the assertion that holds after executing lines 3 and 4, where both reads return the value 1.

This asymmetric dependency is made clearer by looking at the event structure generated for the program, shown in Fig. 51. The dependency edges shown in the futures are represented as yellow arrows. They show that we cannot write 1 to $w$ without observing 1 at $x$, and that we cannot write 1 to $z$ without observing 1 at $y$.

## 5.4 Drawbacks and Room for Improvement

The program logic presented here is a clean foundation for verifying weak memory programs, but a handful of inelegant points remain. One is that the interference freedom requirement means that many programs require the addition of auxiliary variables to be verified, and these auxiliary variables often restate information used by the MRD model. For instance, two sub-programs may be interference free because a combination of inter-thread reads from and intra-thread dependency orderings cause them to be linearly ordered, as we saw earlier, but without explicit representation of a reads from edge within the program logic we need to show interference freedom nonetheless.

The second is that, due to the underlying model, we face a potentially exponential (in the number of reads, with the base being the size of the value range) set of futures for any given program. In many cases these futures behave identically, and we use some notational shorthand to treat them as a single future. However, determining which futures fall into equivalence classes is still a task that falls to the individual creating the proof. With mechanisable models like MRD, it would be ideal for these classes to naturally arise from the calculation of dependencies.

# Chapter 6

# Symbolic Event Structures

There is a particular case that no language-level weak memory model has yet been able to provide a semantics for, and this is the case of memory address literals (i.e. pointers) with dynamic memory allocation. The reason for this is rooted in how weak memory models are forced to model the contents of memory, especially when interpreting reads. For interpretation to proceed, even if we cannot assume which writes have happened yet, we need to fetch something and place it in a register. Different models handle this differently, with the Java model trying to pick a value from a strongly sequenced before write and swapping to another write on commit steps, Promising picking a value from either a write or a promise, and MRD forking its interpretation and continuing with each possible value.

The value of a dynamically allocated object can always be assumed or forced to be initialised to something in a finite range, but the information about the initial state of a newly allocated pointer is totally unavailable to a semantic model. To simply define it as either an unallocated or allocated location is insufficient – what if it was allocated, but the pointer is being used with an illegally large offset? Is our referenced object still live, or has it been freed? How should we answer questions about pointer comparisons, or what optimisations a compiler may to do them, if we have no knowledge of where, precisely, our pointers are pointing?

The regular MRD model is ultimately incapable of answering these questions. We could, theoretically, coproduct over the entirety of the x86 address space whenever we dereference a pointer, but this wouldn't give us information about legal bounds. It would also be unusably slow, and illegible to a human user. Instead, we turn to symbolic evaluation. We were able, in Section 4.5, to replace the initial register environment with a mapping between registers and uninterpreted variables. The natural continuation of this is to replace all mentions of concrete values with uninterpreted symbolic variables, and generalise the semantics as needed to ensure that a symbolic structure can be expanded into a concrete structure without any change in permitted behaviours.

Where previously we needed one new branch per value per read in order to proceed in the computation, we instead use uninterpreted symbols for all values in the program and branch on control flow. Whenever a control flow path limits the potential values a symbolic variable could have, we record this in the structure.

In the proceeding, we introduce symbolic MRD without fences or release/acquire events. This is entirely for ease of presentation. The most substantial changes to the model ignore all orderings which are not semantic dependency, so the omitted features can be re-implemented with minimal difficulty.

**Preliminary: Conditional Labelling Functions** For ease of writing, we here briefly define two functions which take events and return predicates. The first is the *strong conditional labelling function*, which requires that its two input events have matching locations and values:

$$e_1 \Longleftarrow e_2 \triangleq \mathit{loc}(e_1) = \mathit{loc}(e_2) \land \mathit{val}(e_1) = \mathit{val}(e_2)$$

The second is the *weak conditional labelling function*, which only requires them to share a location:

$$e_1 \longleftarrow e_2 \triangleq \mathit{loc}(e_1) = \mathit{loc}(e_2)$$

## 6.1 Structures, Events, and Conditions

A *symbolic event structure* is a tuple $(E, <, \sqsubseteq, \#, \curlyvee)$ where

- $E$ is a set of events, represented as id-label pairs $(e, \mathrm{R}\ x\ v)$. Events may be written as $((e : \mathrm{R}\ x\ v))$ or referred to simply by their id $e$. Some executions may disagree on the label of $e$ due to various choices of forwarding or reads-from edges, but two events are seen as equivalent if they share a unique identifier.
- *Program order* $<$ orders all events according to their syntactic order in the program.
- *Branch order* $\sqsubseteq$ orders events before or after control flow events.
- *Conflict* $\#$ relates events which cannot occur in the same execution, but in symbolic MRD reflects *control flow* conflict and not data flow conflict.

- *Value restriction* $\curlyvee$ is a function from events to unquantified predicates over symbolic values. This component tracks the control flow requirements for us to observe a particular event. If $\curlyvee(e) = \{\alpha = 1\}$, then $e$ is below a guard which checks that the value of $\alpha$ is 1.

Labels are similar to labels in concrete MRD, with the addition of a new branch label. As before, reads and writes represent atomic trips to memory, while the new branch event represents the evaluation of a conditional. Our labels are given as follows:

- Read labels R $x$ $v$, where $\mathcal{R}$ gives the set of all reads
- Write labels W $x$ $v$, where $\mathcal{W}$ gives the set of all writes
- Branch labels $[\varphi]$, where $\mathcal{B}$ gives the set of all branches

An event *introduces* symbolic variable $\alpha$ if it is the unique read in which the variable $\alpha$ occurs.

### 6.1.1 Condition Notation

There are two kinds of predicate which appear in these structures: things we *know*, written using $\psi$, and things we need to *check* at runtime, written using $\varphi$. If an event occurs under a guard, then we need to check the values referenced in the guard unless we can lift it. If whole-program analysis has shown that all values written within the program are within a certain range, then we know that any symbol will be within that range and never need to check during runtime that this property still holds.

Control dependencies in symbolic MRD are checks, which are gathered during the construction of the event structure and attached to events. In the dependency calculation, we can derive pieces of knowledge from analysis of the structure which can simplify checks. If we write the value 1 to $x$ and immediately read it into register $r_1$, we can introduce to an execution the knowledge that the read will observe a value of 1 – this is not necessarily true in every execution, but we are allowed an execution in which we assert that we already know what value the read will observe. If we subseqently branch on the contents of $r_1$, the execution in which we made the assumption is allowed to effectively skip the check and avoid creating any control dependencies, while executions where we do not know the value of $r_1$ must still perform it.

We must ensure that any constraints on symbolic values we generate are consistent with the knowledge used to generate the relations *and* will allow all required checks to pass, so both knowledge and checks are consistency-checked when we construct executions.

## 6.2 Building SES

Symbolic event structures are constructed by recursing over an input program in the same way as the original MRD semantics, but now converting both memory accesses and control flow branches into events.

The semantic interpretation function takes an input line of code and returns the new event, while the append function adds the new event to the structure. The notation $[\![P]\!]_{n\ \rho\ \kappa\ \varphi}$ denotes the semantic interpretation function, which is largely identical to the concrete MRD interpretation function given in Figure 32 but introduces the novel parameter $\varphi$. The condition $\varphi$ is a predicate used to build value the value restriction $\curlyvee$, tracking the branches which have been added to the structure so far.

The notation $e \bullet (E, <, \sqsubseteq, \#, \curlyvee)$ denotes the append function, which is defined per event type in the proceeding.

### 6.2.1 Read Appends

Interpreting a load from a global variable into a local register is relatively simple. A new unique value identifier $\alpha$ is created to represent the unknown value loaded, and the environment is updated to show that the referenced register now contains this value and its contents depend on the single load event. We do not need to generate the summation of all possible reads here, as we did in concrete MRD, thanks to the use of a symbolic value.

$$[\![\mathtt{r}_1 := x]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq (e : \mathrm{R}\ x\ \alpha)[\varphi] \bullet \kappa(\rho[\mathtt{r}_1 \mapsto (\alpha)])$$

Registers can also be updated via expressions over registers and constants. In this case no read event is generated, as there was no access to a global variable, but the environment is updated. We leave the language of expressions uninterpreted, and use $[\![e]\!]_\rho$ to denote the substitution of registers in $e$ with their values in $\rho$.

$$[\![\mathtt{r}_1 := e]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq \kappa(\rho[\mathtt{r}_1 \mapsto [\![e]\!]_\rho])$$

Appending a read to the structure updates $\sqsubseteq$ to place the read $\sqsubseteq$-before any pre-existing branches and any events $\sqsubseteq$-after those branches.

$$(e : \mathrm{R}\ x\ \alpha)[\varphi] \bullet (E, <, \sqsubseteq, \#, \curlyvee) \triangleq (\{e\} \cup E, < \cup <', (\sqsubseteq \cup \sqsubseteq')^*, \#, \curlyvee[e \mapsto \varphi])$$

Where:

$$\begin{aligned} <' &= \{e\} \times E \\ \sqsubseteq' &= \{(e, b) \mid (b : [\varphi_b]) \in E\} \end{aligned}$$

### 6.2.2 Write Appends

The value in a register can be stored in a global variable, appending a write event to the structure. As the contents of registers can be expressions, the label on a write may also be an expression.

$$[\![x := \mathtt{r}_i]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq (e : \mathrm{W}\ x\ \rho(\mathtt{r}_i)) \bullet \kappa(\rho)$$

Appending a write to a structure updates branch and program order in the same manner as a read:

$$(e : \mathrm{W}\ x\ e)[\varphi] \bullet (E, <, \sqsubseteq, \#, \curlyvee) \triangleq (\{e\} \cup E, < \cup <', (\sqsubseteq \cup \sqsubseteq')^*, \#, \curlyvee[e \mapsto \varphi])$$

Where:

$$\begin{aligned} <' &= \{e\} \times E \\ \sqsubseteq' &= \{(e, b) \mid (b : [\varphi_b]) \in E\} \end{aligned}$$

### 6.2.3 Branch Appends

Interpreting an if-then statement causes the interpretation function to also branch, as it does in concrete MRD when encountering a read. Instead of assuming a particular assignment for each read, we assume one context in which the branch condition held, and one in which it did not. The guard condition is reflected in the $\varphi$ parameter, and propagated to all subsequent events.

$$[\![\mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2]\!]_{n\ \rho\ \kappa\ \varphi} = (e : [\varphi_b]) \bullet ([\![P_1]\!]_{n\ \rho\ \kappa\ (\varphi \wedge \varphi_b)} + [\![P_2]\!]_{n\ \rho\ \kappa\ (\varphi \wedge \varphi_b^-)})$$

Where $\varphi_b$ is the result of substituting all registers in expression $b$ with their values in $\rho$.

$$\varphi_b = b_{[\rho(\mathtt{r}_i) \backslash r_i]}$$

The scope of the condition $\varphi$ here ensures that `if b then A else B; C` will push condition $b$ through the interpretation of $A$ but *not* through the interpretation of $C$, as we can no longer assume that $b$ holds when we are executing $C$.

We also add an extra rule for returning an empty structure whenever the scoped condition $\varphi$ becomes unsatisfiable:

$$[\![P]\!]_{n\ \rho\ \kappa\ \bot} \triangleq \emptyset$$

This ensures that all terminating while loops produce finite structures, and reduces the size of structures containing large numbers of redundant branches.

At the structure level, coproduct need only be defined pairwise, as all control flow events have only two branches. It takes the union of each structure's event sets and relations, and then places all events of one structure in conflict with the events of the other.

$$(E_1, <_1, \sqsubseteq_1, \#_1, \curlyvee_1) + (E_2, <_2, \sqsubseteq_2, \#_2, \curlyvee_2) \triangleq (E_1 \cup E_2, <_1 \cup <_2, \sqsubseteq_1 \cup \sqsubseteq_2, \#_1 \cup \#_2 \cup (E_1 \times E_2), \curlyvee_1 \cup \curlyvee_2)$$

Appending the branch event to the top of the resulting structure simply places it before all existing events in branch order. It does not update the $\curlyvee$ function, as we never use $\curlyvee$ of a branch event for any computation, nor does it update $<$, as we do not need the program ordering of branch events for any calculation.

$$(e : [\varphi_b])[\varphi] \bullet (E, <, \sqsubseteq, \#, \curlyvee) \triangleq (\{e\} \cup E, <, \sqsubseteq \cup \sqsubseteq', \#, \curlyvee)$$

Where:

$$\sqsubseteq' = \{(e, e') \mid e' \in E\}$$

### 6.2.4 While loops

Interpreting while loops uses the new function $[\varphi'][\![P_1]\!]_{n\ \rho\ \kappa\ \varphi}$, which applies $\varphi'$ as an *unscoped* condition, adding it to the condition of every event in the generated structure.

$$[\varphi'](E, <, \sqsubseteq, \#, \curlyvee) \triangleq (E, <, \sqsubseteq, \#, \lambda x.\ \curlyvee(x) \wedge \varphi')$$

After exiting the loop, we may assume the negation of the guard for the rest of the program, which we represent with the unscoped condition. It is otherwise a repeated application of the rule for if-then statements which decrements the step index, as it was for concrete MRD.

$$[\![\text{while } (b) P]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq (e : [\varphi_b]) \bullet ([\![P; \text{while } (b)\ P]\!]_{n-1\ \rho\ \kappa\ (\varphi \wedge \varphi_b)} + [\neg\varphi_b][\![skip]\!]_{n\ \rho\ \kappa\ \varphi})$$

## 6.2.5 Sequential Composition

The rule for sequential composition is slightly subtle in how it handles the condition $\varphi$. During evaluation of the code `if (b) {a; b} c`, we want to retain condition $\varphi_b$ while interpreting statements $a$ and $b$, but discard it for statement $c$. We use the sequential composition operator to combine interpretations of $a$ and $b$, but invoke a continuation to build the interpretation of $c$. Following this, the rule for sequential composition retains the condition $\varphi$, but invoking a continuation always discards it:

$$\begin{aligned} [\![P_1; P_2]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq [\![P_1]\!]_{n\ \rho\ (\lambda\rho.\ [\![P_2]\!]_{n\ \rho\ \kappa\ \varphi})\ \varphi} \\ [\![skip]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq \kappa(\rho) \end{aligned}$$

## 6.2.6 Parallel Composition

During interpretation, parallel composition interprets the two sub-programs independently into completed structures and then combines these structures with the $\times$ operator.

$$[\![P_1 \parallel P_2]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq [\![P_1]\!]_{n\ \rho\ \kappa\ \varphi} \times [\![P_1]\!]_{n\ \rho\ \kappa\ \varphi}$$

At the event structure level, the $\times$ operator combines two structures by taking the union of their event sets and relations without adding any new edges.

$$(E_1, <_1, \sqsubseteq_1, \#_1 \curlyvee_1) \times (E_2, <_2, \sqsubseteq_2, \#_2, \curlyvee_2) \triangleq (E_1 \cup E_2, <_1 \cup <_2, \sqsubseteq_1 \cup \sqsubseteq_2, \#_1 \cup \#_2, \curlyvee_1 \cup \curlyvee_2)$$

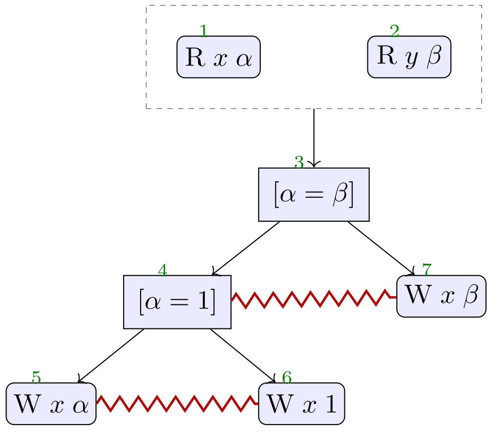
### 6.2.7 Full Semantic Interpretation Function and Examples

$$\begin{aligned}
[\![P_1; P_2]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq [\![P_1]\!]_{n\ \rho\ (\lambda\rho.\ [\![P_2]\!]_{n\ \rho\ \kappa\ \varphi})\ \varphi} \\
[\![\mathit{skip}]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq \kappa(\rho) \\
[\![P]\!]_{n\ \rho\ \kappa\ \bot} = [\![P]\!]_{0\ \rho\ \kappa\ \varphi} &\triangleq \emptyset \\
[\![\mathtt{r}_1 := x]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : \mathrm{R}\ x\ \alpha)[\varphi] \bullet \kappa(\rho[\mathtt{r}_1 \mapsto (\alpha)]) \\
[\![x := \mathtt{r}_1]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : \mathrm{W}\ x\ \rho(\mathtt{r}_1)) \bullet \kappa(\rho) \\
[\![\text{if } b \text{ then } P_1 \text{ else } P_2]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : [\varphi_b]) \bullet ([\![P_1]\!]_{n\ \rho\ \kappa\ (\varphi \wedge \varphi_b)} + [\![P_2]\!]_{n\ \rho\ \kappa\ (\varphi \wedge \varphi_b^-)}) \\
[\![\text{while } (b) P]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq (e : [\varphi_b]) \bullet ([\![P; \text{while } (b)\ P]\!]_{n-1\ \rho\ \kappa\ (\varphi \wedge \varphi_b)} + [\neg\varphi_b][\![\mathit{skip}]\!]_{n\ \rho\ \kappa\ \varphi}) \\
[\![P_1 \parallel P_2]\!]_{n\ \rho\ \kappa\ \varphi} &\triangleq [\![P_1]\!]_{n\ \rho\ \kappa\ \varphi} \times [\![P_1]\!]_{n\ \rho\ \kappa\ \varphi}
\end{aligned}$$

Figure 52: The semantic interpretation function for symbolic MRD without fences.

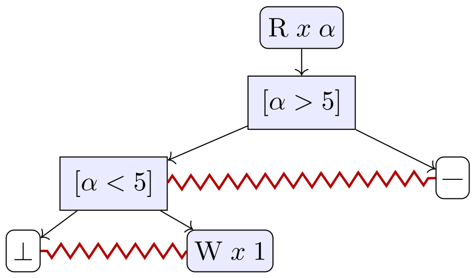
Here we briefly give the symbolic event structure for a small number of programs and explain their structure. The primary ordering represented visually is a union of branch order and preserved program order.

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



Conflict edges are drawn immediately below branch events, and maintain downward closure with respect to ordering edges. Below branch events, the left subtree indicates the true branch while the right subtree indicates the false branch. The two reads at the top of this structure do not necessarily occur in separate threads, but there are no ordering edges between them as they are unrelated reads from unrelated variables. The dashed box indicates that they share outgoing and incoming branch order edges.

```
r1 := x;
if (r1 > 5) {
    if (r1 < 5) {
        y := r1;
    } else {
        x := 1;
    }
}
```



The white box containing a single line denotes the absence of any event in one branch of a guard, for instance as the result of an `if` statement with no `else` block. The box containing a $\perp$ indicates an inaccessible branch, where the semantic interpretation halted due to an unsatisfiable constraint.

## 6.3 Executions

As in concrete MRD, executions are derived by analysis of the complete structure. An execution of a symbolic event structure is given by the tuple:

$$X = (E, \textsc{rf}, \textsc{dp}, \leq, \psi)$$

Where:

- $E$ is a set of events
- RF is the reads-from relation
- DP is the semantic dependency relation
- $\leq$ is a choice of preserved program order relation, which may vary between executions due to differing constraints on symbolic locations when these are introduced in Chapter 7 but may not contravene $<$.
- $\psi$ is the execution condition, a predicate over symbolic values which describes the restrictions applied to those values in this execution

The easiest way to visualise a symbolic value execution is as a recipe for creating a set of concrete value executions: Taking the non-branch events in $E$, exhaustively assign values within a finite range to symbolic variables, and then remove all choices of assignments which violate

$\psi$. The result, by Theorem 3, is a set of executions of the concrete MRD interpretation of the same program with the given value range.

The contents of $\psi$ are constrained by the choices of RF and DP, and by the control flow required for the events in $E$ to execute.

Splitting these constraints by origin, the RF edges each require that the write on the left is writing the value observed by the read on the right, and that there are no writes $\leq$-between them.

$$\forall (w : \mathrm{W}\ x\ \alpha), (r : \mathrm{R}\ x\ \beta).\ w \xrightarrow{\text{RF}}_X r \Rightarrow (\phi_X \Rightarrow \alpha = \beta) \land \nexists (w' : \mathrm{W}\ x\ \gamma).\ w \leq_X w' \leq_X r$$

Naturally, we also require that all branches inside an execution can return the expected value (i.e. true if we take the true path, false if we take the false path). Since these are all recorded per-event in the $\curlyvee$ function, we check the satisfiability of $\psi_X \land \curlyvee(e)$ for all events $e$ in the execution.

$$\forall e \in E_X.\ \psi_X \land \curlyvee(e) \text{ is satisfiable.}$$

To handle DP we use the freeze operation detailed in Section 6.4.4, which returns a triple $(\text{DP}_w, l, \varphi_w)$ for an input write $w$. The dependency edges are contained in $\text{DP}_w$, while the component $l$ refers to a label which may overwrite the label of $w$ if it has been simplified to a write of a constant. The condition $cond_w$ reflects any value assumptions or analysis needed for the calculation of the underlying justifying set, such as the value equivalence used by a forwarding operation or a value constraint which happens to allow additional coproduct operations.

We fold these requirements, in addition to the requirement that all writes are properly frozen, into the definition of coherence.

**Definition 19** (Complete Symbolic Execution)**.** *An execution $X$ is complete if and only if for all $(r : R\ x\ \alpha)$ in $X$ there exists some $(w : W\ x\ \beta)$ such that $w \xrightarrow{RF}_X r$.*

**Definition 20** (Coherent Symbolic Execution)**.** *An execution $X$ is coherent if and only if:*

1. *$(\leq \cup \text{DP}_X \cup \text{RF}_X)^*$ is acyclic.*
2. *Whenever $(w : W\ x\ \alpha) \xrightarrow{RF}_X (r : R\ x\ \beta)$:*
   - *(a) $(\psi_X \Rightarrow \alpha = \beta)$ and*
   - *(b) for all other events $e \in E$ , either $\psi_X \Rightarrow val(e) \neq \alpha$ or $\neg(w \leq_X e \leq_X r)$.*
3. *For all events $e_1$ and $e_2$, if $e_1 < e_2$ and $loc(e_1) = loc(e_2)$ then $e_1 \leq_X e_2$.*

4. *For all writes* $(w : \mathrm{W}\ x\ \alpha) \in E_X$*, there exists a dependency triple* $(\mathit{DP}_w, (\mathrm{W}\ x\ \alpha), \varphi_w) \in \mathit{freeze}(w)$ *such that* $\mathit{DP}_w \subseteq \mathit{DP}_X$ *and* $\psi_X \Rightarrow \varphi_w$*.*

5. *For all events* $e \in E_X$ *, the conjunction* $\psi_X \wedge \Upsilon(e)$ *is satisfiable.*

## 6.4 Dependency Calculations

In symbolic MRD, we split dependency into two components: data and control dependencies. Both data and control dependencies are given relative to ID-label pairs, as an event's data dependencies may vary based on its label and its control dependencies may vary based on its data dependencies. We also drop the representation of in-calculation control dependencies in terms of events and instead represent them as predicates, which provides a more explicit interface for the functional analysis used in simplifying branch guards. The freeze operation then converts these predicates into event sets to create dependency edges. In this section we briefly introduce our notion of dependencies and justifications, and how those differ from concrete MRD, then describe the operations that build them in detail.

