Channels at the head of the queue!

Relaxed Memory Models and Data-Race Detection tailored for Shared-Memory Message-Passing Systems

Doctoral Dissertation by

Daniel S. Fava

Submitted to the
Faculty of Mathematics and Natural Sciences at the University of Oslo
for the degree Philosophiae Doctor in Computer Science

June 2021

Reliable Systems
Department of Informatics
University of Oslo
Norway

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2407

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

Abstract

Instructions, as they appear in a program's text, dictate the behavior of single-threaded programs. Unfortunately, our single threaded intuition does not fully carry over to multi-threaded environments. If the environment is *sequentially consistent*, then we can understand a multi-threaded program in terms of its text—more specifically, in terms of how instructions from different threads can be interleaved. Modern computing environments, however, are not sequentially consistent. As a consequence, the behavior of a multi-threaded program depends not only on the program text, but also on the surrounding hardware and compiler optimizations. The overall effect of an environment and its optimizations are documented in a *memory model.*

In this thesis, we explore the specification of memory models using a *small step operational semantics*. By being concise and executable, we believe that operational semantics can build on our shared understanding of programming. We explore the specification of *relaxed* or *weak memory models,* meaning that these models accommodate optimizations that are precluded under sequential consistency. We are inspired by Go, an open-source programming language developed at Google that has gained traction in the area of cloud computing. Like in Go, we focus on the exchange of messages via channels as the primary means of synchronization.

On a theoretical level, we present the specification of a relatively relaxed model in which writes are delayed. We show two important properties that apply in this setting: the absence of *out-of-thin-air behavior* (OOTA), and the presence of the DRF-SC guarantee, which states that Data-Race Free (DRF) programs behave Sequentially Consistently (SC). We explore mechanisms for extending this initial memory model in support of load buffering, and discuss challenges encountered along the way.

In addition to weak memory, we visit the issue of correctness of multi-threaded applications. Specifically, we investigate the detection of data races in programs that synchronize via channel communication. To that end, we introduce a data-race detector based on what we call *happens-before sets.*

We contrasted our operational semantics and data-race detector specifications against the Go implementation. As a result, we found a discrepancy between the Go implementation and its memory model specification—this mismatch led to the under-reporting of data races in Go programs. Our theoretical investigation guided us to a solution (*i.e.,* a patch) that has been merged and released with Go 1.16. In this thesis, we share our experience applying formal methods to real-world software.

Contents

Preface

As cliché as it sounds, the story of my PhD started in my childhood. Growing up, I wanted to be a scientist. In my teenage years, I had my eyes on a joint math and physics program[1] at the *Universidade Estadual de Campinas* (UNICAMP), Brazil. I started the program in 2001 and still remember Professor Alcibíades Rigas class on linear algebra and analytical geometry. Rigas used a graduate level book written in English as the main resource for this freshman level course—in a country where most don't speak English. It's an understatement to say that the class was hard. But rather than an exercise in gratuitous punishment, Rigas helped us build a solid foundation. I fell in love with the campus and the program, but I left midway through my freshman year. While taking entrance exams in Brazil, I had also submitted applications for college in the US. When notified of my acceptance at the Rochester Institute of Technology, I chose the unknown over a life I had been falling in love with.

Moving to the US was a momentous decision for me. Leaving a liberal, public (free) university that is strong in theory and the sciences and going to a paid, conservative school with a focus on industrial application... had I made a big mistake? The feeling of isolation, the cold, and the political climate post 9/11 weighed hard. But I also made life-long friends during this time, and learned to embraced the engineer in me. In the end, RIT did prepare us well for industry. After college, I worked at Intel on one of the first many-core CPU architectures. At Apple, I worked on the first 64-bit cellphone processor to hit the market. But my childhood dream of being a scientist looked far away in the rear-view mirror. So on the verge of becoming a father, with the encouragement and support of my wife, I took an order of magnitude pay-cut and made a u-turn into graduate school.

I enrolled in the PhD program in Computer Science at the University of California in Santa Cruz, CA. Wanting to find my way back to math and science, I took classes in machine learning and in the theory of programming languages. I became interested in logic and was exposed to formal methods. But I struggled to find my footing, and life in the US was not easy for two graduate students with a kid. With the help of the Good Country Index,[2] I made a list of potential places to live. A serendipitous e-mail from Olaf (now my PhD co-advisor) and the support from amazing friends put us in motion towards Norway.

At the University of Oslo, I continued studying programming languages and formal methods. In this thesis you may sense the pushes and pull of a person with mixed interests. The operational semantics and the proof by simulation that appear early in the document come from wanting to deepen my mathematical background.

[1] The program was known as *cursão* or *Curso 51,* and the idea of combining the subject seemed, like *Caninha 51, uma boa ideia.*

[2] `https://index.goodcountry.org/`

The work of manipulating symbols in a formal system, however, is more fitting to a theoretician than to the engineer who I had become. So I am grateful to Martin, my advisor, for taking my interest and curiosity seriously, for encouraging me to develop my own research style, and for helping me bridge my knowledge gap.

I also wanted to build a modest trail, starting with real world source code and veering towards math. A trail that someone like my past self—a programmer who aspires to learn more but who does not yet have graduate-level training—might find useful. With the goal of bringing the thesis' work to practice, I began looking at source code again. My exposure to industrial code bases and my experience dealing its complexities helped me a lot. I studied the thread sanitizer library (TSan), the Go data race detector, and the implementation of channels in the Go runtime. What started as tinkering developed into the latter part of the thesis. In the process I was exposed to open-source development, which I have been interested in since my undergraduate studies.

I am tremendously grateful for the journey. Risking opening *and finishing* with a cliché: I hope you will find the work interesting. Thank you.

Publications

The contents of this thesis is based on research articles that are published in the following international, peer-reviewed computer science conference proceedings and journals:

- Finding and fixing a mismatch between the Go memory model and data-race detector. Daniel Fava. In **SEFM 2020***: Proceedings of the 18th International Conference on Software Engineering and Formal Methods*.

- Ready, set, Go! Data-race detection and the Go language. Daniel Fava and Martin Steffen. **Science of Computer Programming**, 195:102473, 2020.

- Operational semantics of a weak memory model with channel synchronization. Daniel Fava, Martin Steffen, and Volker Stolz. **Journal of Logial and Algebraic Methods in Programming**, 103:1 – 30, 2019. An extended version of the FM'18 publication with the same title.

- Operational semantics of a weak memory model with channel synchronization. Daniel Fava, Martin Steffen, and Volker Stolz. In **FM 2018***: Proceedings of the 22nd International Symposium on Formal Methods*, volume 10951, pages 1–19.

I have been involved as a main contributor on each one of these articles, and they were written in their entirety between 2017 and 2020 for the purpose of supporting this thesis.

Acknowledgments

I want to thank Martin Steffen, my primary advisor, for sharing with me the seeds from which I was able to grow this PhD thesis. I also want to thank Olaf Owe, my secondary advisor, for his kindness and for taking interest in us students, and in our well-being inside and outside of academia.

I would like to thank Volker Stolz for encouraging my visit to the *Universidade Federal de Campina Grande* in 2019, and to thank Tiago Massoni and Rohit Gheyi for the hospitality. During that same trip, I also had the pleasure of meeting additional members of the Brazilian formal methods community at the XXII Simpósio Brasileiro de Métodos Formais in São Paulo.

The story of my PhD is not complete without mentioning my sons. Lucas was born during my first quarter at the University of California, Santa Cruz, while David was born a year into my studies at the University of Oslo. Their coming encouraged me to pursue my childhood dream of becoming a scientist, and to seek a calm, harmonious setting for our family. Most of all, these guys have added a lot of joy and play into my life.

I am grateful to be sharing my life's journey with wife, and I am thankful for her unwavering support. I would also like to thank my father for always rooting for me—regardless of my choices and their outcome— and for giving my sister and I an Intel 80386, even though computers were an uncommon household item at the time. I would like to thank Renato and Elisabeth for their friendship and for the help and encouragement when moving to Norway. Finally, I want to thank my colleagues and the staff at both the University of California, Santa Cruz, as well as at the University of Oslo, Norway.

I dedicate this book to my mother, my sister, and my wife.

— Daniel S. Fava, Oslo, December 2020.

Introduction 1

Everything should be made as simple as possible, but not simpler.

Albert Einstein

It is fascinating when computing systems, which we think of as exact and well defined, harbor inconsistencies. So many inconsistencies, in fact, that we may no longer agree on what a program is supposed to do. The programs we will see in this chapter are only few lines long, yet there will be lots to talk about. This chapter will give you an appreciation for concurrency; here we discuss issues that you may not have encountered before. We tackle concurrency and synchronization from the point of view of channels, where instead of using locks, threads synchronize by sending each other messages. In this thesis, we formally define a memory model and a data-race detector based on message passing, and we put our theory in perspective by relating it to the Go programming language. Here, we go over the question of what happens when channels are a first-class language construct, as opposed to a construct derived from lower level primitives.

We start with the following example, where the initial value of z is zero and *done* is `true`. What does this program do?

```
  $T1$                 $T2$
$z := 42$            if ($done$)
$done$ := true         print ($z$)
```

There are two threads running concurrently, $T1$ and $T2$, and they are trying to synchronize with each other. $T1$ performs a task, represented here by the setting a shared variable z to 42. $T2$ checks if the work is done, and, if so, the thread prints the value of z.

The answer to “what does this program do” depends on two main factors. One is *interleaving*: the order in which instructions from the different threads execute. For example, $T2$ may execute before $T1$ sets *done* to true, in which case nothing is printed. Alternatively, $T1$ may execute before $T2$, in which case 42 is printed. Concurrent systems are challenging, in part, because each interleaving is potentially associated with a different program outcome, and there can be an immense number of interleavings (as interleavings grow factorially with the number of threads and of atomic blocks in each thread). Over the years, we have learned, for example, to leverage the fact that not all interleavings are visibly different. Two operations may produce the same result regardless of the order in which they execute—we say that these well behaved operations commute. By collapsing interleavings that yield the same result, we have gained some ground dealing with concurrency.

The other factor affecting a concurrent program's outcome is the *memory model*—"a formal specification of how the memory system will appear to the programmer" [1]. Going back to our example, even if the branch-guard in $T2$ observes the value of *done* to be true, it does not mean the thread will print 42. In many execution environments, it is possible for $T2$ to still print zero (or some uninitialized value). So, even when setting interleaving aside and focusing on one execution, the answer to "what does this program do" is not obvious. Here, we will explore the origins and consequences of this complexity.

1.1 A bird's eye view of the history of parallelism

Gordon Moore, the co-founder of the semiconductor company Intel, observed that the number of integrated-circuit components doubles from year to year, roughly. In the first few decades of computing, these new components have accounted for increases in performance through the exploitation of what is called *instruction-level parallelism* (ILP). The idea, which can be traced back to the 1940s [85], involves speeding-up a single-threaded program by, for example, overlapping the execution of instructions, or by changing the order in which instructions execute. The exploitation of instruction-level parallelism has been an integral part of computing. In fact, decades of research and development towards improving single-threaded performance have left lasting impressions on modern hardware designs.