Justifications are pairs of a justifying predicate and a justifying set. A predicate $P$ is a justifying predicate for $(w : \mathrm{W}\ x\ v)$ if an event with the label $\mathrm{W}\ x\ v$ is guaranteed to run whenever $P$ holds. An event $e$ is in the justifying set for $w$ if it must return before $w$ because it is either a data dependency for $w$ or a data dependency for a write we are treating as equivalent to $w$. A write $(w : \mathrm{W}\ x\ v)$ is independent if $(\top, \emptyset) \vdash (w : \mathrm{W}\ x\ v)$.

Justifications also have conditions associated with them, written $(P, D) \vdash^{\psi} (w : \mathrm{W}\ x\ v)$. Justifying predicates and justification conditions have subtly different meanings. Recall the know/check distinction from earlier: the predicate $P$ describes information that must be checked during execution before a write can be performed, while the condition $\psi$ describes information that the executing machine has assumed to hold. If two consecutive reads from the same location show the same value in the event structure, then the machine is free to assume that the second read never loaded anything from memory and instead copied over the first value, thus that the two registers loaded into must contain the same value. The machine itself never performs an equality check, it simply declares the values to be equal and optimises away the second read. On the other hand, a branch event creates a predicate but not a condition. If there is a true dependency between a branch and a write, then the machine cannot execute the write before knowing if the guard holds or not.

The pre-justification for an event $(w : \mathrm{W}\ x\ e)$ is $(\Upsilon(w), D) \vdash (w : \mathrm{W}\ x\ e)$, where $D = \{(r : \mathrm{R}\ x\ \alpha) \mid \alpha \in \mathit{vars}(e)\}$. This is a pair consisting of the value restriction at $w$ and the origin of all symbolic variables in $e$ (which may be the empty set if the write is of a constant). Both the predicate and set are then subject to *forwarding*.

### 6.4.1 Forwarding

Forwarding in Symbolic MRD is identical to forwarding in the symbolic environment structures of Section 4.5, but whenever we try to run the forwarding operation across symbolic variables, we must record the decisions we made about variable equality.

$$(r_1 : \mathrm{R}\ x\ \alpha) \xrightarrow{LF}^{\alpha=\beta} (r_2 : x\beta) \text{ if } \nexists e.\ r_1 \leq e \leq r_2$$

likewise, $w \xrightarrow{SF}^{\psi} r$ denotes that the read $r$ is forwarded to the write $w$:

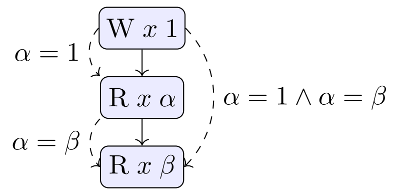
$$(w : \mathrm{W}\ x\ v) \xrightarrow{SF}^{\alpha=v} (r : \mathrm{R}\ x\ \alpha) \text{ if } \nexists e.\ w \leq e \leq r$$

$e_1 \xrightarrow[e]{f}^{\psi} e_2$.

**Definition 21** (The Symbolic Forwarding Relation)**.** *If* $r_1 \xrightarrow{LF}^{\psi} r_2$ *then* $r_1 \xrightarrow{F}^{\psi} r_2$
*If* $w \xrightarrow{SF}^{\psi} r$ *then* $w \xrightarrow{F}^{\psi} r$
*If* $e_1 \xrightarrow{F}^{\psi_1} e_2$ *and* $e_2 \xrightarrow{F}^{\psi_2} e_3$ *then* $e_1 \xrightarrow{F}^{\psi_1 \wedge \psi_2} e_3$

If two reads $r_1$ and $r_2$ from the same variable occur in a single thread, such that $r_1 \leq r_2$, then any later write of the value read at $r_2$ can replace a dependency from $r_2$ with a dependency from $r_1$ as long as the dependency edge notes that the values read must have been the same. If some write $w$ to the same location again occurs in the same thread, then a similar dependency from $r_2$ can be replaced with a dependency from $w$, as long as the dependency edge notes that $r_1$, $r_2$, and $w$ all share values.

```
x := 1;
r1 := x;
r2 := x;
```



### 6.4.2 Initial Justifications and Single-Execution Rewrites

We mentioned earlier the initial justification of a write, which is the most trivial predicate and set possible to associate with it. The initial justification for any event $(w : \mathrm{W}\ x\ v)$ in the structure is always $(\curlyvee(w), \textsc{data}(w : \mathrm{W}\ x\ v)) \vdash^{\top} (w : \mathrm{W}\ x\ v)$, where the $\textsc{data}()$ function returns a potentially empty set containing the origins of all variables in $v$. The value restriction function is tied to event identifier, and therefore based solely on the position of the event in the structure, while the data function is sensitive to event label. We can rewrite a justification by strengthening, value assignment, and forwarding. These are non-destructive operations: the input justification remains a valid justification.

We are always allowed to strengthen a control dependency using variables introduced before the write, as this can never introduce new behaviours in a purely sequential context. It can, however, make part of the guard redundant and thus remove dependency edges. If we have a branch guard of the form $\alpha \leq \beta$, where $\alpha$ and $\beta$ are unsigned integers, then if we observe $\alpha = 0$ we do not need to observe $\beta$ – the condition will always hold.

**Definition 22** (Strengthening)**.**

$$\begin{aligned}
&\textit{If } (P, D) \vdash^{\psi} (w : W\,x\,v) \textit{ and:}\\
&\quad P' \Rightarrow P\\
&\quad \textit{for all symbols } \alpha \textit{ introduced by } P',\ O(\alpha) < w\\
&\textit{then } (P', D) \vdash^{\psi} (w : W\,x\,v)
\end{aligned}$$

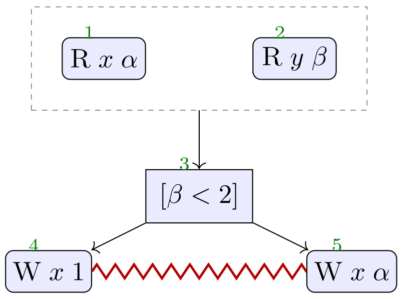
We can also remove data dependencies if we have enough information about the value written. Whenever we can infer the value of a symbolic variable using the contents of its control dependency, for instance if $(w : \mathrm{W}\ x\ \alpha)$ appears under the branch $[\alpha = 0]$, then we can swap the label of $w$ to reference the new value and re-calculate the data dependencies.

**Definition 23** (Value Assignment)**.**

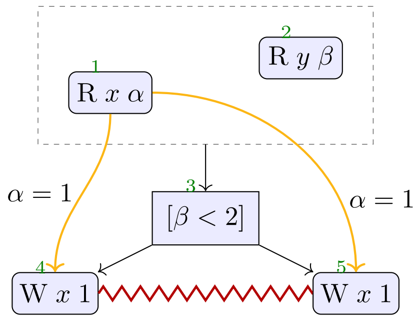
$$\begin{aligned}
&\textit{If } (P, \textsc{data}(w : W\,x\,v)) \vdash^{\psi} (w : W\,x\,v) \textit{ and } P \Rightarrow \alpha = \beta\\
&\textit{then } (P, \textsc{data}(w : W\,x\,v_{[\alpha\backslash\beta]})) \vdash^{\psi} (w : W\,x\,v_{[\alpha\backslash\beta]})
\end{aligned}$$

The combination of strengthening and value assignment can effectively push data dependencies into control dependencies. Whenever we write an unknown value, we can transform this into a check that the value is a particular constant followed by either a write of that constant if the check holds or a write of the unknown value if it does not.

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



This structure initially begins with the pre-justification $([\beta < 2], \{2\}) \vdash (5 : \mathrm{W}\ x\ \alpha)$, which we can weaken to $([\beta < 2 \wedge \alpha = 1], \{2\}) \vdash (5 : \mathrm{W}\ x\ \alpha)$. We can then use the value assignment rule to treat event 5 as a write of the constant 1, giving us $([\beta < 2 \wedge \alpha = 1], \emptyset) \vdash (5 : \mathrm{W}\ x\ 1)$. With the two events under the branch sharing a label, we can remove the $\beta < 2$ clause by similarly weakening the pre-justification for event 4 to $([\beta \geq 2 \wedge \alpha = 1], \emptyset) \vdash (4 : \mathrm{W}\ x\ 1)$ and performing a coproduct.



As in concrete MRD, we allow forwarding operations to rewrite justifications. Load forwarding rewrites justifications and writes to use a copy of a previously read variable $\alpha$ in place of a newly read variable $\beta$, while store forwarding does the same for a copy of a stored value but includes the dependencies of the store.

**Definition 24** (Forwarding)**.**

$$\textit{If } (P, D) \vdash^{\psi} w \textit{ and } (r : R\ x\ \beta) \xrightarrow{F}{}^{\psi_e} (e : R\ x\ \alpha) \in (O(P) \cup D)$$

$$\textit{then } (P_{[\alpha \backslash \beta]}, (D \setminus \{e\}) \cup \{r\}) \vdash^{\psi \wedge \psi_e} (w : W\ x\ v_{[\alpha \backslash \beta]})$$

$$(P, D) \vdash^{\psi} w$$

*If* $(P, D) \vdash^{\psi} w$, $(w : \text{W}\ x\ v) \xrightarrow{F}^{\psi_w} (e : \text{R}\ x\ \alpha) \in (O(P) \cup D)$ *and* $(P_w, D_w) \vdash^{\psi_w} w$

*then* $(P_{[\alpha \backslash v]} \wedge P_w, (D \setminus \{e\}) \cup D_w \cup \{w\}) \vdash^{\psi \wedge \psi_e \wedge \psi_w} w$

$(P, D) \vdash^{\psi \wedge} w$

In Fig. 53 we show the prior forwarding rules impacting the justifications for a write. The initial justification is $([\beta < 3], \{3\}) \vdash (4 : \text{W}\ x\ \beta)$. As $2 \xrightarrow{F}^{\alpha=\beta} 3$ we gain a new justification in which we swap all instances of $\beta$ for $\alpha$:

- $([\beta < 3], \{3\}) \vdash (4 : \text{W}\ x\ \beta)$.
- $([\alpha < 3], \{2\}) \vdash^{\alpha=\beta} (4 : \text{W}\ x\ \alpha)$.

Similarly, as $1 \xrightarrow{F}^{\alpha=1} 2$, we expand our set of justifications once again, this time removing the control dependency by relabelling $\beta$ to 1:

- $([\beta < 3], \{3\}) \vdash (4 : \text{W}\ x\ \beta)$.
- $([\alpha < 3], \{2\}) \vdash^{\alpha=\beta} (4 : \text{W}\ x\ \alpha)$.
- $(\top, \{1\}) \vdash^{\alpha=\beta=1} (4 : \text{W}\ x\ 1)$.

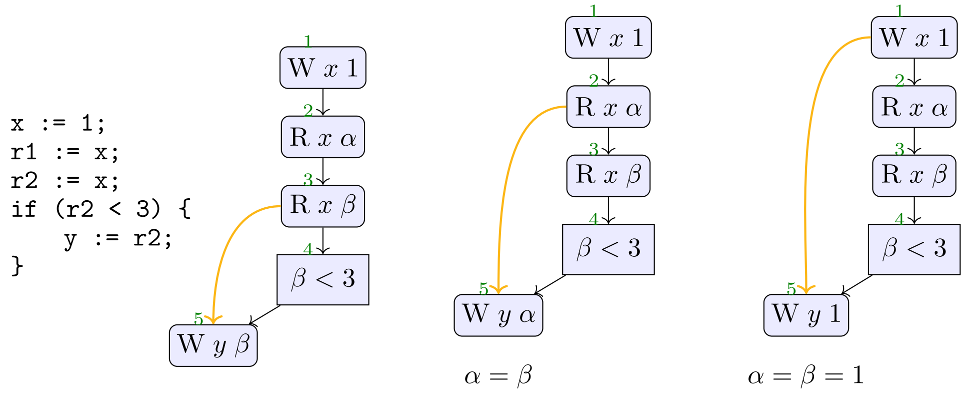


Figure 53: 3 choices of justification for event 5 with their respective data dependencies illustrated.

## 6.4.3 Coproduct

Coproduct in Symbolic MRD is not restricted to computation in a specific order, as it is with the justification-up-to function. Neither does it inherently simplify dependencies. Rather, it combines them.

Whenever two write events have the same label, we can combine their dependencies. If we have some write $(w_1 : \mathrm{W}\ x\ 1)$ which can occur if the value $\alpha$ is strictly greater than 1, and some write to the same location $(w_2 : \mathrm{W}\ x\ 1)$ which can occur if $\alpha$ is less than or equal to one, we can combine these dependencies and say that $w_1$ will happen regardless of the value of $\alpha$. If $w_1$ will happen when $\alpha = 1$ and $w_2$ will happen when $\beta = 1$, we can also combine these dependencies into $\alpha = 1 \vee \beta = 1$. In the first case each write is reliant on fewer values and in the second on more values, but both are valid cases of combining dependencies.

It may be the case that to perform $(w_1 : \mathrm{W}\ x\ \alpha)$ we must load a value $\alpha$ in one execution, but to perform $(w_2 : \mathrm{W}\ x\ \beta)$ in another execution we must first load $\beta$. If these symbolic variables both refer to an unknown value read from $y$, then each write is simply copying the value of $y$ into $x$ and the two should be treated as equivalent. Whenever variable $\alpha$ is introduced in conflict with variable $\beta$, we allow a relabelling operation $\Lambda$ to change references to $\alpha$ into references to $\beta$ and vice versa provided they may be assigned the same set of possible values.

**Definition 25** (Similarity-Preserving Relabelling)**.** *A function $\Lambda$ from symbolic variables in a justification $(P, D) \vdash^{\psi} w$ to symbolic variables originating in conflict with $w$ is* similarity-preserving *iff:*

$$\Lambda(\alpha) = \beta \Rightarrow (\forall v.\ (P \Rightarrow (\alpha \neq v)) \Leftrightarrow (\psi \Rightarrow (\beta \neq v))$$

*Where relabelling is $\leq$-preserving with respect to origin.*

Given $\mathbb{L}$ such that $\forall l \in \mathbb{L}.\ (P_l, D_l) \vdash^{\psi_l} w_l$, if we can define a set of label similarity preserving relabelling functions $\Lambda_l$ over each $(P_l, D_l) \vdash^{\psi_l} w_l$, a write $w$ and, an event set $D$ such that:

- $(P_l, D_l) \vdash^{\psi_l} (w_l : \mathrm{W}\ x\ v)$
- $(w_l : \mathrm{W}\ x\ v) = \Lambda_l^{-1}(w)$
- $\Lambda_l^{-1}(D) \in \uparrow D_l$

Then

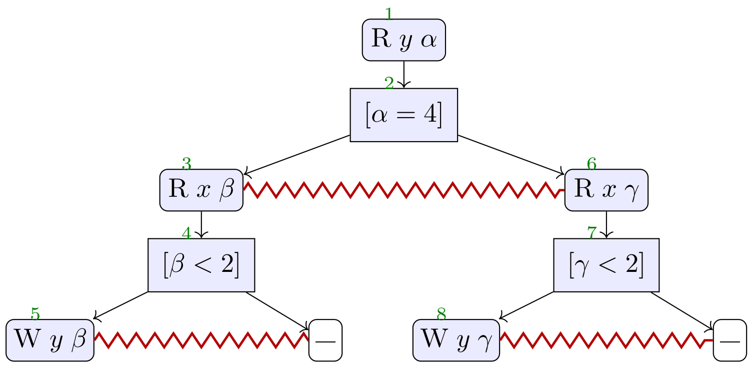
$$\text{Let } P = \bigvee_{l \in \mathbb{L}} \Lambda_l(P_l)$$

$$\text{Let } \psi = \bigwedge_{l \in \mathbb{L}} \psi_l$$

For all $l \in \mathbb{L}$ add the justifying predicate $(\Lambda_l^{-1}(P), \Lambda_l^{-1}(D)) \vdash^{\psi} (w_l : \mathrm{W}\ x\ v)$

We now give a concrete example.

```
r1 := y;
if (r1 = 3) {
    r2 := x;
    y := r2;
} else {
    r2 := x;
    y := r2;
}
```



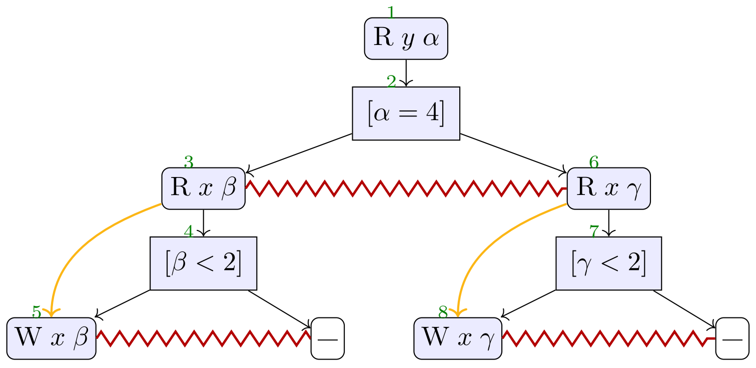
In this program, the statements which copy the value of $x$ into $y$ should run regardless of which branch is taken, so the branch can be removed. This means events 5 and 8 should have the control dependency $\top$, but retain their data dependencies.

Our initial justifications are $([\alpha = 4 \wedge \beta < 2], \{3\}) \vdash 5$ and $([\alpha \neq 4 \wedge \gamma < 2], \{6\}) \vdash 8$. We cannot create a weakened control dependency forcing $\beta$ and $\gamma$ to be equal, as the events in which they appear are in conflict and therefore cannot both be seen in the same execution. Instead, we define a relabelling. We first let $\Lambda_1(\beta) = \gamma$ and $\Lambda_2$ be its inverse, then verify that $O(\beta) \sim O(\alpha)$. Both of these are reads of $x$ with no events $\leq$ before or after them, thus $\Lambda_1$ and $\Lambda_2$ are similarity-preserving.

We finish with the coproduct operation:

- $([\alpha = 4 \wedge \beta < 2], \{3\}) = (P_1, D_1)$ and $([\alpha \neq 4 \wedge \gamma < 2], \{6\}) = (P_2, D_2)$
- $\Lambda_1(5 : \text{W } x\ \beta) = \Lambda_2(8 : \text{W } x\ \gamma) = (w : \text{W } x\ \gamma)$
- $\Lambda_1(\{3\}) = \Lambda_2(\{6\}) = D = \{(r : \text{R } x\ \gamma)\}$

This leaves us with $P = \Lambda_1([\alpha = 4 \wedge \beta < 2]) \vee \Lambda_2([\alpha \neq 4 \wedge \gamma < 2]) = [\gamma < 2]$ Allowing us to add the justification $(\Lambda_1^{-1}([\gamma < 2]), \Lambda_1^{-1}(\{r\})) \vdash (5 : \text{W } x\ \beta)$, which simplifies to $([\beta < 2], \{3\}) \vdash (5 : \text{W } x\ \beta)$.



All that remains is to convert these justifications into DP edges via freezing.

### 6.4.4 Freezing

Given a justifying predicate for $w$, the freeze operation converts this predicate into a set of events which should be DP before $w$.

To convert a predicate to an event set we use the *origin* function $O$, which when applied to predicate $P$ gives the origins of all variables in $P$. If we have a justification $(P, D) \vdash^{\psi} (w : \mathrm{W}\ x\ e)$, we convert this into a dependency edge by taking the origin of the events in $P$, adding all read events in $D$, and establishing $e \xrightarrow{\text{DP}} w$ for all $e$ in the resulting set.

When returning this information to the execution, we also return both $P$ and $\psi$ to check against the execution condition and the write label $l$, in case the write has been relabelled.

Let $\text{DP}_R = \{(r, w) \mid r \in R\}$

$freeze(w) \triangleq \{(\text{DP}_R, l, \psi) \mid \exists (P, D) \vdash^{\psi_P} (w : l).\ R = (O(P) \cup D) \wedge \psi = P \wedge \psi_P\}$

### 6.4.5 Program-Wide Guarantees

The local conditions store guarantees which are bounded in time – the result of checking a guard, for instance, provides guarantees only for events syntactically after the guard. There are also guarantees about program behaviour which cannot be obtained by local reasoning alone. For instance, value range analysis may find that all writes within a program can only write a restricted set of values, and not all values which can be stored in a type.

We represent program-wide guarantees using the $\varPsi$ parameter of a structure, which is created and modified by the extended semantics $(\!|P|\!)_{n\ 0\ \emptyset\ \top}$.

$$(\!|P|\!)_{n\ 0\ \emptyset\ \top} = (E, <, \sqsubseteq, \curlyvee, \varPsi)$$

Where:

- Whenever $[\![P]\!]_{n\ 0\ \emptyset\ \top} = (E, <, \sqsubseteq, \curlyvee)$, then $(E, <, \sqsubseteq, \curlyvee, \top)$ is in $(\!|P|\!)_{n\ 0\ \emptyset\ \top}$
- Whenever $S = (E, <, \sqsubseteq, \curlyvee, \varPsi)$ is in $(\!|P|\!)_{n\ 0\ \emptyset\ \top}$ and $\varPsi'$:
  1. holds over all complete executions of $S$, and
  2. continues to hold for all executions of the symbolic event structure $(E, <, \sqsubseteq, \curlyvee, \varPsi \wedge \varPsi')$,

  then $(E, <, \sqsubseteq, \curlyvee, \varPsi \wedge \varPsi')$ is in $(\!|P|\!)_{n\ 0\ \emptyset\ \top}$.

```
                      x := 1;
                      r1 := x;
                      r2 := y;
                      if (r2 = 2) {
                          r3 := z;
r0 := a;                  if (r3 < 10) {
if (r0 = 1) {                 a := r1;
y := 2;                   }
}                 ||  } else {
z := 1;           ||      r3 := z;
                          if (r3 ≥ 10) {
                              y := 1;
                          } else {
                              a := r3;
                          }
                      }
```

Allowed: $\texttt{r0} = 1 \wedge \texttt{r1} = 1 \wedge \texttt{r2} = 2$

Figure 54: The litmus test FWD-STRENGTHEN-LIFT

The new execution set $S'$ may differ because the inclusion of a program-wide guarantee alters the calculation of dependencies. Just as control dependencies may simplify event labels and remove data dependencies, program-wide guarantees may simplify control flow and remove control dependencies. If we have a branch guarded by $\alpha < 3$, for instance, but the only values written in the entire program are 0 and 1, then we are free to remove the guard.

**Definition 26** (Weakening)**.**

$$\textit{If } (P' \wedge P, D) \vdash^{\psi} (w : W\,x\,v) \textit{ and } \Psi \Rightarrow P$$
$$\textit{then } (P', D) \vdash^{\psi} (w : W\,x\,v)$$

### 6.4.6 Worked Examples

#### Putting it All Together

In Fig. 54 we give the less-trivial program FWD-STRENGTHEN-LIFT and behaviour of interest, in which one thread observes a sequence of optimisations performed in the other.

The right-hand thread first lifts `r3 := z` above the if-statement, as it is performed in both branches. It then alters the line `a := r1` to `a := 1`, since `r1` is loading a value stored earlier in the same thread.

```
if (r2 = 2) {
    r3 := z;
    if (r3 < 10) {
        a := r1;
    }
} else {
    r3 := z;
    if (r3 ≥ 10) {
        y := 1;
    } else {
        a := r3;
    }
}
```

$\longrightarrow$

```
r3 := z;
if (r2 = 2) {
    if (r3 < 10) {
        a := 1;
    }
} else {
    if (r3 ≥ 10) {
        y := 1;
    } else {
        a := r3;
    }
}
```

The writes to $a$ now have nearly-inverse conditions: if $\texttt{r3} < 10 \wedge \texttt{r2} = 2$ then write 1 to $a$, and if $\texttt{r3} < 10 \wedge \texttt{r2} \neq 2$, then write `r3` to $a$. If we were to check the value inside `r3` after it loads from $z$ and find 1, then we are always going to write 1 to $a$ regardless of what happens to `r2`, so we can restructure the control flow to perform that write earlier.

```
r3 := z;
if (r2 = 2) {
    if (r3 < 10) {
        a := 1;
    }
} else {
    if (r3 ≥ 10) {
        y := 1;
    } else {
        a := r3;
    }
}
```

$\longrightarrow$

```
r3 := z;
if (r3 = 1) {
    a := 1;
} else if (r2 = 2) {
    ...
}
```

This might not strike us as a good idea, since we introduce an extra branch in most cases, but it is a valid transformation of the control flow. With the write to $a$ now only needing to observe a value from $z$, which the left-hand thread can perform early as a write of a constant, there is an acyclical set of dependencies which result in this outcome.

We give the symbolic event structure corresponding to the program in Figure 54.

When running litmus tests with a focus on a specific outcome, we generally begin by drawing on the RF edges which result in value constraints compatible with that outcome. In this test, we need an RF edge $3 \xrightarrow{\text{RF}} 8$ to observe a 2 at `r2`, forcing us into the left-hand branch at the $\beta = 2$ check and leaving us with only $12 \xrightarrow{\text{RF}} 1$ to provide a write for event 1. This constrains the value $v$ to be equal to $\alpha$, which means we should read a value of 1 at event 7 to ensure $\alpha = v = 1$, which fortunately is possible via the edge $6 \xrightarrow{\text{RF}} 7$. The only remaining read without a write is event 10, and we are forced to include the edge $4 \xrightarrow{\text{RF}} 10$ to complete the execution.

These RF edges are shown in Figure 56.

We then draw the simplest possible dependency edges. These are the edges we gain from

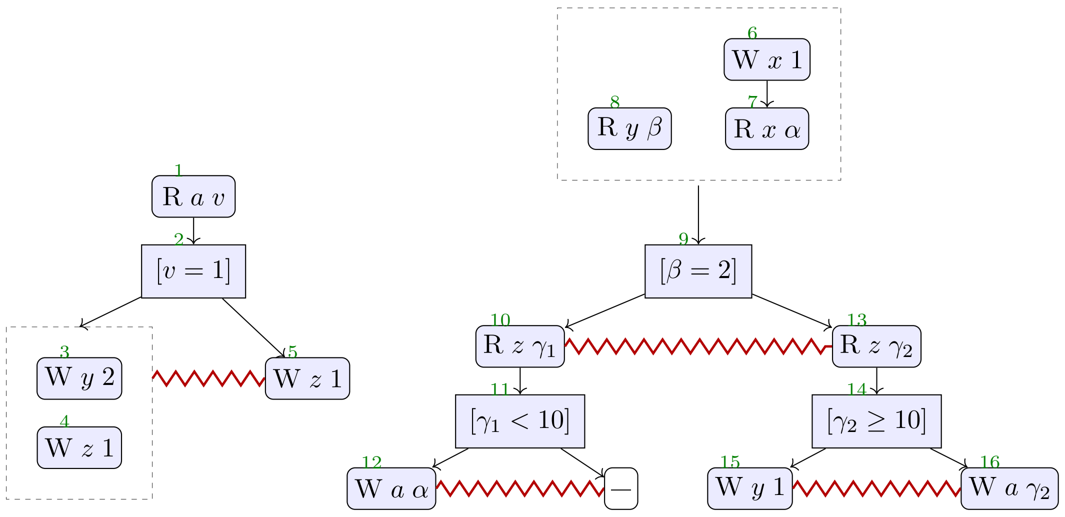


Figure 55: The symbolic event structure for the FWD-STRENGTHEN-LIFT program.

freezing the syntactic control dependencies, and adding in any un-forwarded data dependencies. The simplest justifications we have for all writes in this execution are:

- $([v = 1], \emptyset) \vdash (3 : \text{W } y\ 2)$
- $([v = 1], \emptyset) \vdash (4 : \text{W } z\ 1)$
- $(\top, \emptyset) \vdash (6 : \text{W } x\ 1)$
- $([\beta = 2], \{7\}) \vdash (12 : \text{W } a\ \alpha)$

When drawn onto the structure, as in Figure 57, we can see many $\text{DP} \cup \text{RF}$ cycles.

In particular, we can observe the cycles:

- $3 \xrightarrow{\text{RF}} 8 \xrightarrow{\text{DP}} 12 \xrightarrow{\text{RF}} 1 \xrightarrow{\text{DP}} 3$
- $4 \xrightarrow{\text{RF}} 10 \xrightarrow{\text{DP}} 12 \xrightarrow{\text{RF}} 1 \xrightarrow{\text{DP}} 4$

We can break the second cycle by coproduct, since $1 \xrightarrow{\text{DP}} 4$ is relatively simple to remove.

1. $([v = 1], \emptyset) \vdash (4 : \text{W } z\ 1)$
2. $([v \neq 1], \emptyset) \vdash (5 : \text{W } z\ 1)$

We can combine these two justifications into $([v = 1 \vee v \neq 1], \emptyset) \vdash 4$, making event 4 independent.

However, since we cannot make event 3 independent, breaking the first cycle is harder. The only edge left in the cycle that we could remove is $8 \xrightarrow{\text{DP}} 12$.

Converting the explanation for the program transformation into the structure, we need to use the following reasoning:

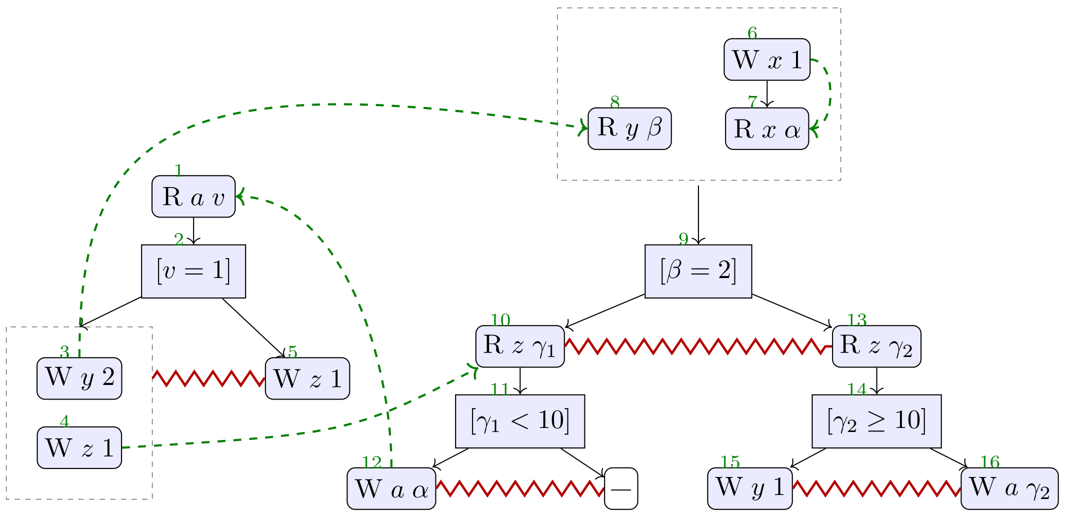


Figure 56: The necessary RF edges for the target execution of this structure.

- Event 16 is also a write to $a$, of the value $\gamma_2$.
- By constraining both $\gamma_1$ and $\gamma_2$ to be $\alpha$, can relabel event 16 to be a write of $\alpha$.
- With one write with the label W $a$ $\alpha$ in each branch of the guard, we can remove the guard condition during coproduct.

We proceed with the strengthening step, giving us the justifications

- $([\beta = 2 \wedge \gamma_1 < 10 \wedge \gamma_1 = \alpha], \{7\}) \vdash (12 : \text{W } a\ \alpha)$
- $([\beta \neq 2 \wedge \gamma_2 < 10 \wedge \gamma_2 = \alpha], \{13\}) \vdash (16 : \text{W } a\ \gamma_2)$

And since the control dependency for event 16 now asserts that $\gamma_2 = \alpha$, using the Value Assignment step of Definition 23 we relabel the event into W $a$ $\alpha$ and change the data dependency accordingly, giving us

- $([\beta = 2 \wedge \gamma_1 < 10 \wedge \gamma_1 = \alpha], \{7\}) \vdash (12 : \text{W } a\ \alpha)$
- $([\beta \neq 2 \wedge \gamma_2 < 10 \wedge \gamma_2 = \alpha], \{7\}) \vdash (16 : \text{W } a\ \alpha)$

Moving on to the coproduct step, we only need to define a similarity-preserving relabelling $\Lambda$ between these justifications. Since both $\gamma_1$ and $\gamma_2$ are constrained by the predicate to be $\alpha$, their originating reads must be label similar, and therefore $\Lambda_1(\gamma_1) = \gamma_2$ and $\Lambda_2(\gamma_2) = \gamma_1$ is similarity-preserving.

- $([\beta = 2 \wedge \gamma_1 < 10 \wedge \gamma_1 = \alpha], \{7\}) = (P_1, D_1)$ and $([\beta \neq 2 \wedge \gamma_2 < 10 \wedge \gamma_2 = \alpha], \{7\}) = (P_2, D_2)$

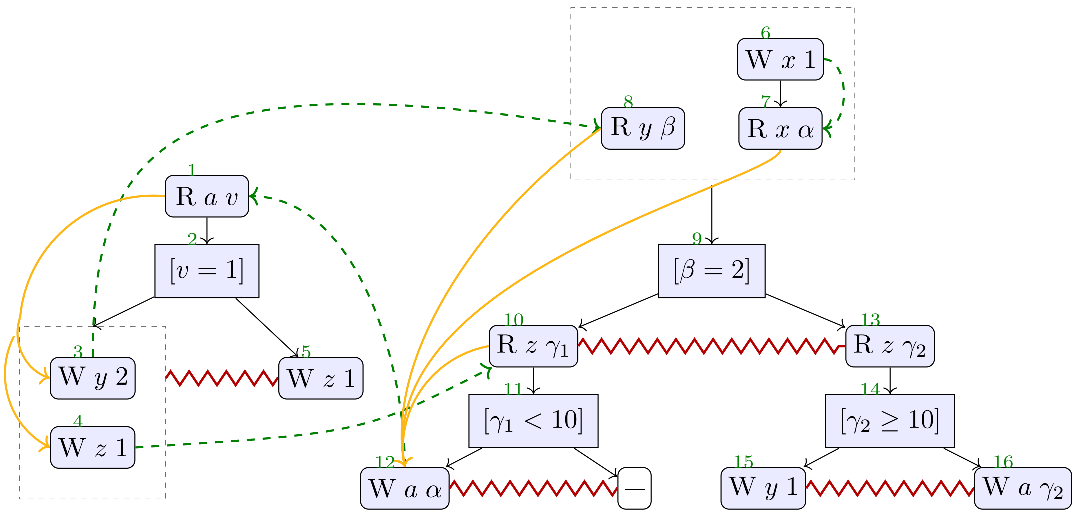


Figure 57: The simplest possible dependencies for the target execution.

- $\Lambda_1(12 : \text{W } a\ \alpha) = \Lambda_2(16 : \text{W } a\ 1) = (w : \text{W } a\ \alpha)$
- $\Lambda_1(\{7\}) = \Lambda_2(\{7\}) = D = \{(7 : \text{R } x\ \alpha)\}$
- $\Lambda_2(P_2) \vee P_1 = [(\beta \neq 2 \wedge \gamma_1 < 10 \wedge \gamma_1 = \alpha) \vee (\beta = 2 \wedge \gamma_1 < 10 \wedge \gamma_1 = \alpha)] = [\gamma_1 < 10 \wedge \gamma_1 = \alpha]$

This gives the justification $([\gamma_1 < 10 \wedge \gamma_1 = \alpha], \{7\}) \vdash (12 : \text{W } a\ \alpha)$, removing the reference to $\beta$ and therefore the $8 \xrightarrow{\text{DP}} 12$ link in the cycle.

We can continue, though, since the write W $x$ 1 at event 6 triggers a forwarding operation which replaces all instances of $\alpha$ with 1, via Definition 24. This gives us a final justification of $([\gamma_1 = 1], \{6\}) \vdash^{[\alpha=1]} (12 : \text{W } a\ 1)$, removing the remaining $7 \xrightarrow{\text{DP}} 12$ edge as well.

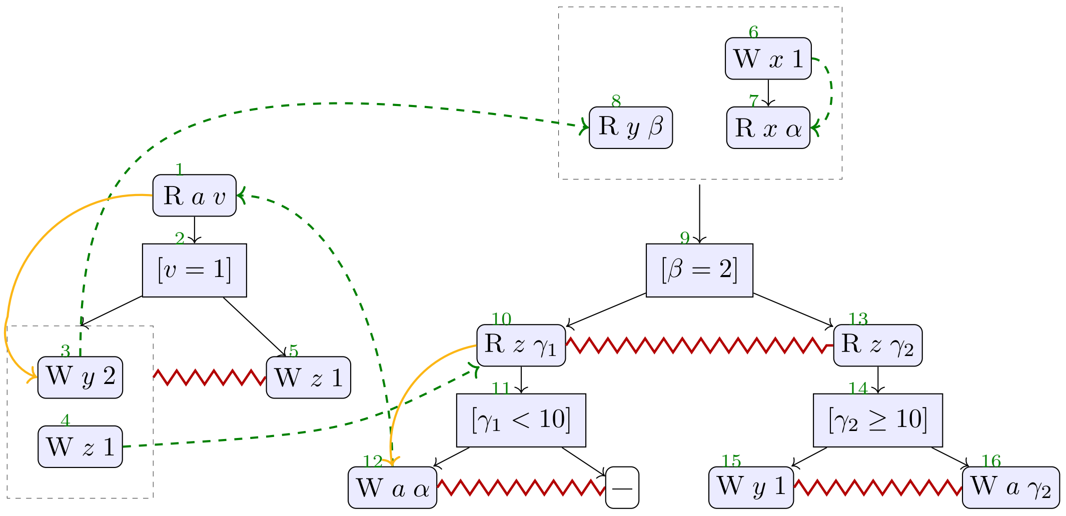
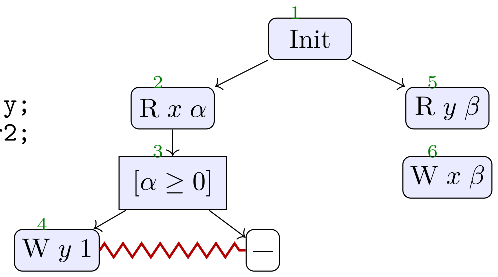


Figure 58: The target execution after dependency calculation.

**Value Range Analysis in JCTC1**

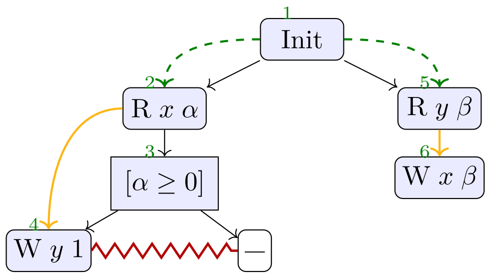
```
r1 := x;
if (r1 >= 0) {          ||    r2 := y;
    y := 1;                   x := r2;
}
```

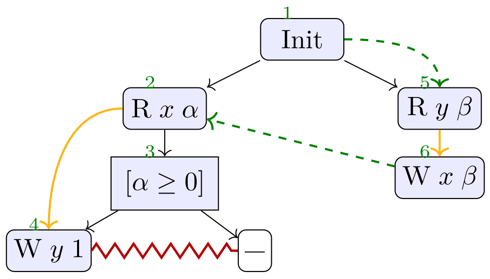
Allowed: r1 = r2 = 1

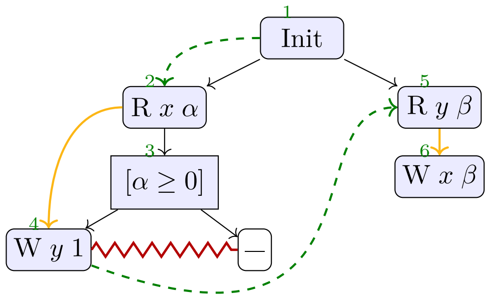


The explanation given for this test case is that an inter-thread analysis pass could determine that a value less than 0 is never written to $x$, thus the guard can be optimised away.

We are unable to remove the clause $\alpha \geq 0$ from our control dependency for event 3, and the data dependency for event 6 is trivially unmodifiable. This leaves us with the following executions in $[\![P]\!]_{1\ 0\ \emptyset\ \top}$:



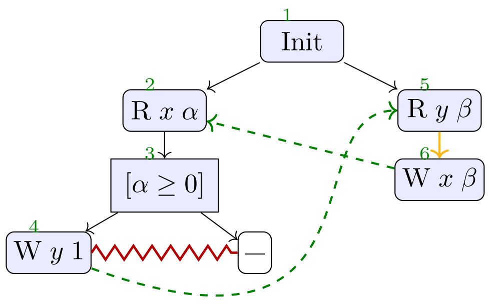




We want to find an execution in which $4 \xrightarrow{\text{RF}} 5$ and $6 \xrightarrow{\text{RF}} 2$, but this will cause a cycle given the non-removable $2 \xrightarrow{\text{DP}} 4$ and $5 \xrightarrow{\text{DP}} 6$ edges.

In all of these executions, the only values written to any location are 0 or 1. The first and second have the condition $[\alpha = \beta = 0]$, while the third has $[\alpha = 0 \wedge \beta = 1]$. This allows us to try again with the program guarantee $\Psi = [\forall v.\ v \in \{0, 1\}]$.

We can now use the weakening rule on the initial justification $([\alpha \geq 0], \emptyset) \vdash (4 : \text{W}\ y\ 1)$, as $\Psi \Rightarrow \alpha \geq 0$, to gain the new pre-justification $(\top, \emptyset) \vdash (4 : \text{W}\ y\ 1)$. This makes event 4 independent, and allows a fourth execution:

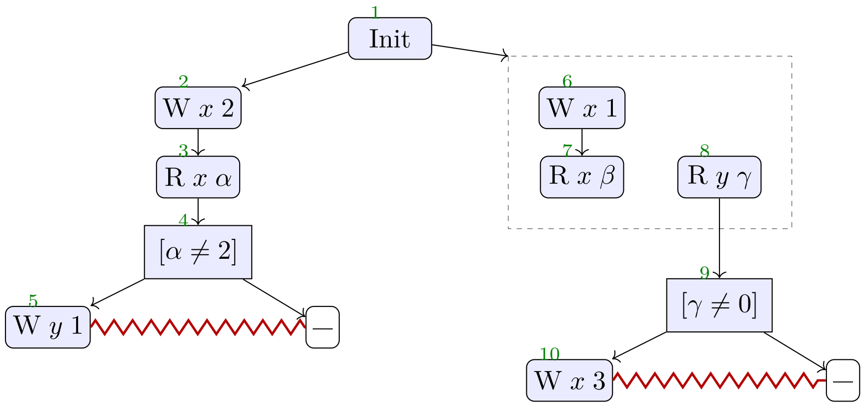


**OOTA7**

```
1: x := 2;              4: x := 1;
2: r1 := x;             5: r2 := x;
if (r1 != 2){     ||    6: r3 := y;
  3: y := 1;            if (r3 != 0){
}                         7: x := 3;
                        }
```

We now run the OOTA7 litmus test. This is the test in which the Promising semantics permits the behaviour $r1 = 3 \wedge r2 = 2 \wedge r3 = 1$, by allowing line 2 to read from line 7, line 5 from line 1, and line 6 from line 3.

As the behaviours are dependent on the initialising writes, we prepend an Init event to the program. This is purely syntactic sugar, and we treat it as a regular write of 0 to all locations.



If we can create an execution where [$\alpha = 3 \wedge \beta = 2 \wedge \gamma = 1$], then we have permitted the OOTA behaviour. This requires reads-from edges $10 \xrightarrow{\text{RF}} 3$, $2 \xrightarrow{\text{RF}} 7$ and $5 \xrightarrow{\text{RF}} 8$, so our execution $E$ looks like this prior to dependency calculation:

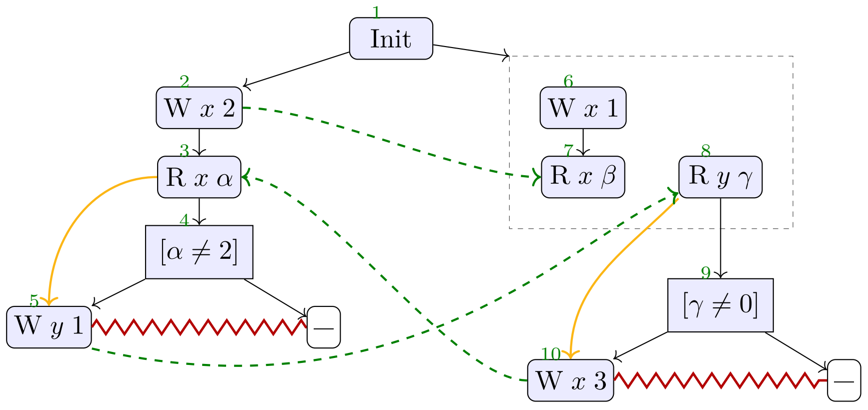
$$E = \{1,2,3,4,5,6,7,8,9,10\}$$
$$\text{RF}_E = \{(10,3),(2,7),(5,8)\} \quad \text{DP}_E = \emptyset$$
$$\psi_E = [\alpha \neq 2 \wedge \gamma \neq 0]$$

To determine the presence of a cycle, we need to know the dependency relations for events 10, 2, and 5.

Event 2 lacks any control or data dependencies at all, being a write of a constant without any guard. Our initial justification for event 5 is $([\alpha \neq 2], \emptyset) \vdash (5 : \text{W } y\ 1)$, which we can only forward to $([2 \neq 2], \{2\}) \vdash^{\alpha=2} (5 : \text{W } y\ 1)$. This is unsatisfiable, as whenever we freeze it we will be forced to add $\alpha = 2 \wedge \alpha \neq 2$ to any execution which uses it, so our only remaining justification is $([\alpha \neq 2], \emptyset) \vdash (5 : \text{W } y\ 1)$ For event 10, we start with $([\gamma \neq 0], \emptyset) \vdash (10 : \text{W } x\ 3)$. Once again our only forwarding option, taking $\gamma$ from the initialising write instead of from memory, leaves us with an unsatisfiable control dependency and we must content ourselves with only $([\gamma \neq 0], \emptyset) \vdash (10 : \text{W } x\ 3)$.

Freezing these edges gives us $\text{DP}'_E = \{(3,5),(8,10)\}$.