With time, however, it became increasingly difficult to speed up single-threaded applications in a power-efficient way. By the mid 2000s, industry was shifting towards exploiting higher levels of parallelism. The new transistors, predicted by Moore's law, went into replicating execution units that allow processors to handle multiple instruction streams at once [88, 95]. This was the beginning of *many-core* CPU designs. As a personal anecdote, from 2007 to 2010 I worked on an early many-core chip code-named Larrabee at Intel [102]—the technology we developed passed on to a series of x86 many-core processors found in supercomputers today.

As it turns out, however, single-threaded optimizations, which yielded tremendous performance benefits in the past, generate adverse effects in the context of multi-threading. These adverse effects transcend the scope of hardware: they are a source of complication for compiler and programming language designers and well as the application programmer. Next, we illustrate some of these effects on an example. First we look at instruction reordering, then we will look at the effects of memory access reordering.

Instruction reordering. The intuition for instruction reordering comes from the fact that commuting instructions can be swapped without changing a single-threaded program's behavior. We can derive alternate programs by swapping adjacent instructions that commute,[1]—although these new programs produce the same results as the original program, these programs may have different performance characteristics. In fact, rather than observing the original order of instructions in a program, performance can be improved by handling instructions according to the availability of inputs and the availability of execution units [98]. Instruction reordering is ordinarily performed statically, by compilers, and dynamically, by processors.

When it comes to multi-threaded programs, it would be natural for us to want to swap commuting instructions that occur within a thread. Programs derived this way preserve *single-threaded semantics*, meaning that each individual thread behaves "the same" despite these swaps. But what can we say about the behavior of the program as a whole? If individual threads behave the same, does the whole program necessarily behaves the same? Unfortunately, the answer is *no*. It is possible to preserve single thread semantics while modifying the overall behavior of a program.

Consider the program of Figure 1.1, where r_1 and r_2 are local variables, and x and y are shared. After inspection, we may come to the conclusion that, because of interleaving, the program can produce the following pairs of values for r_1 and r_2: $(r_1, r_2) \in \{(0,1), (1,0), (1,1)\}$. Note that it is not possible for the pair $(0,0)$ to be printed because if x is zero at the end, then y should have been set to one, and vice versa.

$T1$	$T2$
$x \ := 1$	$y \ := 1$
$r_1 := y$	$r_2 := x$
`print` r_1	`print` r_2

Figure 1.1: A simple multi-threaded example.

Now, based on commutativity, let us swap the first two instructions in $T1$ to obtain a new program, shown in Figure 1.2. The swap is not noticeable from the

$T1$	$T2$
$r_1 := y$	$y \ := 1$
$x \ := 1$	$r_2 := x$
`print` r_1	`print` r_2

Figure 1.2: A variation on the example of Figure 1.1 that is being used to illustrate the effects of instruction reordering.

[1] Here we are talking about commuting instructions within a thread, rather than commuting instructions between threads.

point of view of $T1$: before the swap, $T1$ could print 0 or 1 and, after the swap, $T1$ can still print 0 or 1. Although single-threaded semantics is preserved, the set of pairs that the modified program can produce now contains $(0,0)$ in addition to the values produced by the original program. In summary: individually, $T1$ and its variant behave the same, but as a whole, the original and the modified programs behave differently.

Memory access reordering. Memory access reordering, like instruction reordering, is a performance enhancing technique that exploits instruction level parallelism. The objective is to introduce buffers and caches in order to reduce the latency associated with retrieving values from and committing values to memory. Interestingly, memory access reordering can happen even if the instruction stream is executed in order.

Similar to instruction reordering/scheduling, memory optimizations can also adversely affect multi-threaded programs. Consider a processor with write-buffering. In a nutshell, write-buffers are place-holders for values on their way to be committed to memory. Such buffering allows a processor to continue executing even before the effects of its writes become visible to other processors. When it comes to the example in Figure 1.1, it is possible for $T1$ and $T2$ to deposit their writes into their respective write buffers, and to proceed executing before these writes trickle out to memory and become visible from the point of view of the other thread.[2] In such an execution environment, each thread would then observe the value of 0 when reading from memory, thus producing the pair $(0,0)$. In other words, memory access reordering, like instruction reordering, can introduce novel results to a multithreaded program—results that cannot be justified by interleaving alone.

The examples above show that the effects of instruction rescheduling, buffering and of other optimizations that target single-threaded performance are not *compositional.* While the behavior of each individual thread is preserved, when we compose these optimized threads into a multi-threaded program, the overall program behavior changes. Proverbially, *the whole is greater than the sum of the parts.*

Going back to the example at the very start of the chapter. We have learned that both instruction reordering and buffering can make it seem like the order of $T1$'s instructions has been reversed—reversed from $T2$'s perspective.[3] This alteration makes it possible for $T2$ to print an uninitialized value of z instead of printing 42. This behavior is counterintuitive, even on a tiny program. How can we, then, expect to make sense of large multithreaded applications?

[2] The behavior described is possible under the *processor consistency* memory model.

[3] In the case of memory reorderings, a total-store-order (TSO) architecture would still preserve the order of T1's writes. In order for the instructions in T1 to appear out-of-order from T2's perspective, we would need a more aggressive optimization, such as the partial-store-order (PSO) model.

1.2 Memory models

A *memory model* is "a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system" [1]. Simply put, a memory model dictates what values a read operation can return given previous read and write operations.[4] The memory model is a contract between the application programmer and the compiler/language designer. This contract is responsible for establishing the semantics of a high-level programming language constructs in a way that is intelligible to the application programmer. The contract imposes obligations on a language's compiler and runtime; namely, the obligation of shielding the application programmer from the intricacies of the underlying hardware (the effects of buffers and out-of-order execution) and shielding the programmer from intricacies in the compiler itself (*e.g.,* compiler optimizations).

Now, consider an execution environment where "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [57]. This environment is the execution model of interleaving: instructions execute in the order they appear within each thread and, it is only possible to shuffle which thread executes next. Such a forgiving memory model is called *sequentially consistent*. Under sequential consistency, we can disregard buffering and instruction reordering, thus making the question of "what does this program do," raised previously, much easier to answer.

Although simple to understand, sequential consistency precludes many of the widely adopted compiler and hardware optimizations in use today. For this reason, sequential consistency is not employed in actual designs and is, instead, a conceptual base-line—a departing point in the development of more flexible, more complex models. In fact, for years researchers and developers have explored and implemented different trade-offs in memory models by: starting with sequential consistency, relaxing certain ordering constraints, and studying the effect of these relaxations on performance and intelligibility. As an example, Adve and Gharachorloo [1] list the following design considerations:

- can a write to a variable be delayed after a read from another variable is issued by the same thread?
- can a write to a variable be delayed after a write to another variable is issued by the same thread?
- can a read from a variable be delayed after a read from or a write to another variable is issued by the same thread?

[4] In the memory model proposed in this thesis, a read is dependent only on previous writes. The semantics can, therefore, exhibit what is called *coRR behavior,* which means that consecutive reads can observe different values even when no new write events have taken place.

- can a thread read a write issued by another thread, even before this write has affected memory?

- can a thread read a write, even before this write has affected memory?

In practice, such questions can have counterintuitive consequences, thus yielding memory models that are difficult to understand and to describe—or even memory models that violate design goals [6, 83, 84, 14]. Finding a sweet spot in this design space, one that leads to performance without sacrificing intelligibility, is the holy grail.

Another way to understand memory models is to contrast them against other "features" or "components" of a programming environment. Several features or components of a programming environment are well encapsulated within their implementation. For example, it is easy to point to files that implement a language's parser, or to the files that implement threads and synchronization primitives. The fact that their implementation is encapsulated makes it easier for us to understand and verify these components' behavior. A memory model, however, is not encapsulated. Instead, a memory model's "implementation" (*i.e.,* its behavior) results from the collective choices made in different parts of the system; such as the compiler and its optimizations, the runtime and its handling of asynchronous computations, the relevant synchronization primitives, and sometimes even the underlying hardware. As such, a memory model is akin to an *emergent property* or *emergent behavior*, that is: a property of a collective that is not directly accounted for by the sum of the properties of its individual components. The concept is not often mentioned in the study of programming languages, computer architecture, or formal methods; yet, thinking of memory models as an emergent property may help point to new directions of research.

Given the insight of memory models as an emergent behavior, we are not surprised to know that, while channels in Go are mostly implemented in the `runtime/chan.go` source file, and can thus easily be inspected or tested, we do not find a "memory model" source file in the Go repository. This lack of reification makes memory models difficult to understand and verify. When it comes to verification, current focus has been placed on "testing oriented approaches," such as through the creation of *litmus tests*. Litmus tests are snippets of code that exemplify the behavior of a memory model, and that stress the distinction between models. Such approaches allow us, for example, to treat memory models as black-boxes, thus freeing us from specifying the underlying interconnection between the disparate components that influence a program's execution.

Although there is still little consensus on what the right memory model should be, over the years, we have converged around two principles: (1) the undesirability of *out-of-thin-air* behavior and (2) the desire for the execution of "well-behaved" programs to appear sequentially consistent. We discuss these points next.

Out-of-thin-air. Out-of-thin-air (OOTA) are counterintuitive results that emerge from circular reasoning. One challenge in specifying a programming language's memory model is forbidding OOTA behavior without significantly jeopardizing performance.

For example, given the program below, there exists an execution which results in $r_1 = r_2 = 42$. This execution springs from a common feature called load-buffering: the load of shared variable y in $T2$ is buffered, thereby taking effect after the write to x. Buffering causes the instructions in $T2$ to be executed *out-of-order* from the point of view of $T1$—this reordering can be used to generate alternate executions that may be more *performant* than the original program.

$$\begin{array}{ll@{\qquad\qquad}ll} & T1 & & T2 \\ r_1 & := x & r_2 & := y \\ y & := r_1 & x & := 42 \end{array}$$

Now, let us change to the program above so that, as opposed to writing 42 to x, we write the value of r_2:

$$\begin{array}{ll@{\qquad\qquad}ll} & T1 & & T2 \\ r_1 & := x & r_2 & := y \\ y & := r_1 & x & := r_2 \end{array}$$

Given this modification, we would like to forbid the possibility of producing $r_1 = r_2 = 42$. The value 42 (or any value other than 0) is, in this case, considered to come "out-of-thin-air." Such a value cannot be justified from the program text without the use of circular reasoning: if y contains 42, then 42 is written to x in $T2$, r_1 gets set with the value of 42 in $T1$ and so is y, thus justifying the original assumption of y containing 42.

Hardware forbids such causality cycles. However, compiler optimizations can compile away dependencies that would otherwise be observed at runtime. The interaction between compiler optimizations and hardware relaxations makes it difficult to precisely define and disallow out-of-thin-air results from a language memory-model specification.