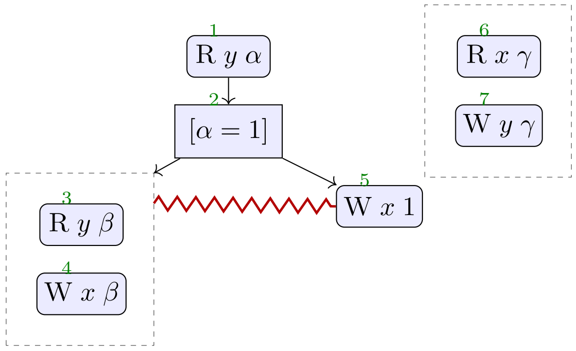
Drawing these dependencies onto $E$ results in the following:



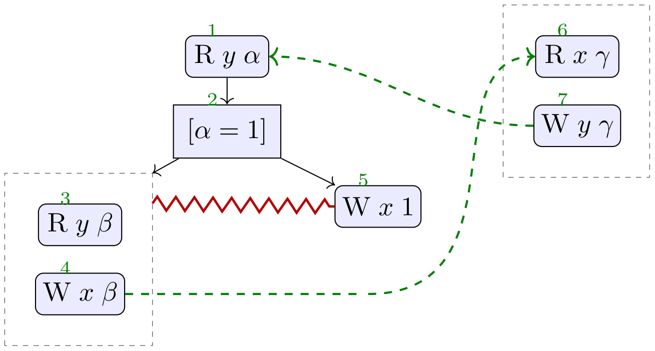
There is a clear cycle in $3 \xrightarrow{\text{DP}} 5 \xrightarrow{\text{RF}} 8 \xrightarrow{\text{DP}} 10 \xrightarrow{\text{RF}} 3$, so this execution would be discarded.

### Hotspot Optimisation

```
1: r1 := y;
if (r1 = 1) {
  2: r2 := y;     ||   5: r3 := x;
  3: x := r2;     ||   6: y := r3;
} else {
  4: x := 1;
}
```



The Hotspot optimisation is permitted if we can construct an execution in which $[\alpha = \beta = 1]$. We can do this by additionally asserting that $\gamma = 1$ and drawing reads-from edges $4 \xrightarrow{\text{RF}} 6$ and $7 \xrightarrow{\text{RF}} 1$, as shown:



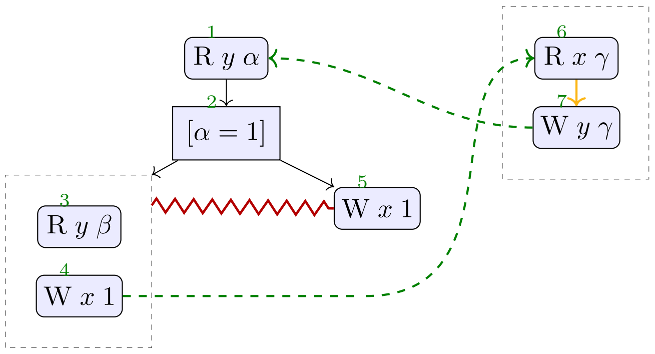
This is forbidden if both $6 \xrightarrow{\text{DP}} 7$ and $1 \xrightarrow{\text{DP}} 4$, causing a $1 \xrightarrow{\text{DP}} 4 \xrightarrow{\text{RF}} 6 \xrightarrow{\text{DP}} 7 \xrightarrow{\text{RF}} 1$ cycle. We cannot remove the $6 \xrightarrow{\text{DP}} 7$ data dependency, as the only available justification for event 7 is $(\top, \{6\}) \vdash 7$. Our only option is to try making event 4 independent.

Event 4 begins with initial justification $([\alpha = 1], \{3\}) \vdash (4 : \text{W } x \ \beta)$. Since our label refers to $\beta$ and $(1 : \text{R } y \ \alpha) \overset{F}{\longrightarrow}^{\alpha=\beta} (3 : \text{R } y \ \beta)$, we can replace all instances of $\beta$ with $\alpha$ provided we annotate the justification with the assumption $[\alpha = \beta]$. This gives us $([\alpha = 1], \{1\}) \vdash^{\alpha=\beta} (w : \text{W } x \ \alpha)$. We can then use the value assignment rule to relabel $w$ as a write of 1, since our control dependency has already established that $\alpha$ must be 1, resulting in the justification $([\alpha = 1], \emptyset) \vdash^{\alpha=\beta} (4 : \text{W } x \ 1)$.

We now have a pair of isomorphic events below the branch at 2:

- $([\alpha = 1], \emptyset) \vdash^{\alpha=\beta} (4 : \text{W } x \ 1)$
- $([\alpha \neq 1], \emptyset) \vdash (5 : \text{W } x \ 1)$

This gives us the new dependency $([\alpha = 1 \vee \alpha \neq 1], \emptyset) \vdash^{\alpha=\beta} (4 : \mathrm{W}\ x\ 1)$, which makes event 4 independent in executions which observe the same value for $\alpha$ and $\beta$. This breaks the DP $\cup$ RF cycle and permits the behaviour.



## 6.5 Equivalence to Concrete Value MRD

In the proceeding, we show that any symbolic MRD structure can, given a finite value range, be expanded into a concrete MRD structure with the same ordering edges.

We do this using an instantiation function $\mathcal{I}$ from symbolic executions to sets of concrete value executions. If any execution of $[\![P]\!]_{n\ \rho\ \emptyset\ \top}$ expands exclusively into a set of legal executions of $[\![P]\!]_{n\ \rho\ \emptyset}$, then symbolic MRD does not permit behaviours forbidden by concrete MRD.

### Defining Instantiation

We construct the instantiation function over events by enforcing that an event and its instantiations must come from interpretations of the same statement, such that whenever $[\![Q]\!]_{n\ \rho_s\ \kappa_s\ \varphi} = e_s \bullet \kappa_s(\rho_s')$, where $Q$ is a sub-program of $P$ and $e_s$ is not a branch, and $[\![Q]\!]_{n\ \rho_c\ \kappa_c} = e_c \bullet \kappa_c(\rho_c)$, then $e_c \in \mathcal{I}(e_s)$.

From this we know that $\mathcal{I}(e)$ cannot be empty for any $e$, due to the culling of unreachable program branches, that it preserves conflict, and that it preserves all relations derived directly from program order (in particular branch and preserved program orders). We lift $\mathcal{I}$ to sets of events by returning all sets which contain one instantiation of each event in its input, as $\mathcal{I}(E) = \{E' \mid \forall e \in E.\ \exists! e' \in E'.\ e' \in \mathcal{I}(e) \wedge \forall e' \in E'.\ \exists e \in E.\ e' \in \mathcal{I}(e)\}$.

Instantiation may be additionally parameterised with a mapping from symbolic to concrete values $V \in A \times \mathcal{V}$ for some symbol set $A$. This is written $\mathcal{I}_V$, where $\mathcal{I}_V(e)$ is well-formed whenever $V$ contains a mapping for every variable introduced $\sqsubseteq$-before or at $e$ and returns a single event $e'$. This is the unique event reachable from the concrete MRD interpretation of the program containing $e$ generated by the same statement.

**Lemma 9** (Bijective Indexed Instantiation)**.** *For any symbolic event $e$, the instantiation $\mathcal{I}_V(e)$ is a single event if $V$ contains a value assignment for every variable introduced $\sqsubseteq$-before $e$.*

*Proof.* Each invocation of the semantic interpretation function in concrete MRD generates a single event, so if the function $\mathcal{I}_V$ traverses a unique path through the function and terminates at a unique point then it must arrive at a unique event. The semantic interpretation function branches on parallel composition and read appends, thus these are the cases in which the instantiation must take the correct path.

If $P = P_1||P_2$, then $e$ is either in $P_1$ or $P_2$. For the sub-program $Q$ for which $[\![Q]\!]_{n\ \rho^s\ \kappa^s\ \varphi} = e \bullet \kappa^s(\rho^{s'})$, if $Q$ is a sub-program of $P_1$ then $\mathcal{I}_V(e)$ is created in $[\![P_1]\!]_{n\ \rho^c\ \kappa^c}$. Otherwise it is created in $[\![P_2]\!]_{n\ \rho^c\ \kappa^c}$.

If $P =$ `r1 := x;` $P'$ then $\mathcal{I}_V(e)$ is somewhere in $\Sigma_{v\in V}(e_v : \mathrm{R}\ x\ v) \bullet \kappa^c(\rho^c[r_1 \mapsto v])$. This means there must be some $v$ for which either $\mathcal{I}_V(e) = e_v$ or $\mathcal{I}_V(e) \in \kappa(\rho[r_1 \mapsto v])$. The corresponding symbolic MRD structure will have the form $(e' : \mathrm{R}\ x\ \alpha) \bullet \kappa_s(\rho_s)$, and if $V$ is well-formed then it must contain a pair $(\alpha, v')$ associating the symbol $\alpha$ with the value $v'$. Choosing $v = v'$, we know that $\mathcal{I}_V(e)$ must be $e'_v$ or in $\kappa(\rho[r_1 \mapsto v'])$, which is the function call $[\![P']\!]_{n\ \rho^c[r_1\mapsto v]\ \kappa^{c'}}$.

□

We say that a predicate $P$ is *consistent with* a value mapping $V$ if $P_{[\alpha\backslash v]}$ for all $(\alpha, v)$ in $V$ is satisfiable.

## Equivalence of Justification

We can now move on to showing that instantiation preserves dependency. In order to obtain a concrete MRD structure by expansion of a symbolic MRD structure, we need to know that whenever there is some dependency set $E$, where $e \xrightarrow{\text{DP}} w$ for all $e$ in $E$, in a symbolic execution then there will be a concrete execution in which $\mathcal{I}_V(E)$ is the dependency set for $\mathcal{I}_V(w)$.

We do this through the construction of the respective justification relations, beginning with forwarding and continuing with coproduct.

**Lemma 10** (Forwarding)**.** *For any write $w$, value mapping $V$ and instantiation $\mathcal{I}_V(w)$, whenever $C \in F(\mathcal{I}_V(w))$ there exists a symbolic justification $(P, D) \vdash^{\psi} w$ where $C = \mathcal{I}_V(O(P) \cup D)$ and $\psi \wedge P$ is consistent with $V$.*

*Proof.* For any given read $(r : \mathrm{R}\ x\ \alpha)$, we can simulate an initial justification edge from every $\mathcal{I}(r)$ by strengthening every pre-justification $(P, D) \vdash w$ to $(P_{[\alpha\backslash v]} \wedge \alpha = v, D) \vdash w$ for each

$v \in V$. Extending this reasoning, we give each write a set of justifying predicates of the form $\alpha = v_1 \wedge \beta = v_2 \wedge ... \wedge \omega = v_n$ for each symbolic variable introduced in $r$ where, for some $r' = \mathcal{I}_V(r)$ and $w' = \mathcal{I}_V(w)$, we have $r' \sqsubseteq w'$. There cannot be any remaining symbolic variables, as the interpretation function which generates these predicates is tied to program order – we would need to interpret `r1 := x` before interpreting `if (f(r1)) {S}` for the symbolic value created in one to appear in the predicate created in another.

Discarding unsatisfiable predicates, we arrive at one justifying predicate per value set observable in an execution in which $w$ occurs, with each predicate pointing to an instance of $w$ with a concrete-value label due to the $P \Rightarrow \alpha = v$ relabelling clause. We index these pre-justifications with the value mapping $V$.

This is isomorphic to branching at every read, illustrated in Fig. 59.

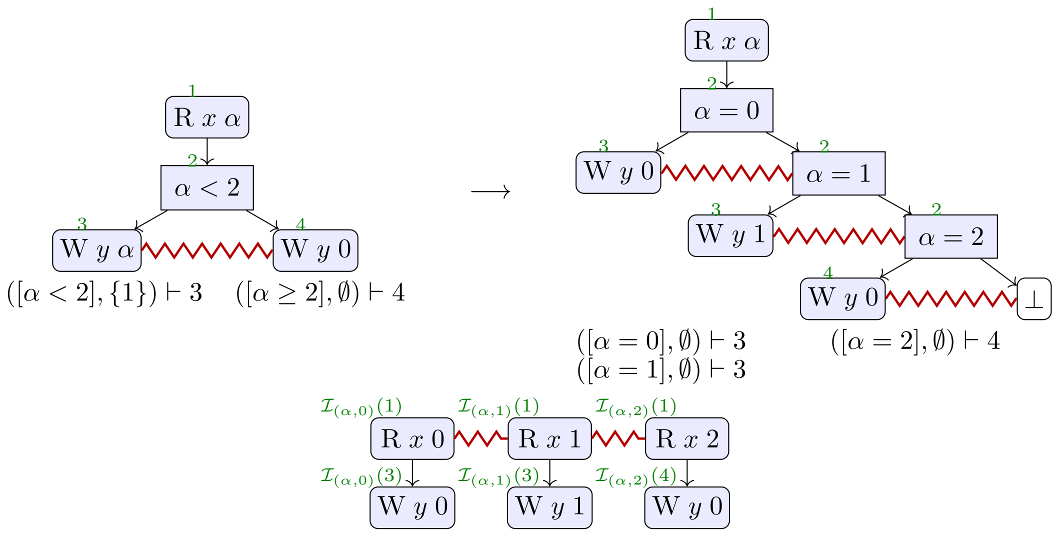

1
R $x$ $\alpha$
2
$\alpha < 2$
3
W $y$ $\alpha$
4
W $y$ 0
$([\alpha < 2], \{1\}) \vdash 3$
$([\alpha \geq 2], \emptyset) \vdash 4$
$\longrightarrow$
1
R $x$ $\alpha$
2
$\alpha = 0$
3
W $y$ 0
2
$\alpha = 1$
3
W $y$ 1
2
$\alpha = 2$
4
W $y$ 0
$\perp$
$([\alpha = 0], \emptyset) \vdash 3$
$([\alpha = 1], \emptyset) \vdash 3$
$([\alpha = 2], \emptyset) \vdash 4$
$\mathcal{I}_{(\alpha,0)}(1)$
R $x$ 0
$\mathcal{I}_{(\alpha,1)}(1)$
R $x$ 1
$\mathcal{I}_{(\alpha,2)}(1)$
R $x$ 2
$\mathcal{I}_{(\alpha,0)}(3)$
W $y$ 0
$\mathcal{I}_{(\alpha,1)}(3)$
W $y$ 1
$\mathcal{I}_{(\alpha,2)}(4)$
W $y$ 0

Figure 59: Value-based expansion for a symbolic program and an equivalent concrete structure.

We index the resulting control dependencies by value mappings, such that $P_V$ is the predicate which only holds under value mapping $V$. We know that an initial set $C = \mathcal{I}_V(O(P_V)) = \{(r : \text{R } x\ v) \mid r \sqsubseteq w\}$ exists due to the manner in which we initially constructed $P_V$, including a value for each read whose instantiation is $\sqsubseteq$-before $w$, and we also know that initially $D_V = \emptyset$ as all write labels have been simplified to constants by $P_V$.

We now need to show that all of our forwarding operations will preserve this property. We start by showing that our forwarding relations are equivalent:

$$\mathcal{I}_V(e) \xrightarrow{F} \mathcal{I}_V(r) \Leftrightarrow e \xrightarrow{F}^{\psi} r \wedge \psi \text{ is consistent with } V$$

Which we split into two separate implications.

In the left direction, we know that $\mathcal{I}_V(e) \xrightarrow{F} \mathcal{I}_V(r)$ and want to establish that $e \xrightarrow{F}^{r}$. If $\mathcal{I}_V(e) \xrightarrow{F} \mathcal{I}_V(r)$ then we know they have some labels $(\mathcal{I}_V(e) : \mathrm{R}\ x\ v)$ or $(\mathcal{I}_V(e) : \mathrm{W}\ x\ v)$, which we shorten to $(e : \_\ x\ \alpha)$, and $(\mathcal{I}_V(r) : \mathrm{R}\ x\ v)$, and that $e \leq r$ with no intervening access. Giving their symbolic equivalents the labels $(e : \_\ x\ \alpha)$ and $(r : \mathrm{R}\ x\ \beta)$, we know that $e \xrightarrow{F}^{\alpha=\beta} r$ by the established $e \leq r$ property. If value assignment $V$ contradicted $\alpha = \beta$, it would be impossible for $V(\alpha) = v$ and $V(\beta) = v$ to hold, therefore it must be consistent.

In the right direction, we know that $(e : \_\ x\ \alpha) \xrightarrow{F}^{\alpha=\beta} (r : \mathrm{R}\ x\ \beta)$ and that $V$ assigns the same values to $\alpha$ and $\beta$. As before, it must be the case that $\mathcal{I}_V(e) \leq \mathcal{I}_V(r)$, and that applying $\mathcal{I}_V$ to these labels results in some $(\mathcal{I}_V(e) : \_\ x\ v)$ and $(\mathcal{I}_V(r) : \mathrm{R}\ x\ v)$, thus $\mathcal{I}_V(r') \xrightarrow{F} \mathcal{I}_V(r)$.

Finally, we want to show that forwarding these equivalent initial justifications can only ever create likewise equivalent new justifications. For load forwarding, this is simple.

$$\forall C \vdash \mathcal{I}_V(w), (P, D) \vdash^{\psi} w.$$
$$C = \mathcal{I}_V(O(P) \cup D) \Leftrightarrow C \setminus \{\mathcal{I}_V(r : \mathrm{R}\ x\ \alpha)\} \cup \{\mathcal{I}_V(r' : \mathrm{R}\ x\ \beta)\} = \mathcal{I}_V(O(P_{[\alpha \backslash \beta]}, D))$$

This follows from the definition of the $O$ function. Removing $\alpha$ from $P$ removes $O(\alpha)$ from $O(P)$ and introducing $\beta$ to $P$ introduces $O(\beta)$ to $O(P)$.

**Store Forwarding** For store forwarding, we proceed through two layers of induction. The outer layer takes us forwards through program order, and restricts how many store forwardings we might see.

We want to assert that if store forwarding $w' \xrightarrow{SF} r'$ applies to some $C \vdash w$ and $(P, D) \vdash^{\psi} w$, it cannot create any new unequal justifications. As store forwarding includes dependencies for $w'$ in the symbolic model, we need to find a $C'$ that includes a choice of dependency for $w'$. This is possible without extra information if there can be no store forwarding for any justification of $w'$, but otherwise we need to know that the inner store forwarding also preserves equivalence under instantiation.

We induct over the set of writes which are $\sqsubseteq$-before $w'$.

$$\begin{aligned}
&\text{If } \forall w'' \sqsubseteq w',\ C \vdash \mathcal{I}_V(w),\ (P,D) \vdash^{\psi} w.\\
&\quad (w'' : \mathrm{W}\ x\ \alpha) \xrightarrow{SF}^{\psi_w} (r : \mathrm{R}\ x\ \beta) \wedge r \in O(P) \cup D \Rightarrow\\
&\quad \forall (P_w, D_w) \vdash w''.\ \exists C' \vdash \mathcal{I}_V(w).\ C' = \mathcal{I}_V(O(P_{[\alpha\backslash\beta]} \wedge P_w), D \cup D_w)\\
&\text{Then } \forall C \vdash \mathcal{I}_V(w),\ (P,D) \vdash^{\psi} w.\\
&\quad (w' : \mathrm{W}\ x\ \alpha) \xrightarrow{SF}^{\psi_w} (r : \mathrm{R}\ x\ \beta) \wedge r \in O(P) \cup D \Rightarrow\\
&\quad \forall (P_w, D_w) \vdash w'.\ \exists C' \vdash \mathcal{I}_V(w).\ C' = \mathcal{I}_V(O(P_{[\alpha\backslash\beta]} \wedge P_w), D \cup D_w)
\end{aligned}$$

We handle the base case in the greatest detail. In the proceeding, we are allowed to assume that no $w''$ exists before $w'$.

The inner induction is over the set $\{e \mid e \sqsubseteq w\}$, in reverse program order. We write as $\lfloor C_V \rfloor_e$ the subset of $C_V$ which is $\sqsubseteq$-before $e$, which is $\{e' \mid e' \in C_V \wedge e' \sqsubseteq e\}$, and as $\lfloor P_V \rfloor_e$ the sub-formula of $P_V$ which only refers to events originating from $\sqsubseteq$-before $e$.

If we begin at our initial justification $C_V$, we can think of all possible forwarding operations as a list of pending operations. The events which will be removed by these forwardings will be totally ordered by program order, so we can perform these forwardings by starting at the last event in $C_V$ and moving upwards. Any justification we gain by doing this can be expressed as the union $\lfloor C_V \rfloor_e \cup C$ for some $C$, and likewise any symbolic justification can be expressed as $(\lfloor P_V \rfloor_e \wedge P, D)$. If forwarding any equivalent pair of these justifications must result in equivalent justifications, then our forwarding rules are sound. The inner induction moves from point $e$ to point $e'$ by allowing a forwarding operation at $e$ – necessarily, all $C'$ generated by this operation will be in $\lfloor C_V \rfloor_{e'}$.

Our inner base case is simply that the untruncated $C_V = \mathcal{I}_V(O(P_V))$, which we established previously. Our inductive case requires us to push the truncation up one step, such that $\lfloor C_V \rfloor_e$ becomes $\lfloor C_V \rfloor_{e'}$ and we are allowed to use event $e$ for a forwarding operation.

$$\begin{aligned}
&\forall C, P, D.\\
&\qquad \text{If } \lfloor C_V \rfloor_{\mathcal{I}_V(w')} \cup C \vdash \mathcal{I}_V(w) \wedge (\lfloor P_V \rfloor_{w'} \wedge P, D) \vdash w \wedge (\lfloor C_V \rfloor_{\mathcal{I}_V(w')} \cup C) = \mathcal{I}_V(O(\lfloor P_V \rfloor_{w'} \wedge P) \cup D)\\
&\quad \text{Then } (w' : \mathrm{W}\ x\ \alpha) \xrightarrow{SF}^{\psi} (r : \mathrm{R}\ x\ \beta) \Rightarrow\\
&\qquad \forall (P_w, D_w) \vdash^{\psi_w} w'. \exists C' \vdash w.\ C' = \mathcal{I}_V(O(\lfloor P_V \rfloor_{w'} \wedge P_{[\alpha\backslash\beta]}), D \cup D_w)
\end{aligned}$$

We know that all $P_w$ are necessarily implied by $\lfloor P_V \rfloor_{w'}$ or contradict $\lfloor P_V \rfloor_{w'}$, as $P_w$ can only refer to variables already given a value in $\lfloor P_V \rfloor_{w'}$. Likewise any reads in $D_w$ will be the origins of variables in $\lfloor P_V \rfloor_{w'}$, thus unless $D_w$ contains a write we have $\lfloor C_V \rfloor_{\mathcal{I}_V w''} \cup C = \mathcal{I}_V(O(\lfloor P_V \rfloor_{w'} \wedge P_w \wedge P), D \cup D_w)$ and we have found a suitable $C'$. If $D_w$ does contain a write, it must be $\sqsubseteq$-before $w'$ and thus there must exist some $w'' \sqsubseteq w'$. As this is a contradiction, the base case ends here.

Proceeding in the outermost inductive case, The event $w''$ can only appear in a justification if there is some $r'$ such that $w'' \leq r' \sqsubseteq w$ and $w'' \xrightarrow{SF} r'$. We know that $r'$ must be in $O(\lfloor P_V \rfloor_{w'})$ by construction, and that $w'' \xrightarrow{LF} r' \Rightarrow \mathcal{I}_V(w'') \Rightarrow \mathcal{I}_V(r')$ as established above. This means we can invoke the inductive hypothesis on the unforwarded $\lfloor C_V \rfloor_{\mathcal{I}_V(w')} \cup C$ and $\lfloor P_V \rfloor_{w'} \wedge P$ justifications. gaining some $C' \vdash w$ such that, whatever our chosen $(P_w, D_w)$ pair, $C' = \mathcal{I}_V(O(\lfloor P_V \rfloor_{w'} \wedge P_w) \cup D \cup D_w)$, as required.

□

Having shown equivalence of the forwarding steps, we now show that every possible coproduct step preserves this equivalence.

**Lemma 11** (Coproduct). *For all $C$*

$$C \in \overset{\{e \mid e \sqsubseteq \mathcal{I}_V(w)\}}{\vdash \mathcal{I}_V(w)} \Leftrightarrow \exists (P, D) \vdash w.\ C = \mathcal{I}_V(O(P) \cup D)$$

*Proof.* To show correctness of coproduct, we proceed by induction over justification-up-to. We can construct a total order over reads in symbolic MRD by reassembling program order, such that if $e'_1 \sqsubseteq e'_2$ where $e'_1 \in \mathcal{I}(e_1)$ and $e'_2 \in \mathcal{I}(e_2)$, then $e_1 \sqsubseteq e_2$. When we add an upper bound to justification-up-to we remove candidate concrete justifications $C$, so we likewise need to remove specific $(P, D)$. Whenever $C \in \overset{S}{\vdash \mathcal{I}_V(w)}$, we know that any reads not in $S$ and not forwarded are guaranteed to be in $C$. We therefore remove all $(P, D)$ from the right hand side which do not contain references to these reads.

Abusing notation a little such that $r \sqsubseteq S$ means $\forall e \in S.\ r \sqsubseteq e$ and $V(\alpha)$ gives the value $v$ for which $(\alpha, v) \in V$, we structure the induction as follows. For all $C$ and $V$, and for all $S$ such that $r \bullet S$ is well-defined, assuming:

$$\begin{aligned} C \in \overset{S}{\vdash \mathcal{I}_V(w)} \iff & \exists (P, D) \vdash^{\psi} w.\ C = \mathcal{I}_V(O(P) \cup D)\ \wedge \\ & (\forall \mathcal{I}_V(r : \mathrm{R}\ x\ \alpha) \sqsubseteq S.\ P \wedge \psi \Rightarrow \alpha = V(\alpha)) \end{aligned}$$

we show that for all $C', V$:

$$C' \in \overset{\mathcal{I}_V(r) \bullet S}{\vdash \mathcal{I}_V(w)} \iff \exists (P', D')^{\psi'} \vdash w.\ C' = \mathcal{I}_V(O(P') \cup D') \wedge$$
$$(\forall (r' : \mathrm{R}\ x\ \alpha) \sqsubseteq r.\ P' \wedge \psi' \Rightarrow \alpha = V(\alpha))$$

In our ultimate goal there can no longer be any events $r$ which are $\sqsubseteq$-before $\{e \mid e \sqsubseteq \mathcal{I}_V(w)\}$, thus this structure shows the required property.

**Base Case** The base case of this induction requires us to show that

$$C \in \overset{\emptyset}{\vdash \mathcal{I}_V(w)} \iff \exists (P, D) \vdash w.\ C = \mathcal{I}_V(O(P) \cup D) \wedge$$
$$\forall (r : \mathrm{R}\ x\ \alpha) \sqsubseteq w. P \wedge \psi \Rightarrow \alpha = V(\alpha)$$

where $\overset{\emptyset}{\vdash \mathcal{I}_V(w)} = F(\mathcal{I}_V(w))$ by the definition of justification-up-to.

This holds by Lemma 10, and the fact that whenever $(r : \mathrm{R}\ x\ \alpha)$ is removed by forwarding from $P$, the assumption $\psi$ will contain some $\alpha = v$ for store forwarding or $\alpha = \beta$ for load forwarding where $\beta$ is given a value in $P$.

**Inductive Case** Unwrapping the definition of justification-up-to in our goal, we must show the following:

$$C' \in \overset{S}{\vdash \mathcal{I}_V(w)} \cup \sum_{\mathcal{I}_V(r)} \mathcal{I}_V(w) \iff \exists (P', D')^{\psi'} \vdash w.\ C' = \mathcal{I}_V(O(P') \cup D') \wedge$$
$$(\forall (r' : \mathrm{R}\ x\ \alpha) \sqsubseteq r.\ P' \wedge \psi' \Rightarrow \alpha = V(\alpha))$$

Splitting $A \Leftrightarrow B$ into cases $A \Rightarrow B$ and $B \Rightarrow A$, we now handle each direction separately.

**Concrete to Symbolic** If $C'$ is in $\overset{S}{\vdash \mathcal{I}_V(w)}$, we can simply invoke the inductive hypothesis. It remains to show that:

$$C' \in \sum_{\mathcal{I}_V(r)} \mathcal{I}_V(w) \Rightarrow \exists (P', D')^{\psi'} \vdash w.\ C' = \mathcal{I}_V(O(P') \cup D') \wedge$$
$$(\forall (r' : \mathrm{R}\ x\ \alpha) \sqsubseteq r.\ P' \wedge \psi' \Rightarrow \alpha = V(\alpha))$$

Declaring $\alpha$ to be the variable originating at read $r$, this means that we can remove the clause $\alpha = v$ from a control dependency for $w$ if we can remove $r$ from a justifying set for $\mathcal{I}_V(w)$.

From $C' \in \sum_{\mathcal{I}_V(r)} \mathcal{I}_V(w)$, we can assume the existence of some justifications $C_v$ and directly conflicting reads $r_v$ such that for each $v \in \mathcal{V}$ such that:

- $C_v \in \overset{S_v}{\vdash} w_v$
- $w_v \sim w$
- $C'_v \in \uparrow(C_v \setminus \{r_v\})$
- $C'_v \sim C'$

We know that one such $w_v$ must be $\mathcal{I}_V(w)$, and that others must be the instantiation of some other event under assignment $V_v$. As all $r_v$ are directly conflicting, they must all be instantiations of $r$ and the subsets of all $V_v$ handling events $\sqsubseteq$ before $r$ must be equal.

Our goal is to find some set $\mathbb{P}$ of predicates $P_v$ and $\mathbb{D}$ of event sets $D_v$, write $w$ and family of relabelling functions $\Lambda_v$, such that:

- For all $v$, some $(P_v, D_v) \vdash^{\psi_v} w_v$
- For all $v$, $\Lambda_v(w_v) = w$
- We can find $D$ such that all $D_v$ have some $D'_v \in \uparrow D_v$ such that $\Lambda_v(D'_v) = D$
- For the resulting dependency $P' = \Lambda^{-1}(\bigvee_{v\in\mathcal{V}} \Lambda_v(P_v))$ and $D' = \Lambda_v^{-1}D$, we have that $\mathcal{I}_V(O(P') \cup D') = C'$

From our inductive hypothesis, for each $C_v$ we can find some $(P_v, D_v) \vdash^{\psi_v} \mathcal{I}_{V_v}(w_v)$ such that $\mathcal{I}_{V_v}(C_v) = O(P_v) \cup D_v$.

All $\mathcal{I}_{V_v}(w_v)$ must be label isomorphic, and as all write labels are now concrete this gives us label similarity across all $w_v$.

We can construct each relabelling $\Lambda_v$ over reads such that, for all $r_{v1} \in C_{v1}$ and $r_{v2} \in C_{v2}$, if $r_1 \sim r_2$, $r_1 = \mathcal{I}_{V_{v1}}(O(\beta_1))$ and $r_2 = \mathcal{I}_{V_{v2}}(O(\beta_2))$ then $\Lambda_{v1}(\beta_1) = \beta_2$. For writes, if $\mathcal{I}_{V_{v1}}(w_{v1}) \in C_v$ then $(w_{v1} : \mathrm{W}\ x\ \beta_1) \in D_{v1}$. We know that some forwarding operation must have caused some $(w_{v2} : \mathrm{W}\ x\ \beta_2) \in D_{v2}$, where potentially $v1 = v2$, and that if $\beta_2$ is not a constant then this same operation must have included $O(\beta_2)$ in $D_{v2}$. If $v1 \neq v2$ then the inclusion of $w_{v1}$ in $D_{v1}$ must have been the result of a forwarding, which must have added $O(\beta_1)$ to $D_{v1}$. In either case, if $\beta_1$ is not a constant then $O(\beta_1) \in D_{v1}$, and thus $\mathcal{I}_{V_{v1}}(O(\beta_1)) \in C_{v1}$ and we already have a relabelling for $\beta_1$ in $\Lambda_{v1}$ by constructing it over reads.

We know that these relabellings must be similarity-preserving because there are no symbolic locations, thus $\mathcal{I}_{V_{v1}}(r_{v1})$ has equivalent $\leq$ edges to $r_{v1}$ and if $\mathcal{I}_{V_{v1}}(r_{v1}) \sim \mathcal{I}_{V_{v2}}(r_{v2})$ then $r_{v1} \sim r_{v2}$.

If some $C_v$ is not immediately isomorphic to $C$, then there must be some $(r' : \mathrm{R}\ x\ v)$ or $(w' : \mathrm{W}\ x\ v)$ which is in one but not another. In the first case, we weaken $P_v$ to $P_v \wedge \beta = v$ for $r' = \mathcal{I}_{V_v}(O(\beta))$ – extending $\Lambda_v$ to continue to respect isomorphism between $D_v$ and $C'$ gives us that all $P_v$ can be weakened to $\Lambda_v^{-1}(P) \wedge \alpha = v$. We also know that, because $r \sqsubseteq S$, for all $v$ either $P_v \Rightarrow \alpha = V_v(\alpha)$ or $\psi_v \Rightarrow \alpha = V_v(\alpha)$. Weakening all $P_v$ in the second case once more into $P_v \wedge \alpha = V_v(\alpha)$, this means that all $P_v$ can be weakened into some $P'_v \wedge \alpha = v$ such that there is a single $P'$ for which $\Lambda_v(P'_v) = P'$ holds for all $v$.

If $C_v$ is missing some read $\mathcal{I}_{V_v}(w')$ which can be added by upwards closure, we must be able to include $w'$ in $D$ as $\mathcal{I}_{V_v}(w') \sqsubseteq w$ means that $w' \sqsubseteq w$, therefore it cannot conflict with anything else in the justification. These two combine to give us that for all $v$, $C'_v \cup \{r_v\} = \mathcal{I}_V(O(\Lambda_v^{-1}(P') \wedge \alpha = v) \cup \Lambda_v^{-1} D)$.

Finally, with all $D_v$ isomorphic, coproduct allows us to transform these justifications into $(\Lambda_v^{-1} P, \Lambda_v^{-1} D) \vdash^{\psi_v} w_v$. As this operation does not remove any variables other than $\alpha$, we also preserve the property that variables introduced before $r$ will be in $P$ or $\psi_v$.

### Symbolic to Concrete

$$\exists (P', D')^{\psi'} \vdash w.\ C' = \mathcal{I}_V(O(P') \cup D')\ \wedge (\forall (r' : \mathrm{R}\ x\ \alpha) \sqsubseteq r.\ P' \wedge \psi' \Rightarrow \alpha = V(\alpha)) \Rightarrow$$
$$C' \in \overset{S}{\vdash} \mathcal{I}_V(w) \quad \cup \sum_{\mathcal{I}_V(r)} \mathcal{I}_V(w)$$

If $P' \wedge \psi \Rightarrow \alpha = V(\alpha)$, then we have found some $(P, D) \vdash^{\psi} w$ such that $\forall \mathcal{I}_V(r' : \mathrm{R}\ x\ \alpha) \sqsubseteq S.\ P' \Rightarrow \alpha = V(\alpha) \vee \psi \Rightarrow \alpha = V(\alpha)$, and by the inductive hypothesis $C'$ must be in $\overset{S}{\vdash} \mathcal{I}_V(w)$ .

Otherwise, there must be some set of labelled writes $w_v$, which we index by $v$, such that $(P' \wedge \alpha = v, D_v) \vdash^{\psi_v} (w_v : \mathrm{W}\ x\ n)$. We know that this must contain one such $w_v$ for each possible $v$ under our value range restriction for the variable $\alpha$ to be fully removed from the control dependency, otherwise it would contain some clause $\alpha \neq v$, and that there must exist some $D$ such that $\lambda_v^{-1} D \in {\uparrow} D_v$ for all $D_v$.

Our inductive hypothesis asserts that for all $w_v$, for some value assignment $V'$ we must be able to find justification $C_v \in \overset{S}{\vdash} \mathcal{I}'_V(w_v)$ such that $C_v = \mathcal{I}_{V'}(O(\Lambda_v^{-1}(P') \wedge \alpha = v) \cup D')$. We now want to use the coproduct mechanism to convert this set of $C_v$ edges into $C'$.

Expanding the definition of coproduct, we must establish the following:

$$
\begin{aligned}
&\text{let } \#^1 = \# \setminus (\#; \sqsubseteq) \\
&\text{let } R = \{r_v \mid r_v \#^1 r\} \\
&\text{let } \mathbb{S} = \{S \mid \forall r_v \in R.\ \exists! w_v.\ w_v \sim w \wedge r_v \sqsubseteq w_v \wedge (r_v, w_v) \in S\} \\
&\text{given } C \vdash w \text{ and } D \in \uparrow(C \setminus \{r\}) \\
&\text{then } \sum_r w \triangleq \bigcup_{S \in \mathbb{S}} \{D \mid \forall (r_v, w_v) \in S.\ \exists C_v \vdash w_v,\ D_v \in \uparrow (C_v \setminus \{r_v\}).\ D_v \sim D\}
\end{aligned}
$$

Because these predicates differ only in $\alpha$, as established earlier, all $C_v$ must be of the form $C'_v \cup (r_v : \mathrm{R}\ x\ v) \cup \mathcal{I}_V(D_v)$. By the label similarity of all $D_v$ to $D$ established above, we must be able to use $C'_v \cup \mathcal{I}_{V'}(D)$ as the upwards closure of $C_v$, thus finishing the coproduct operation. Finally, by construction, $C' = \mathcal{I}_V(O(P_v) \cup D)$.

$\square$

**Equivalence of Structures**

**Theorem 3** (Symbolic to Concrete MRD)**.** *Given a structure $(E, <, \sqsubseteq, \#, \curlyvee) = [\![P]\!]_{n\ 0\ \emptyset\ \top}$ with execution set $S$ and program guarantee $\forall v.\ v \in \mathcal{V}$, and the concrete MRD structure $(E', \sqsubseteq', \#', \lambda')$ with execution set $S'$ generated by $[\![P]\!]_{n\ 0\ \emptyset}$ with the value range $\mathcal{V}$, there exists an instantiation function $\mathcal{I}$ from events in $E$ to sets of events in $E'$ such that for all executions $(E_X, \mathrm{RF}_X, \mathrm{DP}_X, \leq_X, \psi_X)$ in $S$, there exists a set $\mathbb{X}$ of executions of the form $X' = (E_{X'}, \mathrm{RF}_{X'}, \mathrm{DP}_{X'}, \emptyset)$ in $S'$ where, for all such $X'$:*

- *$\mathcal{I}$ is total over reads and writes*
- *$E'_X \in \mathcal{I}(E_X)$*
- *For all events $(e : \mathrm{R}\ x\ \alpha) \in E_X$, if $(e' : \mathrm{R}\ x\ v) \in \mathcal{I}(e)$ then $\psi_X \wedge \alpha = v$ is satisfiable.*
- *For all events $(e : W\ x\ \alpha) \in E_X$, if $(e' : W\ x\ v) \in \mathcal{I}(e)$ then $\psi_X \wedge \alpha = v$ is satisfiable.*
- *For all edges $(e_1, e_2) \in \mathrm{RF}_X$, we have $(\mathcal{I}(e_1), \mathcal{I}(e_2)) \in \mathrm{RF}_{X'}$*
- *For all edges $(e_1, e_2) \in \mathrm{DP}_X$, we have $(\mathcal{I}(e_1), \mathcal{I}(e_2)) \in \mathrm{DP}_{X'}$*

*Proof.* We begin by disambiguating program order in concrete MRD from branch order in symbolic MRD. Program order is instead written $\sqsubseteq_p$ and branch order $\sqsubseteq_b$, to avoid ambiguity.

**Events** Now taking $\mathbb{X}$ to be the set of executions $(E', \textsc{rf}', \textsc{dp}', \emptyset)$ (noting that the justification relation will be empty for a complete program) where $E' \in \mathcal{I}(E_X), \textsc{rf}' \in \mathcal{I}(\textsc{rf}_X), \textsc{dp}' \in \mathcal{I}(\textsc{dp}_X)$, and all labelled events in $E$ satisfy $\psi$, this set must be non-empty by the above requirement that $\mathcal{I}(e)$ is never empty and the fact that all executions with unsatisfiable $\psi$ are discarded. It remains to show that all executions in $\mathbb{X}$ are in $[\![P]\!]_{n\ 0\ \emptyset}$.

**Reads-from and Preserved Program Order** As there are no symbolic locations, every $\leq_X$ relation is a subset of $\leq$. Because the location of $e \in E_X$ must be the same as the location of all $\mathcal{I}(e)$, we know that $\mathcal{I}(\leq_X) = \leq|_{E_{X'}}$.

Because the requirements for an $\textsc{rf}$ edge to be added to $X' \in \mathbb{X}$ are recorded in $\psi$, all executions with an event set in $\mathcal{I}(E_X)$ satisfying $\psi$ must be capable of adding the required edge, though some may have made different valid choices of $\textsc{rf}'$ which satisfy the same constraints. Since MRD generates all possible $\textsc{rf}$ shapes, there must be some set of executions with an event set in $\mathcal{I}(E_X)$ whose $\textsc{rf}$ relation is likewise in $\mathcal{I}(\textsc{rf}_X)$.

**Dependency** It remains to show equivalence between $\textsc{dp}$ edges. This follows from lemmata 10 and 11. $\square$

# Chapter 7

# Pointers

Removing the requirement for global value range analysis from the MRD model enables us to handle a type of operation that other memory models cannot: dynamic memory allocation. Pointers can be represented as global locations whose contents are not of *value* type but of *location* type. This type can contain things like liveness and bounds information, can be generated by memory allocation operations, and can be deleted or poisoned by deallocation.

Until now, we have treated values as abstract entities resembling natural numbers, and locations as elements of a finite set whose contents are known before interpreting the program. Dynamically allocated pointers, however, cannot be represented using this finite set representation. There is no way to know in advance how many locations will be allocated in an execution, how many will be freed, and how many will be reused. Instead, we replace the finite set representation of locations, which resembles the concrete MRD representation of values, with a combination of static locations (still treated as in concrete MRD) and abstract locations.

The semantics of a pointer, however, are nontrivial, and there is no single standardised model to use. We discuss three potential semantics in Sec. 7.2: a plain integer model in which a location is described entirely by an integer address, the block and offset model used by the CompCert verified compiler (Leroy et al. (2016)), and the provenance-based VIP model of Lepigre et al. (2022). In each subsection we show how the same general approach can be used to translate memory layout models into constraints on events.

We begin, however, without picking any semantics in particular. We first establish the foundations that the model will use to represent programs with pointers, including the general structure of allocation, deallocation, and undefined behaviour.

## 7.1 Uninterpreted Symbolic Locations

We start by making four assertions about how we want pointers to behave in the model, based on the behaviour of pointers in the C language standard:

1. Pointers are created by a `malloc` operation writing an unknown value of the location type to the pointer, which has both a *position* and a *bounds*.
2. Abstract locations are made invalid by a `free` operation.
3. Dereferencing an abstract location which does not correspond to an allocated region, or corresponds to an allocated region which has previously been freed, is undefined behaviour.
4. There is an externally enforced total order across `malloc` and `free` operations.

This leaves us with three new types of atomic action on global memory to represent as events: allocating, freeing, and undefined behaviour. We first lightly modify the base symbolic MRD semantics to allow for symbolic locations to be represented and treated appropriately by the dependency calculation, and then add to the language and interpretation function the syntactic constructs required for allocation and dereferencing.

### 7.1.1 Retrofitting Existing Events

To represent the ordering promises made by the implementation of `malloc` and `free`, each execution gains an additional relation: the *external coherence order* relation $\xrightarrow{xco}$. This is used very similarly to a modification order relation, but over all allocation and deallocation events. Each execution must have an $\xrightarrow{xco}$ relation that is total over all $\textsc{Alloc}\ l_1$ and $\textsc{Free}\ l_2$ events, and $\xrightarrow{xco}$ must be consistent with program order. We check this by placing all allocation and free events in maximal program order with all other allocation and free events when they are appended. Different executions may make different choices of $\xrightarrow{xco}$ edges, representing different execution orders of these events. We include $\xrightarrow{xco}$ in both the completeness and coherence requirements.

**Definition 27** (Complete Symbolic Execution)**.** *An execution $X$ is complete if and only if:*

1. *For all $(r : R\ x\ \alpha)$ in $X$ there exists some $(w : W\ x\ \beta)$ such that $w \xrightarrow{RF}_X r$*
2. $\xrightarrow{xco}$ ***is total over all*** $\textsc{Alloc}\ l$ ***and*** $\textsc{Free}\ l$ ***events in*** $X$.

The addition of symbolic locations requires small modifications to condition generation. Whenever an operation requires the equality or inequality of a pair of locations and one or both are symbolic, this equality is recorded in an execution condition. This impacts the RF and $\leq$ condition checking in the definition of coherence. The maximal preserved program order now places *all* accesses to symbolic locations in $<$-order with all accesses of any kind in the same thread, and the execution condition must specify whether or not the locations are equivalent.

**Definition 28** (Coherent Symbolic Execution with Dynamic Memory)**.** *An execution X is coherent if and only if:*

1. $(\leq \cup DP_X \cup RF_X)^*$ *is acyclic.*
2. $(\xrightarrow{\mathbf{xco}} \cup <)$ ***is acyclic.***
3. *Whenever* $(w : \mathrm{W}\, l_1\ \alpha) \xrightarrow{RF}_X (r : \mathrm{R}\, l_2\ \beta)$*:*
   (a) $(\psi_X \Rightarrow \alpha = \beta)$ *and*
   (b) *for all other events* $e \in E$ *, either* $\psi_X \Rightarrow val(e) \neq \alpha$ *or* $\neg(w \leq_X e \leq_X r)$*,* ***, and***
   (c) $\psi_{\mathbf{X}} \Rightarrow \mathbf{l_1} = \mathbf{l_2}$
4. ***For all events*** $\mathbf{e_1}$ ***and*** $\mathbf{e_2}$***, if*** $\mathbf{e_1} < \mathbf{w_2}$ ***and neither event is an allocation then either***
   (a) $\mathbf{e_1} \leq \mathbf{e_2}$ ***and*** $(\psi_{\mathbf{X}} \Rightarrow \mathbf{loc(e1)} = \mathbf{loc(e2)})$***, or***
   (b) $(\psi_{\mathbf{X}} \Rightarrow \mathbf{loc(e1)} \neq \mathbf{loc(e2)})$
5. *For all writes* $(w : \mathrm{W}\ x\ \alpha) \in E_X$*, there exists a dependency triple* $(DP_w, (\mathrm{W}\ x\ \alpha), \varphi_w) \in freeze(w)$ *such that* $DP_w \subseteq DP_X$ *and* $\psi_X \Rightarrow \varphi_w$*.*
6. *For all events* $e \in E_X$ *, the conjunction* $\psi_X \wedge \Upsilon(e)$ *is satisfiable.*

While the updated forwarding rules expand the conditions generated by $\xrightarrow{LF}$ and $\xrightarrow{SF}$:

$$(r_1 : \mathrm{R}\ l_1\ \alpha) \xrightarrow{LF}^{\psi} (r_2 : \mathrm{R}\ l_2\ \beta) \text{ where } \psi \Rightarrow \nexists e.\ r_1 \leq e \leq r_2 \wedge r_1 \Longleftarrow r_2$$

$$(w : \mathrm{W}\ x\ v) \xrightarrow{SF}^{\psi} (r : \mathrm{R}\ x\ \alpha) \text{ where } \psi \Rightarrow \nexists e.\ w \leq e \leq r \wedge w \Longleftarrow r$$

### 7.1.2 Reading, Writing, and Dereferencing Pointers

We also need syntactic constructs for writing to, reading from, and dereferencing pointers. We start by introducing the *address-of* and *dereference operators* `&` and `*` respectively.