DRF-SC guarantee. When defining a memory model, there exists a tension between (1) its flexibility in accounting for compiler and hardware optimizations (which is related to how much a model relaxes ordering constrains), and (2) the memory model's intelligibility from the application programmers point of view, which accounts for, among other things, the absence of out-of-thin-air behavior. Regardless of how a memory model balances this tension, the model should abide by the following principle: if a program is "well-behaved," then its execution should appear sequentially consistent.

When it comes to multi-threaded programs, the notion of well-behavior is captured by the concept of *data races*. Data races—when two threads perform conflicting memory accesses without synchronizing with one another—can lead to counterintuitive behavior that is hard reproduce and debug. Note that two memory accesses *conflict* if they involve the same memory location and at least one of the accesses is a *write*. It is possible to design memory models such that, if a program is Data-Race Free (DRF) then its execution behaves Sequentially Consistently (SC). Memory models that abide by this property are called DRF-SC.

The DRF-SC property is tremendously useful. For one, the property has a "plug-and-play" effect: a data-race free program is sure have the same set of behaviors under any memory model that supports the DRF-SC guarantee. But most importantly, DRF-SC allows programmers to think in terms of sequential consistency, provided their programs are data-race free. This effect is desirable because reasoning under sequential consistency is much simpler than under relaxed memory.

1.3 The goals of this thesis

Using a powerful and well-known formalism called *operational semantics,* this thesis presents a memory model for an interesting and relevant programming paradigm—that of message passing systems. While memory models have extensively been studied in the context of locks, fences, and barriers (constructs which can easily be mapped down to ordinary hardware), message passing systems have arguably received lesser attention. In fact, common misconceptions about message passing systems is that they do not harbor data races. Indeed, in the Scandinavian school, objects are agents in a concurrent system, objects have internal memory, and methods are a communication mechanism on par with message passing. In this programming style, since agents do not share memory, the concept of *data* races is irrelevant.[5] Yet, with the advent of the Go programming language, message passing as a means of synchronization of shared memory systems has gained traction. With the backing of Google, Go has become prominent in the area of cloud computing, where it is used in the development of container management systems such as Kubernetes and Docker. Being the underpinning of vast amounts of virtual infrastructure, Go's concurrency model is well worth of theoretical investigation.

Go is influenced by the work of Tony Hoare on Communicating Sequential Processes (CSP) [46], and the language's memory model is rooted in Leslie Lamport's the concept of *happens-before* and his work on message passing and distributed systems [56, 57]. Although anchored on stable theoretical grounds, there are important differences that set Go apart. For example, in the referenced works, Lamport was

[5] Despite the absence of shared memory, agents can still compete when sending/receiving messages over a channel. These types of races (*i.e.,* channel races) have been widely studied, and their absence is related to determinism rather than data race freedom [24, 25, 73, 97].

studying distributed systems, while Go is a language for concurrency. Both concurrency and distribution speak of independent agents cooperating on a common task. For that to happen, agents need to coordinate, to synchronize. Lamport defines the *happens-before relation* as a way to talk about the partial order of events occurring in a distributed systems where agents interact solely by exchanging messages. Although distribution and concurrency share many goals and techniques, these research areas are not the same. In a concurrent system, we assume that the agents are under a single environment. In Go, for example, all agents (goroutines) are under the umbrella of the Go runtime. This overarching environment allows us to assume that no messages are lost during transmission. In distributed system, however, there is no such point of authority—at least not without making strong assumptions about the system. As a consequence, in general, network delays are indistinguishable from node crashes. In a distributed system, communication is no longer perfect, and we are forced to deal with this fact.

The presence of a central authority (*e.g.,* a language runtime) allows Go to specify the formation of a *happens-before relation* between two agents that do not directly exchange messages! This additional happens-before rule is not present in the works of Lamport. This additional rule links a receiver and a previous sender on a channel, thus allowing Go channels to be used as locks. In this thesis, we explore in detail the effect of this additional rule from both theory and practice. Ultimately, by performing a methodical exposition starting from first principles, we were able to relate a formal language semantics to the implementation of channels in the Go runtime. By doing so, we show that the Go data-race detector is at odds with the Go memory model specification. This mismatch springs from the detector (mis)interpreting channels from the point of view of locks—as opposed to channels as first-class language constructs. In cooperation with the Go maintainers at Google, we resolved the issue with a patch that has been merged into the Go sources.

This thesis also contributes to the theory behind memory model specification. The contemporary research on memory models and data races also does not treat channels as a first-class construct. Instead, channels are higher level constructs built from underlying primitives. We depart from this trend and follow Go's approach of defining a memory model with message passing as the *de facto* synchronization mechanism. We argue that our treatment of message passing has unveiled gaps in the contemporary literature. For example, on a theoretical level, we show that a trace-based definition of data races *à la* Mazurkiewicz [68] does not sit well with a happens-before based definition of data race; although the two interpretations are sometimes conflated in the literature.

Besides elevating message passing, we set off to define a memory model *operationally*, while most do it axiomatically. Our intuition is that, like hardware, an operational semantics can be made to naturally preclude out-of-thin-air behavior.

In summary, we set out to:

(Goal A) Propose and evaluate a memory model for message passing systems using an operational semantics as formalism.

(Goal B) Investigate how relaxed the model can be while keeping out-of-thin-air behavior at bay and while supporting the DRF-SC guarantee—which says that Data-Race Free executions are Sequentially Consistent.

(Goal C) Relate the theoretical model to its "material" counterpart and source of inspiration: the Go programming language.

1.4 Thesis outline

Chapter 2 presents a mathematical description of a subset of the Go programming language.[6] The chapter introduces a weak memory model inspired by the Go specification. Different from many previous memory model formalizations, ours is given in an *operational semantics*. We illustrate the proposed semantics's behavior on a set of *litmus tests*: snippets that highlight features of a memory model. To that end, we highlight similarities and differences between our formalism and that of a representative axiomatic semantics.

Being operational means our model, like the Go implementation, is executable. We implemented our operational semantics in $\mathbb{K}$ [51, 87], an executable semantics framework based on rewriting logic. On a practical side, this implementation helped us work through rough corners in our understanding. We believe the code can also help the interested reader assimilate the reduction rules in our formalism, and to explore alternatives by modifying the sources available online [29]. In Chapter 2, we give the reader a sense of how our implementation in $\mathbb{K}$ follows from the operational semantics.

Also in Chapter 2, we prove that our proposed model abides by the DRF-SC guarantee. The proof uses a simulation relation [70] to connect (1) a program running under a sequentially consistent memory model to (2) a run of the program under our relaxed memory model. Given the desirability of the DRF-SC guarantee, the proof is consequential. As a side note, the lemmas that support the main proof have a connection to distributed systems—see, for example, the consensus lemmas 6 and 8 on page 40.

Although a simplifying property, the DRF-SC guarantee puts a tremendous burden on developers: the burden of programming without data races. We tackle synchronization issues in message passing programs in Chapter 3. While synchronization has been studied extensively in the context of locks, when it comes to communication over channels, the topic has received arguably less attention. This thesis presents a fresh perspective on data-race detection for message passing systems, thus plugging a hole in the contemporary literature.[7]

[6] Chapter 2 is based on the conference paper [33] and its corresponding extension to the journal article [34]

[7] Chapter 3 is based on the journal article [35].

In Chapter 3, we introduce a data-race detector based on what we call *happens-before sets*. We draw comparisons between our approach and that of modern data-race detectors based on *vector clocks*. One outcome from this work is the observation that stale information can accumulate during the execution of a data-race detector. Given that detectors incur large runtime overheads, our observation is a first step in addressing the memory footprint associated with data-race detection. In Chapter 3, we formalize the notion of *stale vector-clock entries* and conjecture the advantages of garbage collection in the context of data-race detection. Finally, in Chapter 3 we expose difficulties in establishing a connection between a *happens-before* definition of data races and a trace based definition *a la* Mazurkiewicz.

While executable like the Go implementation, the terseness of mathematical notation makes our formalism compact and thus more directly relatable to the Go memory model specification [40]. This ability to bridge high (specification) and low levels (source code) opens the door for the results introduced in Chapter 4. Armed with the memory model described in Chapter 2 and a formalization of data race in the context of message passing (Chapter 3), we turn our attention to Go source code—particularly, to how the implementation of channels are connected to the underlying Go data-race detector.[8] With that, we were able to unveil a mismatch between the Go data-race detector and the Go memory model specification. This mismatch, which evaded skilled developers for six years, prevented the data-race detector from flagging certain synchronization bugs in Go programs. Resolving this mismatch translates to improved correctness of software developed using the Go programming language. Our experience bridging academic research and industrial application can be summarized with the following maxims:

- *models do not have to be right, they have to be useful.*

- *mind the gap* between specification and implementation.

- *bad news is good news:* leveraging the complementing strengths (and weaknesses) in industrial development versus academic research.

In Chapter 4, we distill the above maxims and describe our interaction with the open-source Go community.

By design, the memory model we describe in Chapter 2 does not support a relaxation called *load buffering*. The absence of load buffering is a simplifying assumption that allowed us to build intuition and to work through a potentially simpler proof of the DRF-SC guarantee. Having these results as stepping stone, we turn our attention to load buffering. Incorporating load buffering would bring our formalization up to par with the Go memory model specification. In Chapter 5, we explore how to further relax our initial memory model.[9] Our initial intuition was that an operational

[8] Chapter 4 is based on the conference paper [30].

[9] Chapter 5 is based on [32] and on unpublished work.

semantics, like hardware, can be made to avoid out-of-thin-air (OOTA) and still be relaxed. We started by studying the effects of read-delay on a language with just reads and writes. In this simplified scenario, circularity also comes into play, but: as opposed to leading to OOTA behavior (like in axiomatic semantics), circularity led to dead- and live-locks—even for programs that terminate under sequential consistency. Next, we explored mixing conditional branching and delayed reads. In this more realistic scenario, our operational semantics also suffered from OOTA behavior. Resolving these issues remain an open area of research, as well as attempting to prove the DRF-SC guarantee for the operational version a relaxed memory model with read delays.

We conclude, in Chapter 6, with a discussion potential extensions to the work towards *(Sec. 6.1)* data-race detection, *(Sec. 6.2)* model checking and *predictive* data-race detection, and *(Sec. 6.3)* distribution.

2 Operational Semantics of a Weak Memory Model with Channel Synchronization

There exists a multitude of weak memory models supporting various types of relaxations and synchronization primitives. On one hand, such models must be lax enough to account for hardware and compiler optimizations; on the other, the more lax the model, the harder it is to understand and program for. Though the right balance is up for debate, a memory model should provide what is known as the *DRF-SC guarantee*, meaning that data-race free programs behave in a sequentially consistent manner.

We present a weak memory model for a calculus inspired by the Go programming language. Thus, different from previous approaches, we focus on buffered channel communication as the sole synchronization primitive. Our formalization is operational, which allows us to prove the DRF-SC guarantee using a standard simulation technique. Contrasting against an axiomatic semantics, where the notion of a program is abstracted away as a graph with memory events as nodes, we believe our operational semantics and simulation proof can be clearer and easier to understand. Finally, we provide a concrete implementation in $\mathbb{K}$, a rewrite-based executable semantic framework, and derive an interpreter for the proposed language.

2.1 Introduction

A *memory model* dictates which values may be observed when reading from memory, thereby affecting how concurrent processes communicate through shared memory. One of the simplest memory models, called *sequentially consistent*, stipulates that operations must appear to execute one at a time and in program order [57]. SC was one of the first formalizations to be proposed and, to this day, constitutes a baseline for well-behaved memory. However, for efficiency reasons, modern hardware architectures do not guarantee sequential consistency. SC is also considered much too strong to serve as the underlying memory semantics of programming languages; the reason is that sequential consistency prevents many established compiler optimizations and robs from the compiler writer the chance to exploit the underlying hardware for efficient parallel execution. The research community, however, has not been able to agree on exactly what a proper memory model should offer. Consequently, a bewildering array of *weak* or *relaxed memory models* have been proposed, investigated, and implemented. Different taxonomies and catalogs of so-called *litmus tests*, which highlight specific aspects of memory models, have also been researched [1].

Memory models are often defined axiomatically, meaning via a set of rules that constrain the order in which memory events are allowed to occur. The *candidate execution* approach falls in this category [9]. The problem with this approach, however, is that either the model excludes too much "good" behavior (*i.e.,* behavior that is deemed desirable) or it fails to filter out some "bad" behavior [9]. *Out-of-thin-air* is a common class of undesired behavior that often plagues weak memory specifications. Out-of-thin-air are results that can be justified by the model via circular reasoning but that do not appear in the actual executions of a program [16]. In light of these difficulties and despite many attempts, there are no well-accepted comprehensive specification of the C++11 [11, 15] and Java memory models [6, 64, 83].

More recently, one fundamental principle of relaxed memory has emerged: no matter how much relaxation is permitted by a memory model, if a program is *data-race free* or *properly synchronized*, then the program must behave in a sequentially consistent manner [2, 64]. This is known as the *DRF-SC* guarantee. DRF-SC allows for a *write-it-once run-it-anywhere guarantee*, meaning that data-race-free code behaves equally across memory models that provide the guarantee, regardless of which relaxations are supported in the underlying model.

We present an *operational* semantics for a weak memory. Similar to Boudol and Petri [17], we favor an operational semantics because it allows us to prove the DRF-SC guarantee using a standard simulation technique. The lemmas we build up in the process of constructing the proof highlight meaningful invariants and give insight into the workings of the memory model. We think that our formalism leads to an easier to understand proof of the DRF-SC guarantee when compared to axiomatic semantics. Our belief is based on the following observation: the notion of program is preserved in an operational semantics, while in axiomatic semantics, a program is often abstracted into a graph with nodes as memory events.

Our calculus is inspired by the Go programming language: similar to Go, our model focuses on channel communication as the main synchronization primitive. Go's memory model [40], however, is described, albeit succinctly and precisely, in prose. We provide a formal semantics instead.

The main contributions of our work therefore are:

- Few studies focus on channel communication as synchronization primitive for weak memory. We give an operational theory for a weak memory with bounded channel communication.

- Using a standard conditional simulation proof, we prove that the proposed memory upholds the *sequential consistency guarantee for data-race free* programs.

- We implement the operational semantics in the $\mathbb{K}$ executable semantics framework [51, 87] and make the source code publicly available [29]. Here, we provide a detailed description of the $\mathbb{K}$ implementation and walk through a rewriting rule to give the reader a sense of how the implementation follows from the operational semantics.

- We add a discussion section illustrating the proposed semantics's behavior on litmus tests. Here we revisit concepts from the axiomatic semantics of memory models in order to highlight similarities and differences between our semantics and a representative axiomatic semantics.

The remaining of the chapter is organized as follows. Section 2.2 presents background information directly related to the formalization of our memory model. Sections 2.3, 2.4 and 2.5 provide the syntax and the semantics of the calculus with shared relaxed memory and channel communication. Section 2.6 establishes the DRF-SC guarantee. This is done via a *simulation* proof that relates a standard "strong" semantics (which guarantees sequential consistency) to the weak semantics. The proof makes use of an auxiliary semantics detailed in the appendix. Section 2.7 discusses the implementation of the strong and the weak semantics in $\mathbb{K}$. With the goal of contrasting and positioning our work at a wider context, Section 2.8 illustrates the behavior of the proposed memory model on litmus tests. Section 2.9 addresses the model's limitations. Sections 2.10 and 2.11 conclude with related and future work.

2.2 Background

In this section we provide background on the proposed memory model. Its semantics and properties will be covered more formally in the later sections.

Go's memory model The Go language [39, 28] recently gained traction in networking applications, web servers, distributed software and the like. It prominently features goroutines, which are asynchronous functions resembling lightweight threads, and buffered channel communication in the tradition of CSP [46] (resp. the π-calculus [71]) or Occam [50]. While encouraging message passing as the prime mechanism for communication and synchronization, threads can still exchange data via shared variables. Consequently, Go's specification includes a memory model which spells out, in precise but informal English, the few rules governing memory interaction at the language level [40].

Concerning synchronization primitives, the model covers goroutine creation and destruction, channel communication, locks, and the `once`-statement. Our semantics will concentrate on thread creation and channel communication because lock-handling and the `once` statement are *not* language primitives but part of the `sync`-library. Thread destruction, *i.e.,* termination, comes with *no* guarantees concerning visibility: it involves no synchronization and thus the semantics does not treat thread termination in any special way. In that sense, our semantics treats all of the *primitives* covered by Go's memory model specification. As will become clear in the next sections, our semantics does not, however, relax read events. Therefore, our memory model is stronger than Go's. On the plus side, the lack of relaxed read events prevents a class of undesirable behavior called *out-of-thin-air* [16]. On the negative, this absence comes at the expense of some forms of compiler optimizations.

Languages like Java and C++ go to great lengths not only to offer the DRF-SC guarantee, but beyond that, strive to clarify the non-SC behavior of *ill*-synchronized programs. It is far from trivial, however, to attribute a "reasonable" semantics to racy programs. In particular, it is hard to rule out the so called *out-of-thin-air* behavior [16] without inadvertently restricting important memory relaxations. Intuitively, one can think of out-of-thin-air as a class of behavior that can be justified via some sort of circular reasoning. However, according to Pichon-Pharabod and Sewell [79], there is no exact, generally accepted definition for out-of-thin-air behavior. Doubts have also been cast upon a general style of defining weak memory models. For instance, Batty et al. [9] point out limitations of the so-called *candidate of execution* way of defining weak memory models, whereby first possible executions are defined by way of ordering constraints, where afterwards, illegal ones are filtered out. In such formalizations, the distinction between "good," *i.e.,* expected behavior, and "bad," *i.e.,* outlawed behavior, is usually illustrated by a list of examples or litmus tests. The problem is that there exist different programs in the C/C++11-semantics with the *same* candidate executions, yet their resulting execution is deemed acceptable for some programs and unacceptable for others [9]. In contrast, Go's memory model is rather "laid back." Its specification [40] does not even mention "out-of-thin-air" behavior. In that sense, Go has a *catch-fire semantics*, meaning that the behavior of racy programs is not defined.

Happens-before relation and observability Like Java's [64, 83], C++11's [11, 15], and many other memory models, ours centers around the definition of a *happens-before* relation. The concept dates back to 1978 [56] and was introduced in a pure *message-passing* setting, *i.e.,* without shared variables.[1] The relation is a technical vehicle for defining the semantics of memory models.

It is important to note that just because an instruction or event is in a happens-before relation with a second one, it does not necessarily mean that the first instruction *actually* "happens" before the second in the operational semantics. Consider the sequence of assignments $x := 1; y := 2$ as an example. The first assignment "happens-before" the second as they are in program order, but it does not mean the first instruction is actually "done" before the second,[2] and especially, it does not mean that the effect of the two writes become observable in the given order. For

[1] The relation was called happened-before in the original paper.

[2] Assuming that x and y are not aliases in the sense that they refer to the same or "overlapping" memory locations.

example, a compiler might choose to change the order of the two instructions. Alternatively, a processor may rearrange memory instructions so that their effect may not be visible in program order. Conversely, the fact that two events happen to occur one after the other in a particular schedule does not imply that they are in happens-before relationship, as the observed order may have been coincidental.

To avoid confusion between the technical happens-before relation and our understanding of what happens when the programs runs, we speak of event e_1 "happens-before" e_2 in reference to the technical definition (abbreviated $e_1 \rightarrow_{\mathsf{hb}} e_2$) as opposed to its natural language interpretation. Also, when speaking about steps and events in the operational semantics, we avoid talking about something happening before something else, and rather say that a step or transition "occurs" in a particular order.

The happens-before relation regulates observability, and it does so very liberally. It allows a read r from a shared variable to *possibly observe* a particular write w to said variable *unless* one of the following two conditions hold:

$$r \rightarrow_{\mathsf{hb}} w \qquad \text{or} \tag{2.1}$$

$$w \rightarrow_{\mathsf{hb}} w' \rightarrow_{\mathsf{hb}} r \qquad \text{for some other write } w' \text{ to the same variable.} \tag{2.2}$$

There is no memory hierarchy through which write events propagate; there are no buffers or caches that need to be flushed. Visibility of a write event is enabled globally and immediately. The only writes that are not visible are writes that happen-after a read as detailed in condition (2.1), and writes w that have been supplanted or *shadowed* by a more recent write w' as detailed in condition (2.2). We call the knowledge of a write event as *positive information* and the knowledge that a write has been shadowed as *negative information*.

Although knowledge of write events (*i.e.,* positive information) is available globally and immediately, we will see next that knowledge of shadowed events, or negative information, is local. The exchange of this negative information is what allows for synchronization. For the sake of discussion, let us concentrate on the following two constituents for the happens-before relation: 1) *program order* and 2) the order stemming from channel communication.[3] According to the Go memory model [40], we have the following constraints related to a channel c with capacity k:

$$\text{A send on } c \text{ happens-before the corresponding receive from } c \text{ completes.} \tag{2.3}$$

$$\text{The } i^{th} \text{ receive from } c \text{ happens-before the } (i+k)^{th} \text{ send on } c. \tag{2.4}$$

To illustrate how the happens-before and channel communication can be used when reasoning about program behavior, consider the following example.

Example 1 (Synchronization via channel communication)**.** *Listing 2.1 shows the spawning and asynchronous execution of a setup function, which then runs concurrently with* `main`*. The thread executing* `setup` *writes to the shared variable* `a`*,*

[3] There are additional conditions in connection with channel creation and thread creation, the latter basically a generalization of program order; we ignore it in the discussion here.