The address-of function `&` is treated as part of the language of expressions, and returns the uninterpreted symbolic value `&x` when applied to input $x$:

$$[\![\texttt{p := \&x}]\!]_{n\ \rho\ \kappa\ \varphi} = [\varphi](e : \mathrm{W}\ p\ \texttt{\&x}) \bullet \kappa(\rho)$$

To dereference a pointer, we first require that its value be copied into a local register. If $p$ points to address $\alpha_l$, then copying $p$ into register `rp` leads to the register environment $\texttt{rp} \mapsto \alpha_l$. Dereferencing this register then loads from or stores to the location $\alpha_l$.

$$[\![\texttt{*rp := e}]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq [\varphi](w : \mathrm{W}\ \rho(rp)\ [\![e]\!]) \bullet \kappa(\rho)$$

$$[\![\texttt{x := *rp}]\!]_{n\ \rho\ \kappa\ \varphi} \triangleq [\varphi](r : \mathrm{R}\ \rho(rp)\ \alpha) \bullet [\varphi](w : \mathrm{W}\ x\ \alpha) \bullet \kappa(\rho)$$

### 7.1.3 Allocation and Freeing

The allocation of a region in memory can be divided into either two or three separate events, depending on the implementation. It will first designate a region in memory to be referred to by the given pointer, making this region legal to access. It will then write the location of that region to the given pointer. Finally, and optionally, it may initialise its contents.

For our toy language, we require any memory allocations to store the designated address in a register. This means that we allow `malloc` calls only on the right-hand side of an assignment operator, with a register on the left. We typically use the register `rp` as a pointer register in our tests and examples.

First, we introduce to the language the construct `malloc`, with the following valid syntax:

```
rp := malloc s
```

Where $s$ indicates the size of a region and `rp` is a register whose contents are of the pointer type. To avoid discussions of alignment and mixed size accesses due to their complexity on existing hardware Flur et al. (2017), these are treated as an integer multiple of the size of `typeof(*p)` and all pointer offsets are assumed to be well-aligned. More granular representations are considered future work. We also assume every `malloc` will succeed, for the sake of simplicity, though it is a small addition to have it return a possibly-null value. Both allocation and freeing are passed registers, whose contents can be written to and read from global locations in the same manner as other values.

Interpreting these statements creates an Alloc $l$ event, where $l$ is some representation of a location and size:

$$[\![\texttt{rp := malloc s}]\!]_{n\ \rho\ \kappa\ \varphi} = (e : \textsc{Alloc}\ l)[\varphi] \bullet \kappa(\rho)$$

Allocation events may appear in control and data dependencies as the origin of the symbol $l$, in the same manner as read events, and they appear in $<$-order with all accesses to a symbolic location and all other allocation and free events.

$$(e : \textsc{Alloc}\ l)[\varphi] \bullet (E, <, \sqsubseteq, \#, \Upsilon) \triangleq (\{e\} \cup E, < \cup <', (\sqsubseteq \cup \sqsubseteq')^*, \#, \Upsilon[e \mapsto \varphi])$$

Where:

$$\begin{aligned} <' &= \{(e, e') \mid (e' : \mathrm{R}\ \alpha\ v) \in E \vee (e' : \mathrm{W}\ \alpha\ v) \in E\} \\ &\quad \cup \{(e, e') \mid (e' : \textsc{Alloc}\ l') \in E \vee (e' : \textsc{Free}\ l') \in E \\ \sqsubseteq' &= \{(e, b) \mid (b : [\varphi_b]) \in E\} \end{aligned}$$

We must also include the possibility of value-to-location data dependencies involving allocation events. If we read a location into a register and then dereference the register to write to that location, we have introduced a dependency between the read value and the write location. To that end, we modify the initial pre-justifications for writes.

The new initial pre-justification for $(w : \mathrm{W}\ l\ e)$ is $(\Upsilon(w), D) \vdash (w : \mathrm{W}\ x\ e)$, where $D = \{(r : \mathrm{R}\ x\ \alpha) \mid \alpha \in vars(e) \vee vars(l)\} \cup \{(a : \textsc{Alloc}\ \alpha) \mid \alpha \in vars(l)\}$.

Freeing a dynamically allocated region is done by passing a pointer to the start of the region to the `free` operator, which then creates a Free event:

$$[\![\texttt{free rp}]\!]_{n\ \rho\ \kappa\ \varphi} = (e : \textsc{Free}\ \rho(rp))[\varphi] \bullet \kappa(\rho)$$

Freeing a region requires only a single event, as it only marks a region as once again illegal to access and does not overwrite the location stored in any pointer which points to this region, nor the contents of the region itself. The free event itself appears in $<$ in the same manner as allocation events.

$$(e : \textsc{Free}\ l)[\varphi] \bullet (E, <, \sqsubseteq, \#, \Upsilon) \triangleq (\{e\} \cup E, < \cup <', (\sqsubseteq \cup \sqsubseteq')^*, \#, \Upsilon[e \mapsto \varphi])$$

Where:

$$\begin{aligned}
<' &= \{(e, e') \mid (e' : \text{R } \alpha\ v) \in E \vee (e' : \text{W } \alpha\ v) \in E\} \\
&\quad \cup \{(e, e') \mid (e' : \textsc{Alloc } l') \in E \vee (e' : \textsc{Free } l') \in E \\
\sqsubseteq' &= \{(e, b) \mid (b : [\varphi_b]) \in E\}
\end{aligned}$$

Accessing a region after freeing it introduces a new complexity to the model: undefined behaviour.

### 7.1.4 Undefined Behaviour

In the C standard, the execution of certain statements in certain environments may lead to undefined behaviour. The total set of causes of undefined behaviour is largely beyond the scope of what MRD represents, particularly when forbidding mixed-size or misaligned accesses, but we are forced to handle division by zero and use after free statements. The presence of undefined behaviour at any point during execution causes the entire program to have undefined behaviour, and there are no limits on what that behaviour may be. Here we represent undefined behaviour via the special `UNDEF` label. If any complete execution contains an `UNDEF` event, the entire program is rejected by the semantics.

The `UNDEF` label is never assigned to a node during structure generation, but may instead be created by various conditions in the execution in which it appears. We represent `UNDEF`-relabelling as an arrow annotated with a condition, where undefined behaviour can occur whenever the condition holds.

$$(e : l) \xrightarrow{P(X)} (e : \texttt{UNDEF})$$

If any execution $X$ containing $e$ satisfies property $P$, then $e$ takes the label `UNDEF` and, if $X$ is complete and coherent by the time we finish interpreting the program, the entire program has undefined behaviour.

We also introduce the label $\tau$ for any internal step which does not generate an event, such as computing an expression and placing its value in a register, as while it does not touch memory, it may result in undefined behaviour. Whenever an expression $e$ is evaluated using local values, a $\tau : e$ event appears. The $\tau$ events do not appear in any ordering edges, including dependencies, so their inclusion does not impact the behaviour of the model. However, whenever a $\tau : e$ event appears in an execution $X$ in which $\psi_X$ implies that $e$ may be undefined, its label is changed to `UNDEF`.

For instance:

$$(e : \tau : 1 \div \alpha) \xrightarrow{\alpha=0} (e : \texttt{UNDEF})$$

```
x = 0;
r1 = x;
r2 = 1/r2;
```

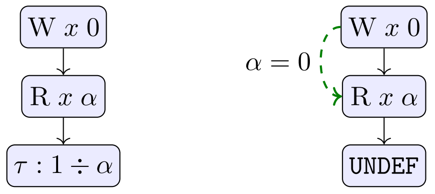


Figure 60: An internal event being relabelled to `UNDEF` after the addition of an RF edge.

We do not abandon evaluation as soon as an `UNDEF` node appears in a partial execution, as this execution may be unobservable due to its execution condition or discarded due to a DP $\cup$ RF $\cup \leq$ cycle. This is illustrated by a program in which an OOTA execution could perform a division by 0.

```
           x := 1;
           y := 1;
r1 = x;           r2 = y;
y = r1;     ||    x := r2;
r1 = 1/r1;
```

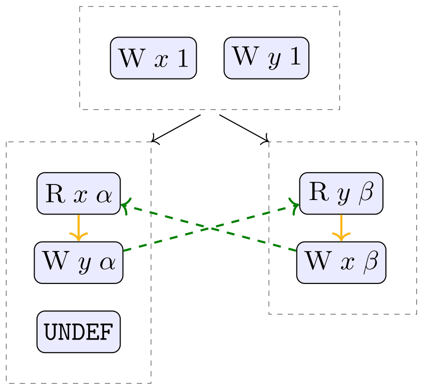


Figure 61: The program OOTA-UB, which cannot in practice observe undefined behaviour at the division operation despite the partial execution in which $\alpha$ may be 0.

We define the relabelling step as a relation across labelled events, where $(e : l) \xrightarrow{P} (e : \texttt{UNDEF})$ indicates that in executions where the contained events satisfy $P$ and the condition does not prohibit $P$, the event $e$ may be undefined. An execution $E$ is *defined* iff, for all $e \in E$ where $(e : l) \xrightarrow{P_e} (e : \texttt{UNDEF})$, the execution condition and relations guarantee $\neg P_e$.

We now give the undefined edges for the events we represent thus far. We assert that a FREE can only happen after an ALLOC , that an access to $\alpha$ must occur after it has been allocated and before it has been freed, and that a read from $\alpha$ must follow an initialising write to it.

We introduced *relaxed happens-before*, $\xrightarrow{rhb}$, as a shorthand for a union of global and execution-local orderings.

$$\xrightarrow{rhb}_X = (\leq_X \cup \mathrm{DP}_X \cup \mathrm{RF}_X \cup \xrightarrow{xco}_X)^*$$

If $e_1 \xrightarrow{rhb}_X e_2$, then if we are observing execution $X$, event $e_1$ will always be visible before event $e_2$.

$$(e : \tau : expr \div v) \xrightarrow[v=0]{} (e : \texttt{UNDEF})$$

$$(e : \textsc{Free}\ l) \xrightarrow{\neg((a:\textsc{Alloc}\ l\ \ s)\xrightarrow{rhb} e)} (e : \texttt{UNDEF})$$

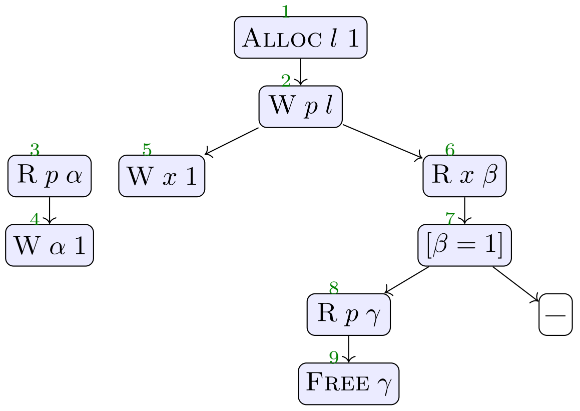
$$(e : \mathrm{R}\ l\ \alpha) \xrightarrow{\neg((a:\textsc{Alloc}\ l\ \ s)\xrightarrow{rhb} e \wedge l \leq \alpha < l+s)} (e : \texttt{UNDEF})$$

$$(e : \mathrm{W}\ l\ \alpha) \xrightarrow{\neg((a:\textsc{Alloc}\ l\ \ s)\xrightarrow{rhb} e \wedge l \leq \alpha < l+s)} (e : \texttt{UNDEF})$$

### 7.1.5 Uninterpreted Location Test Cases

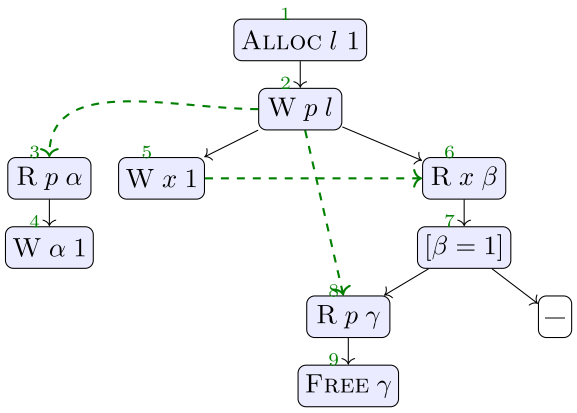
#### Free Race

```
        p := malloc 1;
rp := p;       r2 := x;
*rp := 1;  ||  if (r2 = 1) {
x := 1;            free p;
               }
```



A sequentially consistent semantics will always allow this program to execute, but in a weak memory context it may lead to undefined behaviour. As the writes to $x$ and $*rp$ are unrelated, they made be reordered and thus the pointer may be used after being freed.

We can construct an execution in which there is no $\mathrm{DP} \cup \mathrm{RF} \cup \xrightarrow{xco}$ path from the $\textsc{Free}\ \gamma$ event to the $\mathrm{R}\ p\ \alpha$ event, $\alpha$ and $\gamma$ must be the same variable due to the reads-from edges $2 \xrightarrow{\mathrm{RF}} 3$ and $2 \xrightarrow{\mathrm{RF}} 8$.



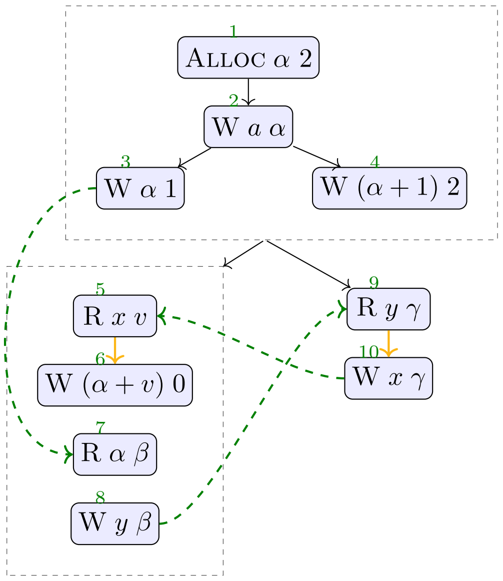
This causes both the read event 6 and subsequent write event 7 to be given the `UNDEF` label in this execution, and as there is no cycle to disqualify it and a full set of RF edges to complete it, this must be an execution of the program. The behaviour of the program is therefore undefined.

It could, however, be given defined behaviour by a semantics that placed every ALLOC and FREE event in preserved program order with all events program order before or after them, effectively placing fences both before and after each `malloc` or `free` call.

**JCTC 12**

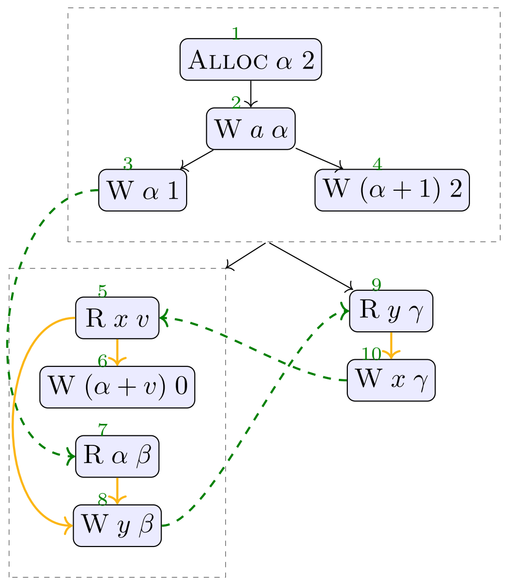
```
          a := malloc 1;
          *a := 1;
          *(a + 1) := 2;
r1 := x;
*(a + r1) := 0;         r3 := y;
r2 := *a           ||   x := r3;
y := r2;
```

Forbidden outcome: $\texttt{r1} = \texttt{r2} = \texttt{r3} = 1$.



In this program, adapted from the Java Causality Test Cases[1], there should be some ordering in the left-hand thread which prevents a value of 1 being copied from $*a$ into $r_2$ before the value of $x$ has been read. We begin by drawing the reads-from edges needed to observe the outcome and the data dependency edges we cannot lift.

The remaining non-trivial dependency is for event 8. We begin with $(\top, \{7\}) \vdash 8$. As per our forwarding rules, we may observe $6 \xrightarrow{F}^{v=0} 7$. This forces us to add a choice of dependency for event 6 to every justification of event 8, resulting in the edges $(\top, \{5, 6\}) \vdash^{v=0} 8$ and $(\top, \{5, 7\}) \vdash^{v \neq 0} 8$. In the execution we are interested in, the condition contains $[\beta = \gamma = v = 1]$, forcing us to accept the latter justification. After freezing, this leads to a $5 \xrightarrow{\text{DP}} 8 \xrightarrow{\text{RF}} 9 \xrightarrow{\text{DP}} 10 \xrightarrow{\text{RF}} 5$ cycle in every potential execution which observes the forbidden outcome, removing them from the permitted outcomes of the model.



## 7.2 Choices of Location Semantics

The most naive option for a pointer semantics is to treat pointers as interchangeable with longs. Under this semantics, any location is a 64-bit unsigned integer and a dereference is permitted whenever that number is associated with an allocated region of memory. The main objections to this model, noted by Memarian et al. (2019), claim that it is overly permissive of programming patterns which are rarely used in practice at the expense of valuable and sometimes widely employed optimisations on pointers. A compiler may, for instance, decide that an equality check

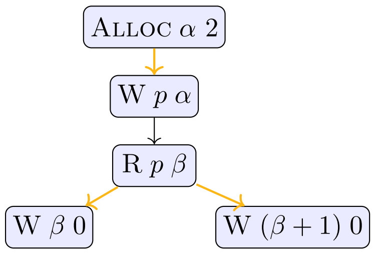
[1]The original test case uses a statically allocated array, and a set of global variables corresponding to array cells suffice to show correctness for Java, but the behaviour should stay the same for a dynamically allocated array.

between two pointers which were declared as references to different objects will always return false and optimise away the branch, regardless of whether or not the offsets given to the pointers in question would make them numerically equivalent.

### 7.2.1 Plain Integers

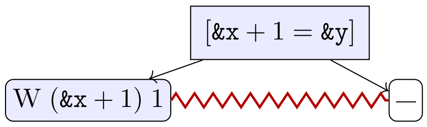
The most direct representation of a location is as simply a positive integer. This is closer both to the hardware representation and the majority of C programmers' intuitions about pointers than more abstract models (Memarian et al. (2016)), and therefore provides a meaningful semantics for the largest body of extant C code . The ALLOC $l$ $s$ event represents the allocation of a region starting at $l$ and terminating at $l + (s - 1)$, and the dereference of any location within that region is permitted. Any dereference is assumed to succeed, but a complete execution containing a dereference is only permitted if an RF edge can unify the unknown location with a known, legal location.

```
p := malloc 2;
rp := p;
*rp := 0;
*(rp + 1) := 0;
```



This model gives a semantics to some patterns that are forbidden by the C standard. One can, for instance, write to $y$ creating a pointer one-past $x$ and verifying that this is the address of $y$:

```
rp := &x + 1;
if (rp = &y) {
    *rp := 1;
}
```



There can only be one integer with the value `&y`, and if this integer is definitely equal to `&x` $+ 1$ then the dereference must result in a legal value.

**Address Disjointedness** When working with plain integers, we also need to make some disjointedness assertions about addresses. If a region starting at $l_2$ of size $s_2$ is allocated while region $l_1$ of size $s_1$ is still live, then the addresses $l_1..l_1 + s_1$ and $l_2..l_2 + s_2$ should be non-overlapping. Likewise, any addresses which correspond to statically allocated variables should be disjoint from all other statically or dynamically allocated addresses. We ensure the latter by establishing a set of statically allocated global variables $\mathcal{G}$, where the address of variable $x$ is the uninterpreted $\&x$, and assert that all values in this set are disjoint from symbolic values allocated by `malloc`. Both of these guarantees are stored in the execution conditions, and maintained when a `malloc` event is appended and when two executions are placed in parallel.

$$\begin{aligned}
&\forall (a : \textsc{Alloc}\ ls) \in E_X, \&x \in \mathcal{G} \\
&\quad \psi_X \Rightarrow (l + s \leq \&x) \vee \&x < l \\
&\forall (a_1 : \textsc{Alloc}\ l_1 s_1), (a_2 : \textsc{Alloc}\ l_2 s_2) \in E_X. \\
&\quad \psi_X \Rightarrow ((l_1 + s_1 \leq l_2) \vee (l_2 + s_2 \leq l_1))
\end{aligned}$$

### 7.2.2 The CompCert Model

The CompCert formally verified C compiler Leroy et al. (2016) is a fully-formalised and proven correct compiler for the C language, including dynamic memory allocation. To formalise the sections of the language handling pointers and address literals, CompCert has a formal memory model Leroy et al. (2012) which defines a memory state containing abstract *blocks*, where pointer contents are treated as pairs of blocks and offsets. This effectively forbids more complex forms of pointer manipulation, such as address comparison using non-equality relations and inter-object pointer arithmetic, simplifying the correctness proofs required and permitting various alias-based optimisations. We give a brief overview of version 1 of the model, as the updated version 2 includes mixed-size operations.

**The CompCert Memory Layout Model** The model describes four operations over memory locations.

- `alloc` $m\ l\ h$ allocates a new block to memory $m$ spanning addresses $l$ to $h$, returning the allocated block and the modified memory.
- `free` $m\ b$ frees block $b$ from memory $m$, returning the modified memory.
- `load` $m\ \tau\ b\ i$ reads a value of type $\tau$ from offset $i$ in block $b$, either succeeding and returning `Some` $v$ or, if out-of-bounds or misaligned, failing and returning `None`.
- `store` $m\ \tau\ b\ i\ v$ stores value $v$ of type $\tau$ into block $b$ at offset $i$, either succeeding and returning `Some` $m'$ as a new memory or, if out-of-bounds or misaligned, failing and returning `None`.

The types described by $\tau$ are 8 or 16-bit signed or unsigned integers, 32-bit integers, and 32 or 64-bit floats. The behaviour of these four operations are given primarily by the "good variable" laws:

1. If $(m', b) = \texttt{alloc}\ m\ l\ h$ and $b \neq b'$ then $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$
2. If $(m', b) = \texttt{alloc}\ m\ l\ h$ and $\texttt{load}\ m\ \tau\ b\ i = \texttt{Some}\ v$ then $v = \texttt{undef}$
3. If $m' = \texttt{free}\ b\ m$ and $b \neq b'$ then $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$
4. If $m' = \texttt{free}\ b\ m$ then $\texttt{load}\ m'\ \tau\ b\ i = \texttt{None}$
5. If $\texttt{Some}\ m' = \texttt{store}\ m\ \tau\ b\ i\ v$ and $b \neq b'$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$ then $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$
6. If $\texttt{Some}\ m' = \texttt{store}\ m\ \tau\ b\ i\ v$ and $|\tau| = |\tau'|$ then $\texttt{load}\ m'\ \tau'\ b\ i = \texttt{Some}(\texttt{convert}\ \tau'\ v)$
7. If $\texttt{Some}\ m' = \texttt{store}\ m\ \tau\ b\ i\ v$ and the above do not hold then $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{Some undef}$ or $\texttt{None}$

**Translating CompCert to Event Structures** Using the points above, we write our symbolic locations as block-offset pairs. Event $e_1$ is *strictly before* event $e_2$ in execution $X$ iff $(e_1, e_2) \in (\leq_X \cup \mathrm{RF}_X \cup \mathrm{DP}_X \cup \xrightarrow{xco}_X)^*$.