Listing (2.1)
Failed sync. [40]

```
var a string
var done bool

func setup() {
  a = "hello world"
  done = true
}

func main() {
  go setup()
  for !done {} // try wait


  print(a)
}
```

Listing (2.2)
Channel sync. [40]

```
var a string
var c = make(chan int, 2)

func setup() {
  a = "hello world"
  c <- 0 // send
}

func main() {
  go setup()
  <-c     // receive


  print(a)
}
```

Listing (2.3)
Sync. via channel capacity

```
var a string
var c = make(chan int, 2)

func setup() {
  a = "hello world"
  <-c // receive
}

func main() {
  go setup()
  c <- 1    // send
  c <- 2    // send
  c <- 3    // send
  print(a)
}
```

Figure 2.1: Synchronization via channel communication

thereby shadowing its initial value from the perspective of `setup`*'s. This means, after being overwritten by the* `hello world` *string, the variable's initial value is no longer accessible for that particular thread. The shadowing here accounts for condition 2.2. In the setup thread, the write to variable* `a` *happens-before the write to* `done`*, as they are in program order. For the same reason, the read(s) of* `done` *happen-before the read of* `a` *in the main thread. Without synchronization, the variable accesses are ordered locally* per thread *but not across threads. Since neither condition (2.1) or (2.2) applies, the main procedure may or may not observe writes performed by* `setup`*. Thus, it is possible for main to observe the initial value of* `a` *as well as its updated value. Such ambiguity in observation is what allows the writes to* `a` *and* `done` *performed by* `setup` *to potentially appear out-of-order from the main thread's perspective. This example illustrates how shadow information (i.e., negative information) is thread-local: only* `setup` *is in a happens-before relation with the write of* `hello world` *to* `a`*, and only* `setup` *is* unable *to observe* `0`.

Replacing the use of `done` *by channel communication properly synchronizes the two functions (cf. Listing 2.2). As the receive happens-after the send, an order is established between events belonging to the two threads. One can think of the main thread as receiving not only a value but also the knowledge that the write event to* `a` *in* `setup` *has taken place. With condition (2.3), channels implicitly communicate the happens-before relation from the sender to the receiver. Then, with condition (2.2), we can conclude that once the main thread receives a message from* `setup`*, the initial value of* `a` *is no longer observable from* `main`*'s perspective.*

The previous example shows how condition (2.3) can be used to synchronize a program; namely, using the fact that a message carries not only a value but also happens-before information from a sender to its corresponding receiver. There exists yet another form of synchronization, formulated in condition (2.4), which hinges on a channel's bounded capacity. This synchronization comes from the fact that a sender is only allowed to deposit a message into a bounded channel when the channel is not full. The boundedness of a channel, therefore, relates a sender to some previous receiver who, by reading from the channel, created an empty slot onto which the sender can deposit its message. Happens-before information, in this case, flows *backwards*: from some *receiver* to a later *sender.*

Example 2 (Synchronization via channel capacity)**.** *Listing 2.3 shows a modification to the synchronization example where, as opposed to sending a message when the shared variable is modified, the* `setup` *thread* receives *a message. Note that information flows backwards: the fact that the message is received implicitly communicates information back to the message's sender. The sender, in this case* `main`*, uses the limited channel capacity to its advantage: it sends three messages on a channel of capacity two; the third message can only be successfully deposited onto the channel once the* `setup` *thread receives from the channel (until then the third send will block). Therefore, the main thread can infer that, when the third message is sent, the receive at* `setup` *has completed, which in turn means that the shared variable has been initialized.*

Note that for *synchronous* channels, which have capacity zero, conditions (2.3) and (2.4) degenerate: the send and receiving threads participate in the rendezvous and symmetrically exchange their happens-before information.

In summary, the operational semantics captures the following principles:

Immediate positive information: a *write* is globally observable instantaneously.

Delayed negative information: in contrast, negative information overwriting previously observable *writes* is *not* immediately effective. Referring back to the example of Figure 2.1, the fact that `setup` has overwritten the initial value of variable `a` is not immediately available to other threads. Instead, the information is spread via message passing in the following way:

Causality: information regarding condition (2.3) travels with data through channels.

Channel capacity: *backward channels* are used to account for condition (2.4).

Local view: Each thread maintains a local view on the happens-before relationship of past write events, *i.e.,* which events are unobservable. Thus, the semantics does not offer multi-copy atomicity [21].

2.3 Abstract syntax

The abstract syntax of the calculus is given in Figure 2.2. *Values* v can be of two forms: r is used to denote the value of local variables or registers, while n in used to denote references or names in general and, in specific, p for processes or goroutines, m for memory events, and c for channel names. We do not explicitly list values such as the unit value, booleans, integers, etc. We also omit compound local expressions like $r_1 + r_2$.

Shared variables are denoted by x, z etc, $\texttt{load}\, z$ represents reading the shared variable z into the thread, and $z := v$ denotes writing to z. Unlike in the concrete Go surface syntax, our chosen syntax for reading global variables makes the shared memory access explicit. Specifically, global variables z, unlike local variables r, are not expressions on their own. They can be used only in connection with loading from or storing to shared memory. Expressions like $x \leftarrow \texttt{load}\, z$ or $x \leftarrow z$ are disallowed. Therefore, the languages obeys a form of at-most-once restriction [5], where each elementary expression contains at most one memory access.

References are dynamically created and are, therefore, part of the *run-time* syntax. Run-time syntax is highlighted in the grammar with an underline as in $\underline{n}$. A new channel is created by $\texttt{make}\ (\texttt{chan}\ T, v)$, where T represents the type of values carried by the channel and v a non-negative integer specifying the channel's capacity. Sending a value over a channel and receiving a value as input from a channel are written respectively as $v_1 \leftarrow v_2$ and $\leftarrow v$. After the operation `close`, no further values can be sent on the specified channel. Attempting to send values on a closed channel leads to a panic.

Starting a new asynchronous activity, called goroutine in Go, is done using the `go`-keyword. In Go, the `go`-statement is applied to function calls only. We omit function calls, asynchronous or otherwise, since they are orthogonal to the memory model's formalization. See Steffen [94] for an operational semantics dealing with goroutines and closures in a purely functional setting, that is, without shared memory.

The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches which are called communication clauses by the Go specification [39]. These branches act as guarded threads. General expressions in Go can serve as guards. Our calculus, however, imposes the restriction that only communication statements (*i.e.,* channel sending and receiving) and the `default`-keyword can serve as guards. This restriction is in line with the A-normal form representation [89] and does not impose any actual reduction in expressivity. Both in Go and in our formalization, at most one branch is guarded by `default` in each select-statement. The same channel can be mentioned in more than one guard. "Mixed choices" [77, 78] are also allowed, meaning that sending- and receiving-guards can both be used in the same select-statement. We use `stop` as syntactic sugar for the empty select statement; it represents a permanently blocked thread, see Figure 2.3. The `stop`-thread is also the only way to syntactically "terminate" a thread, meaning that it is the only element of t without syntactic sub-terms.

The `let`-construct $\texttt{let}\ r = e\ \texttt{in}\ t$ combines sequential composition and the use of scopes for local variables r: after evaluating e, the rest t is evaluated where the resulting value of e is handed over using r. The let-construct is seen as a binder for variable r in t. When r does not occur free in t, `let` then boils down to *sequential composition* and, therefore, is replaced by a semicolon; see Figure 2.3.

$$\begin{array}{lcll}
v & ::= & r \mid \underline{n} & \text{values} \\
e & ::= & t \mid v \mid \mathtt{load}\, z \mid z := v \mid \mathtt{if}\, v\, \mathtt{then}\, t\, \mathtt{else}\, t \mid \mathtt{go}\, t & \text{expressions} \\
 & \mid & \mathtt{make}\, (\mathtt{chan}\, T, v) \mid \leftarrow v \mid v \leftarrow v \mid \mathtt{close}\, v & \\
g & ::= & v \leftarrow v \mid \leftarrow v \mid \mathtt{default} & \text{guards} \\
t & ::= & \mathtt{let}\, r = e\, \mathtt{in}\, t \mid \sum_i \mathtt{let}\, r_i = g_i\, \mathtt{in}\, t_i & \text{threads}
\end{array}$$

Figure 2.2: Abstract syntax

$$\begin{array}{rcll}
e;\, t & ::= & \mathtt{let}\, r = e\, \mathtt{in}\, t & \text{when } r \notin fn(t) \\
\mathtt{stop} & ::= & \sum_0 &
\end{array}$$

Figure 2.3: Syntactic sugar

2.4 Strong operational semantics

Before introducing the main contribution, we discuss a sequentially consistent semantics for the calculus. It is a stripped down version of the weak one and serves as a stepping stone into the relaxed memory model presented in Section 2.5. Secondly, the strong semantics will later be used in the DRF-SC proof of Section 2.6, where we establish that the sequentially semantics conditionally simulates the weak one. We start by fixing the run-time configurations of a program before giving the operational rules in Sections 2.4.2 and 2.4.3.

2.4.1 Configurations

Let X be a set of shared variables such as x, z... A run-time configuration is given by the following syntax:

$$S ::= p\langle t\rangle \mid (\!| z{:=}v |\!) \mid \bullet \mid S \parallel S \mid c[q] \mid \nu n\, S\,. \tag{2.5}$$

where p, m, c, and n are drawn from an infinite set of names or identifiers N. As mentioned earlier, for readability, we will typically use p, $p'_1 \ldots$ for goroutines or processes, $c, c_1, \ldots$ for channels, and $n, n_1, \ldots$ for names in general (where the object being name is of no particular relevance).

Configurations, therefore, consist of the parallel composition of goroutines $p\langle t\rangle$ where t is the code to be executed, write events $(\!|z{:=}v|\!)$ where variable z takes value v, and channels $c[q]$ where q is a queue. The symbols $\bullet$ stands for the empty configuration. The ν-binder, known from the π-calculus, indicates dynamic scoping [71].

The strongly consistent semantics is a standard interleaving semantics, which means that reads and writes immediately interact with a shared global state. Later we will see that, in the case of the weak semantics, memory events are labeled and goroutines hold thread-local information.

There is only one goroutine, which we refer to as "main," at the beginning of execution. Also, no channels have been created yet and each shared variable in the program is initialized to a known value. Thus, an initial configuration takes the following form.

Definition 1 (Initial configuration)**.** *Initially, a strong configuration is of the form* $p\langle t_0\rangle \parallel (\!|z_0{:=}v_1|\!) \parallel \ldots \parallel (\!|z_k{:=}v_k|\!)$*, where* $z_0, \ldots z_k$ *are all shared variables of the program and* t_0 *contains no run-time syntax.*

The initial configuration evolves according to operational semantic rules. The rules are given in several stages. We start with local steps, that is, steps not involving shared variables.