We then define symbolic locations and their accesses as follows:

- An allocation event is given the label $\textsc{Alloc}\ b\ s$ to indicate the creation of block $b$ of size $s$.
- A free event is given the label $\textsc{Free}\ b$ to indicate the deallocation of block $b$.
- $\textsc{Alloc}\ b\ s$ and $\textsc{Free}\ b$ are placed in $\leq$ order with any accesses to block $b$.
- A read from a symbolic location $(b, i)$ is permitted if there exists a write of some value $v$ to $(b, i)$ which is strictly before the read and any $\textsc{Free}\ b$ event occurs strictly after it.
- A write to a symbolic location $(b, i)$ is permitted if there exists a prior allocation event $\textsc{Alloc}\ b\ s$ where $i < s$ and any $\textsc{Free}\ b$ event occurs strictly after it.
- A free of a block $b$ is permitted if there exists an allocation $\textsc{Alloc}\ b\ s$ strictly before it.

All other loads and stores are given the `UNDEF` label.

We formalise these into the program semantics $[\![P]\!]^{\mathrm{CC}}_{n\ \rho\ \kappa\ \varphi}$ and expression semantics $[\![e]\!]^{\mathrm{CC}}$, adding conditions to ensure blocks allocated at different events are always disjoint and never

$$
\begin{aligned}
[\![\texttt{rp := malloc s}]\!]^{\text{CC}}_{n\ \rho\ \kappa\ \varphi} &\triangleq [\varphi](e : \textsc{Alloc}\ b\ [\![s]\!]_\rho) \bullet \kappa(\rho[rp \mapsto (b,0)]) \\
[\![\texttt{free(rp)}]\!]^{\text{CC}}_{n\ \rho\ \kappa\ \varphi} &\triangleq [\varphi](e : \textsc{Free}\ \pi_1 \bullet \rho(rp)) \bullet \kappa(\rho) \\
(e : \text{R}\ (b,n)\ v) &\xrightarrow{\neg((a:\textsc{Alloc}\ b'\ \ s)\xrightarrow{rhb} e \wedge b=b' \wedge n<s)\wedge(b,n)\notin\mathcal{G}} (e : \texttt{UNDEF}) \\
(e : \text{R}\ (b,n)\ v) &\xrightarrow{(f:\textsc{Free}\ b')\xrightarrow{rhb} e \wedge b=b'} (e : \texttt{UNDEF}) \\
(e : \text{R}\ (b,n)\ v) &\xrightarrow{\neg((w:\text{W}\ (b',s')\ v)\xrightarrow{rhb} e \wedge (b,s)=(b',s'))} (e : \texttt{UNDEF}) \\
(e : \text{W}\ (b,n)\ v) &\xrightarrow{\neg((a:\textsc{Alloc}\ b'\ \ s)\xrightarrow{rhb} e \wedge b=b' \wedge n<s)\wedge(b,n)\notin\mathcal{G}} (e : \texttt{UNDEF}) \\
(e : \text{W}\ (b,n)\ v) &\xrightarrow{(f:\textsc{Free}\ b')\xrightarrow{rhb} e \wedge b=b'} (e : \texttt{UNDEF}) \\
(e : \textsc{Free}\ b) &\xrightarrow{\neg((a:\textsc{Alloc}\ b'\ \ s)\xrightarrow{rhb} e \wedge b=b')\vee(b,0)\in\mathcal{G}} (e : \texttt{UNDEF}) \\
[\![\texttt{\&x}]\!]^{\text{CC}} &= (x,0) \\
[\![(b,n)+v]\!]^{\text{CC}} &= (b,n+v) \\
(E,S,\sqsubseteq,\psi) \in [\![P]\!]^{\text{CC}}_{n\ 0\ \emptyset\ \varphi} &\Rightarrow (E,S,\sqsubseteq,\psi,\Psi) \in (\![P]\!)^{\text{CC}}_{n\ 0\ \emptyset\ \varphi} \\
&\text{Where } \Psi = \forall\alpha \in \texttt{blocks}, \beta \in \texttt{blocks}\setminus\{\alpha\}.\ \alpha \neq \beta \\
&\text{and } \texttt{blocks} = \{\alpha \mid (a : \textsc{Alloc}\ \alpha\ s) \in E \vee (\alpha,0) \in \mathcal{G}\}
\end{aligned}
$$

Figure 62: The semantic interpretation and `UNDEF` relabellings for $[\![P]\!]^{\text{CC}}_{n\ \rho\ \kappa\ \varphi}$. The program guarantee $\Psi$ can be derived from all $\psi_X$ in all executions $X$ containing the same assertion.

alias with global variables and adding the above requirements for defined events, resulting in the rules shown in Fig. 62.

```
p := malloc 2;
rp := p;
*rp := 0;
*(rp + 1) := 0;
```

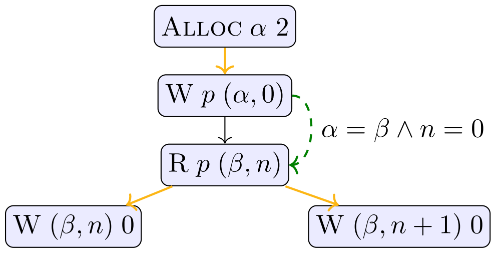


Figure 63: An event structure built using the CompCert-compatible pointer semantics.

**Theorem 4.** *For any program $P$, $[\![P]\!]^{CC}_{n\ 0\ \emptyset\ \top}$ is defined iff any trace of $P$ permitted by MRD contains only memory accesses permitted by the CompCert model.*

*Proof.* A trace $T$ of program $P$, represented as a totally ordered set of events, is permitted whenever there is some execution $E$ in $[\![P]\!]^{\text{CC}}_{n\ 0\ \emptyset\ \top}$ such that the ordering of two events in $T$ never contradicts their ordering in $E$. Any series of memory events in $T$ can be expressed as a chain of `alloc`, `free`, `load` and `store` actions in the CompCert semantics, generated by the evaluation of some single-threaded program whose only complete execution under MRD is the execution $E$. If, for instance, $T$ is an allocation, followed by a write, followed by a free, then we can construct the program text $P =$ `rp := malloc 1; *rp := 1; free(rp);` and then find the CompCert state resulting from the interpretation of $P$.

$$T = (1 : \textsc{Alloc}\ \alpha\ 2) \to (2 : \mathrm{W}\ (\alpha, 0)\ 1) \to (3 : \textsc{Free}\ \alpha)$$

$$P = \texttt{rp := malloc 1; *rp := 1; free(rp);}$$

$$CC(T) = \texttt{free}\ m\ \alpha \qquad \texttt{Some}\ \ m = \texttt{store}\ m'\ \texttt{Int}\ \alpha\ 0\ 1 \qquad (m', \alpha) = \texttt{alloc}\ m''\ l\ (l+2)$$

$$m'' = \texttt{empty}$$

We now show that whenever a read event $(r : \mathrm{R}\ (b, i)\ v)$ maintains its label and does not become undefined, the corresponding chain of CompCert memory actions must lead to case 6, meaning it must have the form `load` $m'\ \tau'\ b\ i$ where `Some` $m' =$ `store` $m\ \tau\ b\ i\ v$, while whenever a write event $(w : \mathrm{W}\ (b, i)\ v)$ maintains its label, the corresponding chain of CompCert memory actions must have the form `store` $m'\ \tau\ b\ i\ v$ where there is some block $b$ of size $s$ allocated in $m'$ and $i < s$.

Any accesses to disjoint locations can be ignored for the calculation of the memory state, due to laws 1, 3, and 5. We concern ourselves entirely with the subset of $T$ which accesses the same location as $e$, beginning with read events. We show that any case other than case 6 will result in the corresponding event taking the `UNDEF` label.

If we are in case 2, then the CompCert chain has the form `load` $m'\ \tau\ b\ i$ where $m' =$ `alloc` $m''\ l\ h$. In this case, some trace has no write event to location $\alpha$ before a read from $\alpha$, which means there cannot be such a read event $\xrightarrow{rhb}$-before the write. In this case, we relabel $e$ to `UNDEF` due to the lack of write event $\xrightarrow{rhb}$-before the read event.

If we are in case 4, there must likewise be a $\textsc{Free}$ event strictly before the read event, and the event is likewise given the `UNDEF` label.

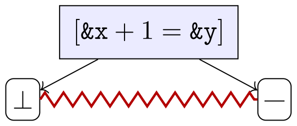
To arrive at cases 6 or 7, where $m' =$ `store` $m''\ \tau'\ b\ i\ v'$, there must be a write event to the same location before the read event. We cannot leave a read event without an $\textsc{rf}$ edge, as this would result in an incomplete execution, and we cannot construct an $\textsc{rf}$ edge to events which occurred $\leq$-before the $\leq$-latest write. This means that either we have an $\textsc{rf}$ edge from a write in the same thread, which must be the latest write to $(b, i)$ and of value $v$, or we have such an $\textsc{rf}$ edge from another thread which, in this trace, happened first. The inclusion of a $\textsc{rf}$ edge between these events enforces $v = v'$ and $\tau = \tau'$, ensuring that we must be in case 6 and not case 7.

To ensure the correctness of a store event, it suffices to show that its CompCert chain contains the action `alloc` $m\ l\ h$ which returned block $(b, s)$ where $i < s$. We know that the returned block will be reflected in the event label for the allocation event, meaning as long as some $(a : \textsc{Alloc}\ b\ s)$ appears in $T$ before $w$ and no event $(f : \textsc{Free}\ b)$ does, then the access must be live and in-bounds. These are both ensured by the relabelling conditions for writes.

$\square$

**Behaviour of the CompCert Semantics** As a consequence of the block-offset representation, pointer arithmetic cannot create a pointer to $y$ by performing arithmetic on a pointer to $x$. Two addresses in separate blocks will never be equal, regardless of offset.

```
rp := &x + 1;
if (rp = &y) {
    *rp := 1;
}
```



This program still has defined behaviour under the CompCert model. The illegal write never appears in the structure, because the comparison always returns false.

### 7.2.3 Provenance Models

Memarian et al. (2019) and Lepigre et al. (2022) define a set of candidate semantics based on the notion of *provenance*, which maintains an explicit relationship between pointer and object. Both note the potential pitfalls of a naïve definition, due to the relative frequency with which pointer-integer casts and pointer bit masking occur in real-world code. They also discuss varying approaches to preserving pointer provenance during casts and arithmetic on the integers obtained through casts, with the goal of presenting an easy-to-use semantics which modifies the behaviour of as little real-world C as possible.

These provenance models differ from the CompCert model in that pointers are a provenance-address pair rather than a block-offset pair. This permits inter-object pointer arithmetic and pointer aliasing, as two pointers can be constructed with the same address but differing provenances, while still allowing for machine-equivalent pointers to be treated as unequal by analysis passes.

In this section we give a rough overview of adapting the VIP model, presented in (Lepigre et al. (2022)), to symbolic MRD.

**The VIP Memory Layout Model** Pointers in VIP are represented as a pair $p = (@i, a)$ of a *provenance*, which may be some allocation id $@i$ or the empty provenance $@empty$, and an integer address value $a$. The model defines memory state as a pair $(A, M)$ of *allocation map* $A$ and *heap* $M$

$$A : alloc\_id \rightharpoonup \{base : address;\ length : \mathbb{N};\ killed : \mathbb{B}\} \qquad M : address \rightharpoonup mbyte$$

All operations over memory are described by a transition relation $(A, M) \rightarrow_{\mathrm{VIP}} (A', M')$. As before, we abstract away the majority of the memory state and instead ensure specific properties of $A$ and $M$ by requiring orderings over events.

The allocation and freeing operations are given by the VIP-ALLOC and VIP-KILL rules respectively:

$$\textrm{VIP-ALLOC}\ \frac{n = sizeof(\tau) \quad i \notin dom(A) \quad a \in new\_alloc(A, al, n) \quad p = (@i, a)}{(A, M) \rightarrow_{\mathrm{VIP}} (A[i \mapsto (a, n, alive)], M[a..a + n - 1 \mapsto (@empty, unspec, none)])}$$

$$\textrm{VIP-KILL}\ \frac{p = (@i, a) \quad A(i) = (a, n, alive)}{(A, M) \rightarrow_{\mathrm{VIP}} (A[i \mapsto (a, n, killed)], M)}$$

These give rise to the same restrictions on allocations and frees as in the CompCert model, namely that any store or load involving location $(@i, l)$ must be strictly after an ALLOC $@i\ l'\ s$ where $l' \leq l \leq l + s$ and before a FREE $l'$, and that every load from $l$ must occur after some store to $l$. When determining $\leq$ orderings, we use only the address component of the pointer and ignore provenance. We also ensure the `UNDEF` relabelling rule for FREE $l$ includes the case in which `free` has been passed a pointer which does not point to the start of the region.

Allocations ALLOC $@i\ \alpha\ s$ require a triple of a unique ID $i$, unique symbolic address $\alpha$, and (potentially symbolic) integer size $s$, and are $\leq$-before any accesses within the range $\alpha$ to $\alpha + s - 1$. Free events FREE $@i\ \alpha$ require the allocation ID and address, but must come strictly after an ALLOC $@i\ \alpha\ s$ event with the same allocation ID and address to have defined behaviour. Any free which does not come after such an allocation event takes the label `UNDEF`.

The loads and stores themselves are given by the VIP-LOAD and VIP-STORE rules:

$$\textrm{VIP-LOAD}\ [\mathbf{load}(\tau, \mathrm{p}) = \mathrm{v}]\ \frac{\begin{array}{c} p = (@i, a) \quad A(i) = (a, n, alive) \quad [a..a + |\tau| - 1] \subseteq [a_i..a_i + n_i - 1] \\ v = abst(A, \tau, M[a..a + |\tau| - 1]) \end{array}}{(A, M) \rightarrow_{\mathrm{VIP}} (A, M)}$$

$$\textsc{VIP-store}\ [\textbf{store}(\tau,\ \mathrm{p},\ \mathrm{v}) = ()]\ \frac{p = (@i, a) \quad A(i) = (a, n, alive) \quad n = |\tau| \\ [a..a + |\tau| - 1] \subseteq [a_i..a_i + n_i - 1]}{(A, M) \to_{\mathrm{VIP}} (A, M[a..a + n - 1 \mapsto repr(v)])}$$

**Translating VIP to Event Structures** We treat all global variables $x$ as having a special location $(@x, \&x)$ which never requires allocation, and let the address-of operation return the provenance-address pair.

We simplify out the complications surrounding representation bytes, again to avoid handling mixed-size accesses. We also only allow a single address to be written to in an atomic operation, thus only requiring $a_i \leq a \leq a_i + n_i$. An access to location $a$ is only permitted if the pointer holds value $(@i, a)$ and the access is strictly after Alloc $@i\ a_i\ n$, where the execution containing the access guarantees $a_i \leq a \leq a_i + n$.

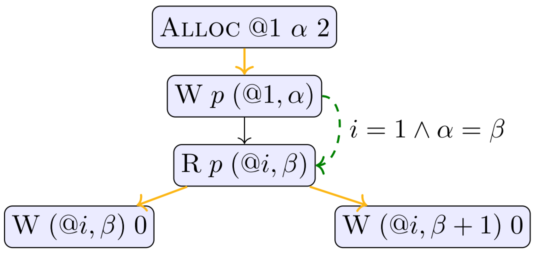
This gives us the program semantics $[\![P]\!]^{\mathrm{VIP}}_{n\ \rho\ \kappa\ \varphi}$, shown in Fig. 64.

$$
\begin{aligned}
[\![\texttt{rp := malloc s}]\!]^{\mathrm{VIP}}_{n\ \rho\ \kappa\ \varphi} &\triangleq [\varphi](e : \textsc{Alloc}\ @i\ \alpha\ [\![s]\!]_\rho) \bullet \kappa(\rho[rp \mapsto (b, 0)]) \\
[\![\texttt{free(rp)}]\!]^{\mathrm{VIP}}_{n\ \rho\ \kappa\ \varphi} &\triangleq [\varphi](e : \textsc{Free}\ \pi_2 \bullet \rho(rp)) \bullet \kappa(\rho) \\
(e : \mathrm{R}\ (@i, l)\ v) &\xrightarrow{\neg(a:\textsc{Alloc}\ @i'\ l'\ s)\xrightarrow{rhb} e \wedge i=i' \wedge l' \leq l < l'+s} (e : \texttt{UNDEF}) \\
(e : \mathrm{R}\ (@i, l)\ v) &\xrightarrow{(f:\textsc{Free}\ @i'\ l')\xrightarrow{rhb} e \wedge i=i'} (e : \texttt{UNDEF}) \\
(e : \mathrm{R}\ (@i, l)\ v) &\xrightarrow{\neg(w:\mathrm{W}\ (@i',l')\ v)\xrightarrow{rhb} e \wedge (@i,l)=(@i',l')} (e : \texttt{UNDEF}) \\
(e : \mathrm{W}\ (@i, l)\ v) &\xrightarrow{\neg(a:\textsc{Alloc}\ @i'\ l'\ s)\xrightarrow{rhb} e \wedge i=i' \wedge l' \leq l < l's} (e : \texttt{UNDEF}) \\
(e : \mathrm{W}\ (@i, l)\ v) &\xrightarrow{(f:\textsc{Free}\ (@i',l'))\xrightarrow{rhb} e \wedge i=i'} (e : \texttt{UNDEF}) \\
(e : \textsc{Free}\ (@i, l)) &\xrightarrow{\neg(a:\textsc{Alloc}\ @i'\ l'\ s)\xrightarrow{rhb} e \wedge (@i,l)=(@i',l')} (e : \texttt{UNDEF}) \\
[\![(@i, l) + i]\!]^{\mathrm{VIP}} &= (@i, l + i)
\end{aligned}
$$

Figure 64: The semantic interpretation and UNDEF relabellings for $[\![P]\!]^{\mathrm{VIP}}_{n\ \rho\ \kappa\ \varphi}$ without casts.

The correctness of this translation follows from the same reasoning as the CompCert translation, with chains of CompCert actions replaced by chains of operational rule applications.

```
p := malloc 2;
rp := p;
*rp := 0;
*(rp + 1) := 0;
```



**Handling Provenance Information** The VIP model also allows the reconstruction of provenance provided an address, via the COPY_ALLOC_ID command. This requires a pointer with allocation ID $@i$, and copies this allocation ID and a given address into a second pointer provided the address is within range of the original allocation.

$$\text{VIP-COPY-A-ID}[\textbf{copy\_alloc\_id}(\text{x, p1}) = \text{p2}]\frac{\begin{array}{c} p_1 = (@i, \_) \quad A(i) = (a, n, alive) \\ to\_int(x) \in [a..a+n] \quad p_2 = (@i, to\_int(x)) \end{array}}{(A, M) \rightarrow_{\text{VIP}} (A, M)}$$

We represent this as part of the expression language, as it does not alter the memory state, returning the new allocation ID and address according to the VIP-COPY-A-ID rule and generating a $\tau$ event with the appropriate label.

$$[\![\texttt{copy\_alloc\_id(a, r)}]\!]^{\text{VIP}}_{\rho} = (\pi_1(\rho(r)), a)$$

Equality between locations in VIP checks both the allocation ID and the address. If the allocation IDs and addresses are both equal, the equality returns true. If both are unequal, it returns false. If the allocation IDs are unequal but addresses are equal, it nondeterministically returns true or false.

$$((@i_1, a_1) = (@i_2, a_2)) \triangleq \begin{cases} true & \text{if } @i_1 = @i_2 \wedge a_1 = a_2 \\ b \in \{true, false\} & \text{if } a_1 = a_2 \\ false & \text{otherwise} \end{cases}$$

Any branch which checks pointer equality therefore guarantees $a_1 = a_2$ when it holds, as it could have observed the first or second case, and $@i_1 \neq @i_2$ when it does not, as it could have observed the second or third case.

$$[\![(@i, a) = (@i', a')]\!]^{\text{VIP}} = (a = a')$$
$$[\![(@i, a) \neq (@i', a')]\!]^{\text{VIP}} = (i \neq i')$$

To avoid using provenance information and perform computations using machine addresses only, pointers in VIP can be cast to integers and back. Casting a pointer to an integer is only defined if the pointer location is live and attaches the pointer's provenance to the integer, while casting an integer to a pointer either attaches the integer's provenance to the pointer or returns a pointer with empty provenance.

$$\text{VIP-}\textsc{p-i-cast}[\textbf{cast\_pval\_to\_ival}(\mathcal{T},\text{ p}) = \text{x}]\frac{\begin{array}{c}p = (@i, a) \quad A(i) = (\_, \_, alive)\\ x = Loc(@i, a) \quad a \in value\_range(\tau)\end{array}}{(A, M) \rightarrow_{\text{VIP}} (A, M)}$$

$$\text{VIP-}\textsc{i-p-cast}\text{-1}[\textbf{cast\_ival\_to\_pval}(\mathcal{T},\text{ x}) = \text{p}]\frac{\begin{array}{c}p = (@i, a) \quad A(i) = (a_i, n_i, alive)\\ x = Loc(@i, a) \quad a \in [a_i..a_i + n]\end{array}}{(A, M) \rightarrow_{\text{VIP}} (A, M)}$$

$$\text{VIP-}\textsc{i-p-cast}\text{-2}[\textbf{cast\_ival\_to\_pval}(\mathcal{T},\text{ x}) = \text{p}]\frac{\begin{array}{c}x = Int(z) \quad z \in valid\_addresses\\ p = (@empty, z)\end{array}}{(A, M) \rightarrow_{\text{VIP}} (A, M)}$$

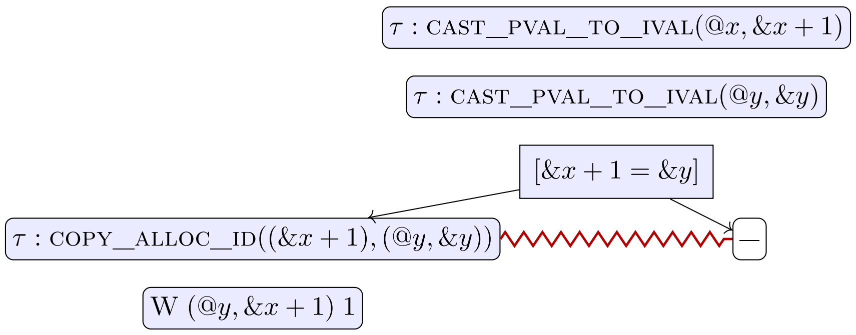
The VIP-I-P-CAST1 rule only applies to integers gained directly from casting a pointer to an integer, as any arithmetic operations over integers alter them from *Loc* type to *Int* type. These can also be subsumed into the expression language and given the appropriate `UNDEF` relabellings when the referenced addresses are no longer live or not within an allocated region. These rules are shown in Fig. 65.