2.4.2 Local steps

The reduction steps are given modulo structural congruence $\equiv$ on configurations. The congruence rules are standard and given in Figure 2.4. Besides specifying parallel composition as a binary operator of an Abelian monoid and with $\bullet$ as neutral element, there are two additional rules dealing with the ν-binders. They are likewise standard and correspond to the treatment of name creation in the π-calculus [71].

$$\begin{array}{rcll}
P_1 \parallel P_2 & \equiv & P_2 \parallel P_1 & \\
(P_1 \parallel P_2) \parallel P_3 & \equiv & P_1 \parallel (P_2 \parallel P_3) & \\
\bullet \parallel P & \equiv & P & \\
P_1 \parallel \nu n\, P_2 & \equiv & \nu n\, (P_1 \parallel P_2) & \text{if } n \notin \mathit{fn}(P_1) \\
\nu n_1\, \nu n_2\, P & \equiv & \nu n_2\, \nu n_1\, P &
\end{array}$$

Figure 2.4: Structural congruence

Reduction modulo congruence and other "structural" rules are given in Figure 2.5. There are two basic reduction steps $\rightsquigarrow$ and $\rightarrow$. Local steps $\rightsquigarrow$ reduce a thread t without touching shared variables; see Figure 2.6. Global steps are given in the next section.

$$\frac{P \equiv P_1 \qquad P_1 \to P_2 \qquad P_2 \equiv P'}{P \to P'} \qquad \frac{P_1 \to P_1'}{P_1 \parallel P_2 \to P_1' \parallel P_2} \qquad \frac{P \to P'}{\nu n\, P \to \nu n\, P'}$$

Figure 2.5: Congruence and reduction

$\texttt{let}\, x = v \,\texttt{in}\, t \rightsquigarrow t[v/x] \quad$ R-RED

$\texttt{let}\, x_1 = (\texttt{let}\, x_2 = e \,\texttt{in}\, t_1) \,\texttt{in}\, t_2 \rightsquigarrow \texttt{let}\, x_2 = e \,\texttt{in}\, (\texttt{let}\, x_1 = t_1 \,\texttt{in}\, t_2) \quad$ R-LET

$\texttt{if true then}\, t_1 \,\texttt{else}\, t_2 \rightsquigarrow t_1 \quad$ R-COND$_1$

$\texttt{if false then}\, t_1 \,\texttt{else}\, t_2 \rightsquigarrow t_2 \quad$ R-COND$_2$

Figure 2.6: Operational semantics: Local steps

2.4.3 Global steps

To differentiate the strong global steps introduced here from the weak global ones introduce in Section 2.5, we use a subscript s in the strong semantics rules. For example R-WRITE$_s$ versus R-WRITE.

2.4.3.1 Reads and writes to shared memory

As mentioned previously, in the sequentially consistent semantics, reads and writes to memory take effect immediately. Writes simply update the value associated its corresponding variable, and reads obtained that value; see Figure 2.7. Since the initial configuration has one write event per shared variable, and since write events are not created or destroyed by any of the reduction steps, the following is an invariant of the semantics.

$$\frac{}{p\langle z := v; t\rangle \parallel (\!|z{:=}v'|\!) \to p\langle t\rangle \parallel (\!|z{:=}v|\!)} \text{ R-WRITE}_s$$

$$\frac{}{p\langle \texttt{let}\, r = \texttt{load}\, z \,\texttt{in}\, t\rangle \parallel (\!|z{:=}v|\!) \to p\langle \texttt{let}\, r = \ v \,\texttt{in}\, t\rangle \parallel (\!|z{:=}v|\!)} \text{ R-READ}_s$$

Figure 2.7: Strong operational semantics: read and write steps

Definition 2 (Well-formed strong configuration)**.** *An strong configuration S is* well-formed *if, for every variable $z \in V_s$, there exists exactly one write event $(\!|z{:=}v|\!)$ in S. We write $\vdash_s S : ok$ for such well-formed configurations.*

2.4.3.2 Channel communication

Channels in Go are the primary mechanism for communication and synchronization. They are typed and assure FIFO communication from a sender to a receiver sharing the channel's reference. In Go, the type system can be used to actually distinguish "read-only" and "write-only" usages of channels, *i.e.,* usages of channels where only receiving (resp. sending) is allowed. Very few restrictions are imposed on the types of channels. Data that can be sent over channels include channels themselves (more precisely references to channels) and closures, including closures involving higher-order functions. Channels can be dynamically created and closed. Channels are *bounded, i.e.,* each channel has a finite capacity fixed upon creation. Channels of capacity 0 are called *synchronous.*

We largely ignore that channel values are typed and that only values of an appropriate type can be communicated over a given channel. We also ignore the distinction between read-only and write-only channels.

$$\frac{q = [\sigma_\perp, \ldots, \sigma_\perp] \qquad |q| = v \qquad \mathit{fresh}(c)}{p\langle \texttt{let}\ r = \texttt{make}\ (\texttt{chan}\ T, v)\ \texttt{in}\ t\rangle \ \rightarrow\ \nu c\ (p\langle \texttt{let}\ r = c\ \texttt{in}\ t\rangle \parallel c_f[] \parallel c_b[q])}\ \textsc{R-Make}_s$$

$$\frac{\neg \mathit{closed}(c_f[q_2])}{c_b[q_1 :: \sigma_\perp] \parallel p\langle c \leftarrow v; t\rangle \parallel c_f[q_2] \ \rightarrow\ c_b[q_1] \parallel p\langle t\rangle \parallel c_f[v :: q_2]}\ \textsc{R-Send}_s$$

$$\frac{v \neq \perp}{\begin{array}{llll} c_b[q_1] \parallel & p\langle \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: v] & \rightarrow \\ c_b[\sigma_\perp :: q_1] \parallel & p\langle \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}\ \textsc{R-Rec}_s$$

$$\frac{}{p\langle \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \parallel c_f[\perp] \ \rightarrow\ p\langle \texttt{let}\ r = \perp\ \texttt{in}\ t\rangle \parallel c_f[\perp]}\ \textsc{R-Rec}_s^\perp$$

$$\frac{}{\begin{array}{lllll} c_b[] \parallel & p_1\langle c \leftarrow v; t\rangle & \parallel p_2\langle \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t_2\rangle & \parallel c_f[] & \rightarrow \\ c_b[] \parallel & p_1\langle t\rangle & \parallel p_2\langle \texttt{let}\ r = v\ \texttt{in}\ t_2\rangle & \parallel c_f[] & \end{array}}\ \textsc{R-Rend}_s$$

$$\frac{\neg \mathit{closed}(c_f[q])}{p\langle \texttt{close}\ (c); t\rangle \parallel c_f[q] \ \rightarrow\ p\langle t\rangle \parallel c_f[\perp :: q]}\ \textsc{R-Close}_s$$

Figure 2.8: Strong operational semantics: channel communication

In our semantics (see Fig. 2.8), a channel c is composed of two queues: the *forward queue* $c_f[q]$ and the *backward queue* $c_b[q]$. When a channel of capacity k is created, the forward queue is empty and the backward queue is initialized so that it contains dummy elements $\sigma_\perp$ (cf. rule R-MAKE$_s$). The dummy elements represent the number of empty or free slots in the channel. Upon creation, the number of dummy elements equals the capacity of the channel. Values sent on (resp. received from) a channel are stored in (resp. removed from) the forward queue; see rule R-SEND$_s$ and R-REC$_s$. When a message is sent on (resp. removed from) the channel, the number of dummy elements in the backward queue is decremented (resp. incremented). Closing a channel resembles sending a special end-of-transmission value $\perp$; see rule R-CLOSE$_s$.

Starting from an *initial weak configuration*, the semantics assures the following invariant.

Lemma 1 (Invariant for channel queues)**.** *The following global invariant holds for a channel c created with capacity k:*

$$|q_f| + |q_b| = k \text{ when c is open} \quad \text{and} \quad |q_f| + |q_b| = k + 1 \text{ when closed.}$$

In the case of synchronous channels, the invariant boils down to $q_f = q_b = []$ *for open channels and* $q_f = [\perp]$ *and* $q_b = []$ *for closed ones.* □

Channels can be closed, after which no new values can be sent otherwise a panic ensues (panics are a form of exception in Go). Values “on transit” in a channel when it is being closed are *not* discarded and can be received as normal. Note that a close operation takes immediate effect regardless of whether the channel is full or not. After the last sent value has been received from a closed channel, it is still possible to receive “further values.” As opposed to blocking, a receive on a closed channel returns the *default* value of the type T, where T is the type passed to `make` when creating the channel. Note that in Go, each type has a well-defined default value. In order to help the receiver disambiguate between 1) receiving a default value on a closed channel and 2) receiving a properly communicated value on a non-closed channel, Go offers the possibility to *check* whether a channel is closed by using so-called *special forms* of assignment. Performing this check is a good defensive programming pattern, although it is not enforced in Go. Instead of using this “in-band signaling” of default values and special forms of assignments, we use a special *value* $\perp$ designating end-of-transmission. Once a channel is closed and the “value“ $\perp$ is placed in the forward queue, it can be no longer be removed. Therefore, clients attempting to receive from the closed channel receive the $\perp$ marker. Note that a difference exists between an empty open channel $c[]$ and an empty closed one $c[\perp]$. Note that the value $\perp$ is pertinent to the forward channel only.

2.4.3.3 Thread creation and select statement

The thread creation rule, presented in Figure 2.9, is unsurprising: the newly spawned thread executes the code t' passed to the corresponding go statement.

$$\frac{\mathit{fresh}(p_2)}{p_1\langle \mathsf{go}\, t';t\rangle \;\to\; \nu p_2\;(p_1\langle t\rangle \parallel p_2\langle t'\rangle)}\ \text{R-G}\textsc{o}_s$$

Figure 2.9: Strong operational semantics: thread creation

The treatment of select statement is identical in the strong and the weak semantics. We therefore postpone the discussion on *select* until Section 2.5.3.4.

2.4.4 Example

Before concluding the sequentially consistent memory model's exposition, we walk through the execution of a program and illustrate the application of many of the derivation rules. As example, we will use the program of Listing 2.2 translated to our syntax (see Listing 2.4).

Listing 2.4: Channel synchronization example using the syntax of Section 2.3

```
int x;
let c = make(chan int, 2) in
  let _ = go { x := 42; c <- 0 } in
    let _ = <- c in load x
```

Figure 2.10 shows a run of the program; [4] the execution steps are enumerated. The first three lines are the initial runtime configuration. It shows the shared variable initialized to 0 and the main thread. In the first step of execution, a channel of size 2 is created according to rule R-MAKE$_s$: the backward queue is initialized to $\sigma_\perp, \sigma_\perp$ and the forward queue is empty. The setup function, here called p_s, is spawn in the second execution step via the application of R-GO$_s$. Since there are no values in the forward queue of channel c, the main thread is blocked on the receive: `let` $\_ = \leftarrow c$. The only possible reduction then is for the setup thread to write to the shared variable x, thus modifying x's associated write event. This happens with the application of

[4] Technically, we have made a small simplification to the program and its execution, which is: we elided the fact that `stop` (*i.e.,* the empty select statement) is the only terminal in the grammar. For ease of exposition, we allow the main thread to end after loading from x and we allow the setup thread to end after sending 0 on x.

R-WRITE$_s$ on execution step 3. In step 4, the setup thread sends 0 onto the channel: the forward queue is appended and the backward queue is shortened by one element (see R-SEND$_s$). At this point p_s has run to the end and the main thread is unblocked. Main receives (and ignores) a value from the channel (rule R-REC$_s$) then loads the content of variable x (rule R-READ$_s$).

$$
\begin{array}{l}
(\!|x{:=}0|\!) \parallel p\langle \texttt{let}\ c\ = \texttt{make}\ (\texttt{chan}\ \texttt{int}, 2)\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \_\ = \texttt{go}\ \{x := 42;\ c \leftarrow 0\}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \_\ = \ \leftarrow c\ \texttt{in}\ \texttt{load}\ x\rangle
\end{array}
$$

$$
\begin{array}{llllll}
\xrightarrow{1} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}0|\!) & \parallel p\langle \texttt{let}\ \_\ = \texttt{go}\ \{x := 42;\ c \leftarrow 0\}\ \texttt{in} \\
 & & & & \qquad \texttt{let}\ \_\ = \ \leftarrow c\ \texttt{in}\ \texttt{load}\ x\rangle \\
\xrightarrow{2} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}0|\!) & \parallel p\langle \texttt{let}\ \_\ = \ \leftarrow c\ \texttt{in}\ \texttt{load}\ x\rangle \parallel \\
 & & & & \quad p_s\langle x := 42;\ c \leftarrow 0\rangle \\
\xrightarrow{3} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}42|\!) & \parallel p\langle \texttt{let}\ \_\ = \ \leftarrow c\ \texttt{in}\ \texttt{load}\ x\rangle \parallel p_s\langle c \leftarrow 0\rangle \\
\xrightarrow{4} & c_f[0] & \parallel c_b[\sigma_\perp] & \parallel (\!|x{:=}42|\!) & \parallel p\langle \texttt{let}\ \_\ = \ \leftarrow c\ \texttt{in}\ \texttt{load}\ x\rangle \parallel p_s\langle\rangle \\
\xrightarrow{5} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}42|\!) & \parallel p\langle \texttt{let}\ \_\ = \ 0\ \texttt{in}\ \texttt{load}\ x\rangle \parallel p_s\langle\rangle \\
\xrightarrow{6} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}42|\!) & \parallel p\langle \texttt{load}\ x\rangle \parallel p_s\langle\rangle \\
\xrightarrow{7} & c_f[] & \parallel c_b[\sigma_\perp, \sigma_\perp] & \parallel (\!|x{:=}42|\!) & \parallel p\langle 42\rangle \parallel p_s\langle\rangle
\end{array}
$$

Figure 2.10: Reduction of a simple program according to the strong operational semantics.

2.5 Weak operational semantics

In this section we define the operational semantics of the main calculus. We fix the run-time configurations of a program before giving the operational rules in Sections 2.5.2 and 2.5.3. Besides processes (or goroutines) running concurrently, the configuration will contain “asynchronous writes” to shared variables.

2.5.1 Local states, events, and configurations

The weak run-time configuration is given by the following syntax:

$$
P ::= p\langle\sigma, t\rangle \ \mid\ m(\!|z{:=}v|\!) \ \mid\ c[q] \ \mid\ \bullet \ \mid\ P \parallel P \ \mid\ \nu n\, P \tag{2.6}
$$

Similar to the strong semantics of Section 2.4, configurations in the weak semantics consist of the parallel composition of goroutines, write events and channels. Different from the strong semantics, a write event is labeled by a unique identifier, typically m, $m_2'\ldots$ Also different, goroutines $p\langle\sigma,t\rangle$ contain, besides the code t to be executed, a local view $\sigma = (E_{hb}, E_s)$ detailing the observability of write events from the perspective of p.

The problem with reasoning about memory writes in the presence of concurrency is similar to the problem of generalizing the assignment Hoare triple $\{Q[e/x]\}x := e\{Q\}$ to the concurrency setting. What is true after an assignment in a single thread model may not be true if the assignment takes place along threads executing concurrently. In particular, interference from other threads may falsify the post condition Q. One has then to either prove interference freedom or to weaken the assertion. We choose not the say what the value of a shared variable *is* at the end of an assignment. Instead, we keep track of what value it *is not*. We call this *local negative information* since it is kept on a per-thread basis.

In our weak operational semantics, all write events of a given configuration are observable by default. If there is more than one write event to a variable, those write events are, by default, observable. So, from a thread's perspective, a variable may hold a superposition of values. It is possible for an event to no longer be visible from a thread's perspective. For example, say thread p writes value v to the shared variable z, thus creating the write event $m(\!|z{:=}v|\!)$. All write events m' in a happens-before relation with p's current action become *shadowed* from p's perspective and are no longer observable. In other words, for all m' such that $m' \rightarrow_{\mathsf{hb}} m$, the value associated with m' is not observable by p. What p can observe by reading from the shared variable z then is the value of any write event $m''(\!|z{:=}v''|\!)$ where $m'' \not\rightarrow_{\mathsf{hb}} m$. This includes the value v that p last wrote to z as well as the value of any other write event that is concurrent with m.

Shadowed events are tracked in the local state σ, specifically in E_s. In order to properly update the list of shadowed events, the local state must also contain thread-local information about the "happens-before" relationship between write events. This information is kept in E_{hb}. We will see how thread local information is updated when we introduce the derivation rules of Section 2.5.3.

Definition 3 (Local state)**.** *A* local state σ *is a tuple of type* $2^{(N\times X)} \times 2^N$. *We use the notation* (E_{hb}, E_s) *to refer to the tuples and abbreviate their type by* Σ. *Let us furthermore denote by* $E_{hb}(z)$ *the set* $\{m \mid (m, !z) \in E_{hb}\}$. *We write* $\sigma_\perp$ *for the local state* $(\emptyset, \emptyset)$ *containing neither happens-before nor shadow information.*

For $\sigma = (E_{hb}, E_s)$ *and* $\sigma' = (E'_{hb}, E'_s)$*, we define* $\sigma + \sigma'$ *as the pairwise union, i.e.,* $\sigma + \sigma' = (E_{hb} \cup E'_{hb}, E_s \cup E'_s)$*. Also, we use* $E_{hb} + (m, !z)$ *as a shorthand for* $E_{hb} \cup \{(m, !z))\}$.

The following holds at the beginning of execution: there is only one goroutine and no channels have been created yet; each shared variable is initialized to a known value; the happens-before set of the main thread records the shared variables' initialization; and there are no shadowed writes from main's perspective.

Definition 4 (Initial weak configuration)**.** *An* initial *weak configuration is of the form*

$$\nu\vec{m}\ (\langle\sigma_0,t_0\rangle \parallel m_0(\!|z_0{:=}v_1|\!) \parallel \ldots \parallel m_k(\!|z_k{:=}v_k|\!))$$

where $z_0,\ldots z_k$ are all shared variables of the program, $\vec{m}$ represents $m_0,\ldots,m_k$, and $\sigma_0 = (E^0_{hb}, E^0_s)$ where $E^0_{hb} = \{(m_0, !z_0),\ldots,(m_k, !z_k)\}$ and $E^0_s = \emptyset$.

The initial weak configuration evolves according to the steps detailed next.

2.5.2 Local steps

Structural congruence $\equiv$ and the local transition steps $\rightsquigarrow$ defined in Section 2.4.2 are carried unchanged from the strong to the weak semantics (cf. Figures 2.4, 2.5, and 2.6). The only addition is rule R-LOCAL, which "lifts" the local reduction relation to the global level of configurations.

$$\frac{t_1 \rightsquigarrow t_2}{\langle\sigma,t_1\rangle \rightarrow \langle\sigma,t_2\rangle}\ \text{R-LOCAL}$$

2.5.3 Global steps

Steps that touch, besides local thread information, shared variables and channels, are detailed next.

2.5.3.1 Reads and writes to shared memory

Rules R-WRITE and R-READ deal with the two basic interactions of threads with shared memory: writing a local value into a shared variable and, inversely, reading a value from a shared variable into the thread-local memory. Writing a value records the corresponding event $m(\!|z{:=}v|\!)$ in the global configuration, with m freshly generated, see rule R-WRITE. The write events are remembered without keeping track of the order of their issuance. Therefore, as far as the global configuration is concerned, no write event ever invalidates an "earlier" write event or overwrites a previous value in a shared variable. Instead, the global configuration accumulates the "positive" information about all available write events which potentially can be observed by reading from shared memory. Values which have never been written cannot be observed, *i.e.,* no out-of-thin-air behavior. Whereas the global configuration remembers all write events indefinitely, filtering out values which are *no longer observable* is handled thread-locally. In other words, which writes are observable depends on the threads' local perspective.

The *local* state σ of a goroutine captures which events are actually observable from a thread-local perspective. Its primary function is to contain "negative" information: a read can observe all write events *except* for those shadowed. A write event is shadowed if its identifier is contained in E_s, see rule R-READ. In addition, the local state keeps track of write events that are thread-locally known to have *happened-before*. These events are stored in E_{hb}. So, issuing a write command (rule R-WRITE) with a write event labeled m updates the local E_{hb} by adding $(m, !z)$. Additionally, the execution of a write instruction causes all previous writes to the variable z (*i.e.,* all writes which are known to have happened-before according to E_{hb}) to become shadowed, thus enlarging E_s. Later in this section, on page 33, we look at an example of reads and writes, their effect on the happened-before and shadow sets, and their impact on a thread's ability to observe memory events.

So the global configurations remember writes indefinitely while the overwriting and thus forgetting previous values is done individually per thread. This, perhaps counter-intuitively, has the following consequence: if a goroutine reads the same shared variable repeatedly, observing a certain value once does not imply that the same value is read next time (even if no new writes are issued to the shared memory). This is because all subsequent readings of the variable are independent and non-deterministically chosen from the set of write events which are not yet shadowed. This also means the semantics allows for a type of relaxation referred to in the literature as coRR. The coRR behavior will be illustrated in the example at the end of this Section, on page 33, and will be addressed further in the Discussion section.

As a final remark, note that it is possible for a write event to be shadowed by all threads in a configuration. A write event that is shadowed by all threads can never again be observed; it can never service any future reads from memory. Although not included in the semantics, we could add a garbage collection rule that removes globally shadowed write events from a configuration.

2.5.3.2 Channel communication

Different from the strong semantics, channel synchronization in the weak semantics must also carry thread-local information. Recall from our discussion of the Go memory model that there are two conditions that need to be satisfied. Condition 2.3 states that a send happens-before its corresponding receive. Therefore, events that are in the sender's past (at the time a message was sent), will also be in the receiver's past when the message is received. In our semantics, this is captured by not only placing into the channel the value being sent, but also the sender's local state. When a goroutine receives a message, it receives the values sent as well as the sender's local state. Information from the sender to its corresponding receiver flows through what we call the channel's forward queue.