Using casting, the one-past pointer construction example can be defined under VIP and may terminate with the value of $y$ altered.

```
rp := &x + 1;
r1 := cast_pval_to_ival rp;
r2 := cast_pval_to_ival &y;
if (r1 = r2) {
    rp := copy_alloc_id (r1, &y)
    *rp := 1;
}
```

$$
\begin{aligned}
\llbracket \texttt{cast\_pval\_to\_ival(@i, a)} \rrbracket^{\text{VIP}} &= \mathit{Loc}(@i, a) \\
(e : \tau : \text{CAST\_PVAL\_TO\_IVAL}(@i, a)) &\xrightarrow{a \notin \mathit{value\_range}(\tau)} (e : \texttt{UNDEF}) \\
(e : \tau : \text{CAST\_PVAL\_TO\_IVAL}(@i, a)) &\xrightarrow{\neg (a{:}\textsc{Alloc}\ @i'\ a'\ s) \xrightarrow{rhb} e \wedge i = i'} (e : \texttt{UNDEF}) \\
(e : \tau : \text{CAST\_PVAL\_TO\_IVAL}(@i, a)) &\xrightarrow{(f{:}\textsc{Free}\ (@i', a')) \xrightarrow{rhb} e \wedge i = i'} (e : \texttt{UNDEF}) \\
\llbracket \texttt{cast\_ival\_to\_pval(Loc(@i, a))} \rrbracket^{\text{VIP}} &= (@i, a) \\
(e : \tau : \text{CAST\_IVAL\_TO\_PVAL}(\mathit{Loc}(@i, a))) &\xrightarrow{\neg (\textsc{Alloc}\ (@i\ a_i\ s) \in E \wedge a_i \leq a \leq a_i + s)} (e : \texttt{UNDEF}) \\
(e : \tau : \text{CAST\_IVAL\_TO\_PVAL}(\mathit{Loc}(@i, a))) &\xrightarrow{(f{:}\textsc{Free}\ (@i', a')) \xrightarrow{rhb} e \wedge i = i'} (e : \texttt{UNDEF}) \\
\llbracket \texttt{cast\_ival\_to\_pval(Int(z))} \rrbracket^{\text{VIP}} &= (@\mathit{empty}, z) \\
\llbracket \mathit{Loc}(@i, a) + e \rrbracket^{\text{VIP}} &= \mathit{Int}(\llbracket a + e \rrbracket^{\text{VIP}}) \\
\llbracket \mathit{Loc}(@i, a) - e \rrbracket^{\text{VIP}} &= \mathit{Int}(\llbracket a - e \rrbracket^{\text{VIP}})
\end{aligned}
$$

Figure 65: The extra expression language semantics and `UNDEF` relabelling rules for the semantics $\llbracket P \rrbracket^{\text{VIP}}_{n\ \rho\ \kappa\ \varphi}$ with casts.


$\tau$ : CAST_PVAL_TO_IVAL$(@x, \&x + 1)$
$\tau$ : CAST_PVAL_TO_IVAL$(@y, \&y)$
$[\&x + 1 = \&y]$
$\tau$ : COPY_ALLOC_ID$((\&x + 1), (@y, \&y))$
—
W $(@y, \&x + 1)$ 1

## 7.3 Exploring the Differences Between Pointer Semantics

```
rp := malloc 1;
p := rp;
free(rp);
rq := malloc 1;
q := rq;
if (rp = rq) {
    x := 1;
} else {
    y := 1;
}
```

In this program, an implementation could make one of three decisions:

1. I am always going to reclaim the location used for $q$ when I allocate $p$, so I can optimise this guard to true and reorder the write to $x$.

2. The pointers $q$ and $p$ have different provenances, so I can optimise this guard to false and reorder the write to $y$.

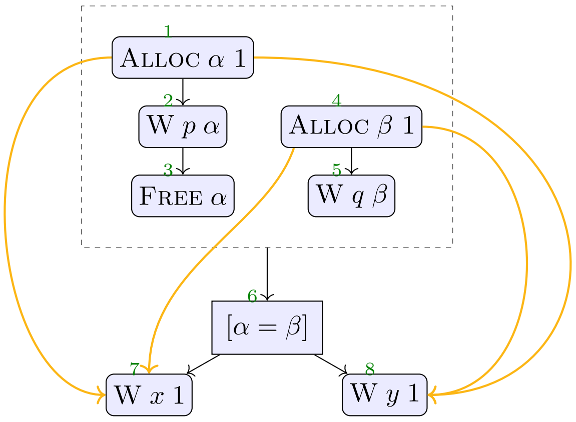
3. I don't know what the addresses of $p$ or $q$ will be until runtime, so I cannot optimise out this guard.

These decisions are ultimately based on the guarantees that the various models make about the behaviours of `malloc` and the equality operator over pointers. The first option could be considered a security risk, as it leaks layout information. Determining the location of critical regions during runtime has been found to be a common feature in various real-world exploits (Xu, Kalbarczyk and Iyer (2003)), and an early write of $x$ would indicate that $p$ and $q$ are guaranteed to alias whenever the program executes.

The second option leads to the variables $x$ and $y$ being unusable as flag variables, as the write to $y$ may become visible before the allocations have completed, but represents a commonly-used optimisation which is allowed by the current C standard. If the optimisation is permitted on pointers with unequal provenance but equal machine addresses, it leaks no layout information. If it is only permitted on pointers whose addresses are guaranteed to be unequal, it leaks substantially less layout information than in the equality case but still a non-zero amount.

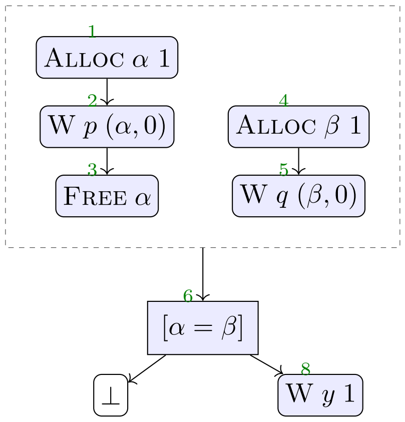
The third option restricts compiler optimisations the most, but forbids any layout leaks and provides defined behaviour in the most contexts.

### 7.3.1 Plain Integers



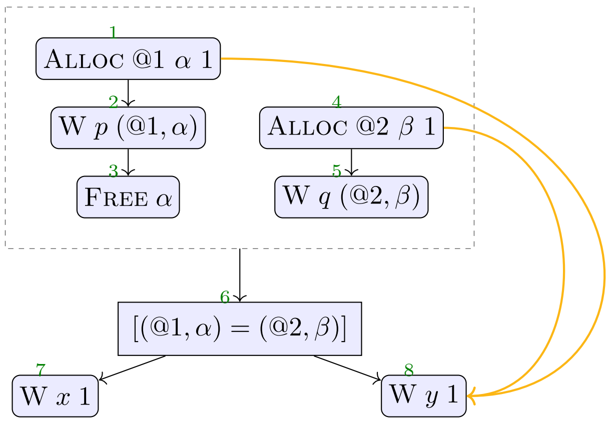
The plain integer semantics requires that any execution which does not choose $1 \xrightarrow{xco} 4 \xrightarrow{xco} 3$ ensure that $\alpha$ and $\beta$ are unequal, but in every execution we have $1 \xrightarrow{xco} 3 \xrightarrow{xco} 4$ as these are in the same thread. Without any information about locations available to the dependency calculation mechanisms, there will always be dependency edges between events 1 and 4, the origins of $\alpha$ and $\beta$ respectively, and events 7 and 8. The true branch cannot be taken if the free occurs after $\beta$ is allocated, but the writes will still only be visible after both allocations have completed in any execution.

### 7.3.2 CompCert



The CompCert semantics simply declares the values disjoint, as they have separate allocation sites and are therefore in separate blocks. We begin with the pre-justification $([\alpha \neq \beta], \emptyset) \vdash (8 : \text{W } y \ 1)$, but we also have the program guarantee $\alpha \neq \beta$ and can thus simplify this to $(\top, \emptyset) \vdash (8 : \text{W } y \ 1)$, making event 8 independent.

### 7.3.3 VIP



In the VIP semantics, the two locations will have separate identifiers and the equality check will therefore be nondeterministic. Event 7 has the pre-justification $([\alpha = \beta], \emptyset) \vdash (7 : \mathrm{W}\ x\ 1)$, which cannot be simplified. Event 8, however, has the pre-justification $([1 \neq 2], \emptyset) \vdash (8 : \mathrm{W}\ y\ 1)$, which simplifies to $(\top, \emptyset) \vdash (8 : \mathrm{W}\ y\ 1)$.

This means that any execution, regardless of the addresses returned by `malloc`, may perform the write to $x$ early, but executions in which the addresses are equal may instead perform the write to $y$ after checking for equality.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusion

The problem of constructing a formal semantics for shared memory concurrency with relaxed accesses in a real-world programming language has been an open one for at least two decades, with early work by Adve and Hill (1990) establishing the definitions for weak ordering. In this work, we have presented a number of steps towards that goal.

In Chapter 3 we introduced a method of giving a weakly ordered semantics to a programming language, and in Chapter 5 we translated that weakly ordered semantics into a weakly ordered program logic. This allows the integration of our approach into an existing body of reasoning tools, increasing practical useability and allowing smoother integration of racy, weakly ordered fragments into otherwise race-free, sequentially consistent programs. In Chapter 6 we introduce a more abstract semantics for values returned by relaxed accesses, which not only simplifies the representation of the majority of programs but also allows us to introduce dynamic memory allocation in Chapter 7. This is the first model capable of giving a semantic interpretation of `malloc` in a weak memory context, and, as illustrated in Section 7.2, it is not strongly tied to a single model of a pointer's value. This not only allows us to represent a substantially larger portion of value C and C++ programs, but also indicates that any future changes to pointer semantics adopted by the standard, such as a provenance model, are unlikely to invalidate the correctness of the model.

Throughout Chapters 3, 6 and 7, we have been able to illustrate which reorderings may be visible exclusively by interpreting a single thread. This succinctly illustrates the advantage of context-ambiguous, nonlinear semantics such as MRD over semantics which require adequate justification for every intermediate step such as the JMM and Promising: abstracting away context is done for free by the semantics rather than being an additional burden placed on any correctness proofs for program fragments, resulting in more predicable behaviour in a wider variety of contexts. This not only has advantages in verifying small critical sections of a program, as the relaxed behaviours can be more effectively enumerated, but also in discovering abnormal behaviours permitted by the semantics. If two events are ordered by MRD then the mechanisms by which that order may be broken immediately give rise to the set of contexts in which they are unordered.

## 8.2 Future Work

### 8.2.1 Program Logics Over Symbolic MRD

The program logic presented in Chapter 5 functions entirely over concrete MRD, and while we introduce some notational shorthand to better handle equivalently-behaving executions, the logical representation of control dependencies in symbolic MRD present an interesting interface to a potential program logic.

### 8.2.2 Integration of Canonical Synchronisation Orders

Programs which use purely weak memory are vanishingly uncommon, with most shared memory concurrency using some hardware and software provided synchronisation primities. These include release/acquire memory orders on reads and writes, lock/unlock operations on mutexes, and thread fence operations. As the concrete value MRD model has already integrated the synchronisation orders from the IMM model of Podkopaev, Lahav and Vafeiadis (2019) and the RC11 model of Lahav et al. (2017), a similar integration into symbolic MRD is a natural extension. These models provide formal descriptions of various synchronisation primitives available on widely-used hardware specifications which have already been shown to be correct, thus their inclusion would bring MRD a substantial step closer to being able to represent real-world code.

### 8.2.3 Tooling and Automation

The MRDer tool (Paviotti et al. (2020)) is an OCaml program capable of providing an MRD denotation for an input program over a toy language and value range. Given the reduction in both structure size and number of coproduct operations created by symbolic MRD, the creation of a symbolic MRDer variant is a potentially worthwhile goal. Some initial work has been done on adapation, including hooks to the Z3 SMT solver (Moura and Bjørner (2008)) for condition satisfiability checks, but the construction and evaluation of a full-scale tool remains future work.

### 8.2.4 Completeness Results Over Optimisations

An ideal property of any calculation of semantic dependency would be that, given program $P$ and program transformation $\tau$, if we are able to build some correctness proof for $\tau$ then the dependencies of $[\![P]\!]_{n\ \rho\ \kappa\ \varphi}$ are in some sense equivalent to the dependencies of $[\![\tau(P)]\!]_{n\ \rho\ \kappa\ \varphi}$. This decouples the dependency calculation from extant sets of optimisations and effectively future-proofs it against future developments in optimisation passes. The comparison of dependency calculations to an extant system of showing correctness of program transformations, such as the Relational Hoare Logic of Benton (2004), could be illuminating for both the MRD model and the target proof technique.

### 8.2.5 Mixed Size Accesses and Alignment

As stated in Chapter 7, mixed size accesses are not yet represented in symbolic MRD due to their relative complexity. Work has been done by Flur et al. (2017) and Alglave et al. (2021) on extending memory models to handle mixed size accesses, providing a guideline for how to adapt events and synchronisation orders to a mixed-size setting, but the calculation of mixed-size semantic dependency may be complicated by overlapping but nonequal memory footprints between events.

### 8.2.6 Dependency Beyond Weak Memory: Garbage Collection

The implementation of garbage collection in a language which allows pointer manipulation is rare, but it does occur. The Microsoft .NET framework includes garbage collection for C# programs, while a small number of garbage collection libraries have been written for C++ such as the Boehm-Demers-Weiser garbage collector (Boehm, Demers and Weiser (1988)) and Oilpan.

Adding memory management to a language may naturally create shared memory concurrency even for well-synchronised programs which would ordinarily use more stringent memory models. The problem of finding a safe point, a point at which the collector may modify the heap without impacting the behaviour of other threads, can be modelled by creating a context which touches all global variables in the program and finding the points at which this does not change the behaviour of the program or cause a data race. In single-threaded programs there is minimal overhead to having sparse safe points within a program, but if a program contains multiple threads then they must *all* be at a safe point before the garbage collector can run (Agesen (1998)), causing a reduction in program performance proportional to the number of threads (Jones and King (2005)). This implies a potential performance benefit to statically finding closely packed safe points.

```
m := malloc 5;
p := m + 1;
q := m + 4;
                      STOP;
                      r1 := *p;
1: r1 := p;           r2 := *q;
2: r2 := q;           p := m
3: *r1 := 1;     ||   q := m + 1;
4: r3 := *r2;         *p := r1;
                      *q := r2;
                      START;
```

In this program, something like a compacting garbage collector in a second thread has moved pointers $p$ and $q$. To represent a stop-the-world garbage collector, we propose the language primitives `STOP` and `START`, and enforce via $\textsc{lk}$ that no event in a thread external to a `STOP`/`START` pair may be between `STOP` and `START` via lock order. These are similar to standard lock/unlock pairs, but order external threads more strictly in the absence of any corresponding lock/unlock event local to that thread.

$$(e:\textsc{stop}) \bullet S \triangleq \{(\{e\}\cup E, \textsc{rf}_E, \textsc{dp}_E, \leq_E \cup \leq, \xrightarrow{xco}_E, \textsc{lk}_E \cup \textsc{lk}, \psi_E \wedge \psi) \mid (E, \textsc{rf}_E, \textsc{dp}_E, \leq_E, \xrightarrow{xco}_E, \textsc{lk}_E, \psi_E) \in S\}$$

$$\text{Where: } \leq\ = \{(e,e')| e' \in E \wedge \nexists (s:\textsc{start}) \leq e'\}$$

$$\textsc{lk} = \{(e,e') \mid (e' : \textsc{Alloc}\ l') \in E \vee (e' : \textsc{Free}\ l') \in E\}$$

Similarly, appending any event to a structure will place it $\leq$-before any $\textsc{stop}$ or $\textsc{start}$ event.

Without any further synchronisation, it is possible for this program to terminate with the value 1 stored in $r_3$. Pointer $p$ can be pointing to address $m+1$ when its value is loaded into $r_1$, and likewise pointer $q$ may have been moved to $m+1$ before its value is loaded into $r_2$. This means the write `*r1 := 1` stores the value 1 at location $m+1$ and the subsequent read `r3 := *r2` loads from this location.

Annotating the program text with safe point indicators would require us to know the register contents at a syntactic point in the program, which naturally conflicts with our established semantic model. Annotating *configurations* with safe points allows us to establish that, if we have executed lines 1 and 3, lines 2 and 4, or all or none of these, we can run the parallel thread without impacting correctness. This gives us two more safe points than function entry and function exit, which means that if another thread is being blocked by the garbage collector it will have less time to wait.

Using the semantic dependencies framework, garbage collection can be framed as an event ordering problem: we are at a safe point, if, for every pointer $p$, if we have executed some read $\mathrm{R}\ p\ \alpha$ then we have passed all references to $\alpha$. We can formalise this into the constraint that if $(r : \mathrm{R}\ p\ \alpha) \xrightarrow{\text{DP}} e$ where $\alpha$ appears in the label of $e$, then $e \xrightarrow{xco} (s : \textsc{stop})$ should be enforced by the implementation of the garbage collector. Alternatively, we can describe safe points as the inverse of *unsafe sections*, and unsafe sections as configurations in which our execution set contains some $\mathrm{R}\ p\ \alpha$ but has yet to perform some operation at $\alpha$.

# Bibliography

(2014). LLVM's analysis and transform passes — LLVM 3.6 documentation. Tech. rep., LLVM Project.

Adve, S. V. and Hill, M. D. (1990). Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, New York, NY, USA: Association for Computing Machinery, ISCA '90, p. 2–14.

Agesen, O. (1998). Gc points in a threaded environment. Tech. rep., Sun Microsystems.

Alglave, J., Maranget, L., Sarkar, S. and Sewell, P. (2011). ARM litmus tests. https://www.cl.cam.ac.uk/ pes20/arm-supplemental/.

Alglave, J., Deacon, W., Grisenthwaite, R., Hacquard, A. and Maranget, L. (2021). Armed cats: Formal concurrency modelling at Arm. *ACM Trans Program Lang Syst*, 43(2).

Batty, M. et al. (2019). D1780r0 modular relaxed dependencies: A new approach to the out-of-thin-air problem. Tech. rep., International Standards Organization.

Benton, N. (2004). Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not*, 39(1), p. 14–25.

Bodík, R., Gupta, R. and Soffa, M. L. (1997). Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, PLDI '97, p. 146–158.

Boehm, H., Demers, A. and Weiser, M. (1988). A garbage collector for C and C++. https://www.hboehm.info/gc/.

Boehm, H.-J. and Demsky, B. (2014). Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pp. 1–6.

Calder, B., Grunwald, D. and Zorn, B. (1994). Quantifying behavioral differences between c and c++ programs. *Journal of Programming languages*, 2(4), pp. 313–351.

Chakraborty, S. and Vafeiadis, V. (2019). Grounding thin-air reads with event structures. *Proc ACM Program Lang*, 3(POPL).

Dalvandi, S., Doherty, S., Dongol, B. and Wehrheim, H. (2020). Owicki-Gries Reasoning for C11 RAR. In R. Hirschfeld and T. Pape, eds., *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 166, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 11:1–11:26.

Dalvandi, S., Dongol, B., Doherty, S. and Wehrheim, H. (2022). Integrating owicki–gries for c11-style memory models into isabelle/hol. *Journal of Automated Reasoning*, 66, pp. 141–171.

Doherty, S., Dongol, B., Wehrheim, H. and Derrick, J. (2019). Verifying C11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA: Association for Computing Machinery, PPoPP '19, p. 355–365.

Doko, M. and Vafeiadis, V. (2016). A program logic for c11 memory fences. In B. Jobstmann and K. R. M. Leino, eds., *Verification, Model Checking, and Abstract Interpretation*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 413–430.

Flur, S. et al. (2017). Mixed-size concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not*, 52(1), p. 429–442.

Jagadeesan, R., Jeffrey, A. and Riely, J. (2020). Pomsets with preconditions: A simple model of relaxed memory. *Proc ACM Program Lang*, 4(OOPSLA).

Jeffrey, A. and Riely, J. (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*, New York, NY, USA: Association for Computing Machinery, LICS '16, p. 759–767.

Jones, R. and King, A. (2005). A fast analysis for thread-local garbage collection with dynamic class loading. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pp. 129–138.

Kotzmann, T. et al. (2008). Design of the java hotspot™ client compiler for java 6. *ACM Trans Archit Code Optim*, 5(1).

Lahav, O. and Vafeiadis, V. (2015). Owicki-gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi and B. Speckmann, eds., *Automata, Languages, and Programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 311–323.

Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K. and Dreyer, D. (2017). Repairing sequential consistency in C/C++ 11. *ACM SIGPLAN Notices*, 52(6), pp. 618–632.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9, pp. 690–691.

Lepigre, R. et al. (2022). VIP: Verifying real-world C idioms with integer-pointer casts. *Proc ACM Program Lang*, 6(POPL).

Leroy, X., Appel, A. W., Blazy, S. and Stewart, G. (2012). The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA.

Leroy, X. et al. (2016). CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France: SEE.

Mador-Haim, S. et al. (2012). An axiomatic memory model for POWER multiprocessors. In P. Madhusudan and S. A. Seshia, eds., *Computer Aided Verification*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 495–512.

Memarian, K. et al. (2016). Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, PLDI '16, p. 1–15.

Memarian, K. et al. (2019). Exploring C semantics and pointer provenance. *Proc ACM Program Lang*, 3(POPL).

Moura, L. d. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340.

Mueller, F. and Whalley, D. B. (1995). Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pp. 56–66.

Nielsen, M., Plotkin, G. and Winskel, G. (1981). Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1), pp. 85–108, special Issue Semantics of Concurrent Computation.

Paviotti, M. et al. (2020). Modular relaxed dependencies in weak memory concurrency. *Programming Languages and Systems LNCS 12075*, p. 599.

Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *SIGPLAN Not*, 51(1), p. 622–633.

Podkopaev, A., Lahav, O. and Vafeiadis, V. (2019). Bridging the gap between programming languages and hardware weak memory models. *Proc ACM Program Lang*, 3(POPL).

Pugh, W. (1999). Fixing the java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pp. 89–98.

Pugh, W. (2004). Causality test cases. http://www.cs.umd.edu/ pugh/java/memoryModel/CausalityTestCases.html.

Pugh, W., Adve, S. and Lea, D. (2011). JSR 133: Java(TM) memory model and thread specification revision. https://download.oracle.com/otndocs/jcp/memory_model-1.0-pfd-spec-oth-JSpec/.

Pulte, C. et al. (2017). Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc ACM Program Lang*, 2(POPL).

Sarkar, S., Sewell, P., Alglave, J., Maranget, L. and Williams, D. (2011a). Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 175–186.

Sarkar, S., Sewell, P., Alglave, J., Maranget, L. and Williams, D. (2011b). Understanding power multiprocessors. *SIGPLAN Not*, 46(6), p. 175–186.

Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z. and Myreen, M. O. (2010). X86-TSO: A rigorous and usable programmer's model for X86 multiprocessors. *Commun ACM*, 53(7), p. 89–97.

Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R. and Zappa Nardelli, F. (2015). Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 209–220.

Ševčík, J. and Aspinall, D. (2008). On validity of program transformations in the java memory model. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, Berlin, Heidelberg: Springer-Verlag, ECOOP '08, p. 27–51.

Wright, D., Batty, M. and Dongol, B. (2021). Owicki-gries reasoning for C11 programs with relaxed dependencies. In *International Symposium on Formal Methods*, Springer, pp. 237–254.

Xu, J., Kalbarczyk, Z. and Iyer, R. (2003). Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pp. 260–269.