Condition 2.4 describes a synchronization effect due to channels' capacity limitation. In our semantics, the capacity limitation is modeled by having local state information flowing in the *opposite direction,* meaning, from a previous receiver to a later sender. This local state information flows through what we call the backward queue. The backward queue accounts for the fact that a sender is only able to place an item into a channel when the channel is not full. The channel not being full means

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (m, !z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(m)}{p\langle \sigma, z := v; t\rangle \;\rightarrow\; \nu m\ (p\langle \sigma', t\rangle \parallel m\llparenthesis z{:=}v \rrparenthesis)}\ \textsc{R-Write}$$

$$\frac{\sigma = (\_, E_s) \qquad m \notin E_s}{p\langle \sigma, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel m\llparenthesis z{:=}v \rrparenthesis \;\rightarrow\; p\langle \sigma, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel m\llparenthesis z{:=}v \rrparenthesis}\ \textsc{R-Read}$$

$$\frac{q = [\sigma_\perp, \ldots, \sigma_\perp] \qquad |q| = v \qquad \mathit{fresh}(c)}{p\langle \sigma, \mathtt{let}\ r = \mathtt{make}\ (\mathtt{chan}\ T, v)\ \mathtt{in}\ t\rangle \;\rightarrow\; \nu c\ (p\langle \sigma, \mathtt{let}\ r = c\ \mathtt{in}\ t\rangle \parallel c_f[] \parallel c_b[q])}\ \textsc{R-Make}$$

$$\frac{\neg \mathit{closed}(c_f[q_2]) \qquad \sigma' = \sigma + \sigma''}{c_b[q_1 :: \sigma''] \parallel p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q_2] \;\rightarrow\; c_b[q_1] \parallel p\langle \sigma', t\rangle \parallel c_f[(v, \sigma) :: q_2]}\ \textsc{R-Send}$$

$$\frac{v \neq \perp \qquad \sigma' = \sigma + \sigma''}{\begin{array}{llll} c_b[q_1] \parallel & p\langle \sigma, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t\rangle & \parallel c_f[q_2 :: (v, \sigma'')] & \rightarrow \\ c_b[\sigma :: q_1] \parallel & p\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}\ \textsc{R-Rec}$$

$$\frac{\sigma' = \sigma + \sigma''}{p\langle \sigma, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t\rangle \parallel c_f[(\perp, \sigma'')] \;\rightarrow\; p\langle \sigma', \mathtt{let}\ r = \perp\ \mathtt{in}\ t\rangle \parallel c_f[(\perp, \sigma'')]}\ \textsc{R-Rec}^{\perp}$$

$$\frac{\sigma' = \sigma_1 + \sigma_2}{\begin{array}{lllll} c_b[] \parallel & p_1\langle \sigma_1, c \leftarrow v; t\rangle & \parallel p_2\langle \sigma_2, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t_2\rangle & \parallel c_f[] & \rightarrow \\ c_b[] \parallel & p_1\langle \sigma', t\rangle & \parallel p_2\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t_2\rangle & \parallel c_f[] & \end{array}}\ \textsc{R-Rend}$$

$$\frac{\neg \mathit{closed}(c_f[q])}{p\langle \sigma, \mathtt{close}\ (c); t\rangle \parallel c_f[q] \;\rightarrow\; p\langle \sigma, t\rangle \parallel c_f[(\perp, \sigma) :: q]}\ \textsc{R-Close}$$

$$\frac{\mathit{fresh}(p_2)}{p_1\langle \sigma, \mathtt{go}\ t'; t\rangle \;\rightarrow\; \nu p_2\ (p_1\langle \sigma, t\rangle \parallel p_2\langle \sigma, t'\rangle)}\ \textsc{R-Go}$$

Figure 2.11: Operational semantics: Global steps

that there must have been a previous receiver who, by receiving and thus removing an item from the channel, created an empty slot on the channel. Therefore, this old receiving action can be placed in the past of the current sender. (There may also be space in the queue because the queue was newly created. As we will see later, this is taken into account by the R-MAKE rule, which governs channel creation.)

Thus, in order to account for their synchronization power, channels in our semantics are composed of two queues. Given that they carry slightly different information, these queues have different types as detailed next.

Definition 5 (Channels)**.** *A channel is of the form $c[q_1,q_2]$, where c is a name and (q_1,q_2) a pair of queues. The first queue, q_1, is also referred to as the* forward *queue. It contains elements of type $(Val \times \Sigma) + (\{\perp\} \times \Sigma)$, where Val is the value sent on the channel, Σ is the local state of the sender when the message was placed on the channel, and $\perp$ is a distinct, separate value representing the "end-of-transmission." The second queue, q_2, is referred to as the* backward *queue. It contains elements of type Σ and propagate the local state of a past receiver to a sender.*

We write (v,σ) and $(\perp,\sigma)$ for forward queue values and (σ) for the backward queue values. Furthermore, we use the following notational convention: We write $c_f[q]$ to refer to the forward queue of the channel and $c_b[q]$ to the backward queue. We also speak of the forward channel and the backward channel. We write $[]$ for an empty queue, $e :: q$ for a queue with e as the element most recently added into q, and $q :: e$ for the queue where e is the element to be dequeued next. We denote with $|q|$ the number of elements in q. A channel is closed*, written closed$(c[q])$, if q is of the form $\perp :: q'$. Note that it is possible for a non-empty queue to be closed.*

When creating a channel (cf. rule R-MAKE) the forward direction is initially empty but the backward is initialized to a queue of length v corresponding to the channel's capacity. The backward queue contains *empty* happens-before and shadow information, represented by the elements $\sigma_\perp$. The rule R-MAKE covers both synchronous and asynchronous channels. A synchronous channel is created with empty forward $c_f[]$ and backward queue $c_b[]$. Channel creation does not involve synchronization.

Rules R-SEND and R-REC govern asynchronous channel communication while R-REND implements synchronous communication. In an asynchronous send, a process places a value on the forward channel along with its local state, provided the channel is not full, meaning: the backward queue is non-empty. In the process of sending, the sender's local state is updated with the knowledge that the previous k^{th} receive has completed; this update is captured by $\sigma' = \sigma + \sigma''$ in the R-SEND rule. To receive a value from a non-empty asynchronous channel (cf. rule R-REC), the communicated value v is stored locally in the rule, ultimately in variable r. Additionally, the local state of the receiver is updated by adding the previously sent local-state information. Furthermore, the state of the receiver before the update is sent back via the backward channel.

In synchronous communication, the receiver obtains a value from the sender and together they exchange local state information. Recall that the Go memory model specifies a send as happening-before its corresponding receive, and the i^{th} receive happening-before the $(i+k)^{th}$ send where k is the channel capacity. Therefore, when a channel is synchronous, $k = 0$, we have that a send happens-before its corresponding receive and the receive happens-before the corresponding send. In other words, synchronous send and receive boil down to a rendezvous between two goroutines. Note that the R-REND can apply only to open synchronous channels, which have empty forward $c_f[]$ and backward queue $c_b[]$. Also note that the rules R-SEND and R-REC do not apply to synchronous channels.

The R-CLOSE rule closes both sync and async channels. R-SEND and R-REC, resp. R-REND no longer apply to closed channels. Executing a receive on a *closed* channel results in receiving the end-of-transmission marker $\perp$ (cf. rule R-REC$^{\perp}$) and updating the local state σ in the same way as when receiving a properly sent value. This happens regardless of whether the channel is synchronous or not. The "value" $\perp$ is not removed from the queue, so that all clients attempting to receive from the closed channel obtain the communicated happens-before synchronization information. Furthermore, there is no need to communicate happens-before constraints from the receiver to a potential future sender on the closed channel: after all, the channel is closed. Consequently the receiver does not propagate back its local state over the back-channel. Closing a channel resembles sending the special end-of-transmission value $\perp$ (cf. rule R-CLOSE). An already closed channel cannot be closed again. In Go, such an attempt would raise a panic. Here, the panic is captured by the absence of enabled transitions.

2.5.3.3 Select statement

Rules dealing with the select statement in the weak semantics are given in Figure 2.12. The R-SEL-SEND and R-SEL-REC rules apply to asynchronous channels and are analogous to R-SEND and R-REC. The R-SEL-SYNC rules apply to open synchronous channels (*i.e.,* the forward and backward queues are empty). The R-SEL-REC$\perp$ is analogous to R-REC$\perp$. Finally, the default rule (R-SEL-DEF) applies when no other select rule applies.

2.5.3.4 Thread creation

Lastly, thread creation leads to a form of a synchronization where the spawned goroutine inherits the local state of the parent (cf. rule R-GO).

2.5.4 Example

Before concluding the memory model's exposition, we revisit the example from the Background section. What follows next is a highlight the differences in execution under the weak semantics versus under the strong one presented in Section 2.4.4. The example involves a main thread that spawns a setup thread, `setup` writes to a shared variable that is later read from `main`. The two threads communicate over a shared channel reference. See Listing 2.4.

The first thing to notice from the run depicted in Figure 2.13 is that, contrasted with the sequentially consistent semantics, write events are now labeled, including the event associated with the initialization of the shared variable x. As we will see, "knowledge" of these events is stored on a per-thread basis and transmitted through

$$\frac{g_i = c \leftarrow v \qquad \neg closed(c_f[q_f]) \qquad \sigma' = \sigma + \sigma''}{\begin{array}{lll} c_b[q_b :: (\sigma'')] \parallel & p\langle \sigma, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle & \parallel c_f[q_f] \ \rightarrow \\ c_b[q_b] \parallel & p\langle \sigma', t_i[()/r_i] \rangle & \parallel c_f[(v, \sigma)) :: q_f] \end{array}} \ \textsc{R-Sel-Send}$$

$$\frac{g_i = \leftarrow c \qquad q_f = q'_f :: (v, \sigma'') \qquad v \neq \bot \qquad q'_b = (\sigma) :: q_b \qquad \sigma' = \sigma + \sigma''}{\begin{array}{lll} c_b[q_b] \parallel & p\langle \sigma, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle & \parallel c_f[q_f] \ \rightarrow \\ c_b[q'_b] \parallel & p\langle \sigma', \mathtt{let}\ r_i = v\ \mathtt{in}\ t_i \rangle & \parallel c_f[q'_f] \end{array}} \ \textsc{R-Sel-Rec}$$

$$\frac{g_i = c \leftarrow v \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{ll} p_1\langle \sigma_1, \sum_i r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel & p_2\langle \sigma_2, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t_2 \rangle \ \rightarrow \\ p_1\langle \sigma', t_i[()/r_i] \rangle \parallel & p_2\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t_2 \rangle \end{array}} \ \textsc{R-Sel-Sync}_1$$

$$\frac{g_i = \leftarrow c \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{ll} p_1\langle \sigma_1, c \leftarrow v; t_1 \rangle \parallel & p_2\langle \sigma_2, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \ \rightarrow \\ p_1\langle \sigma', t_1 \rangle \parallel & p_2\langle \sigma', \mathtt{let}\ r_i = v\ \mathtt{in}\ t_i \rangle \end{array}} \ \textsc{R-Sel-Sync}_2$$

$$\frac{g_i = c \leftarrow v \qquad g_j = \leftarrow c \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{ll} p_1\langle \sigma_1, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel & p_2\langle \sigma_2, \sum_j \mathtt{let}\ r_j = g_j\ \mathtt{in}\ t_j \rangle \ \rightarrow \\ p_1\langle \sigma', t_i[()/r_i] \rangle \parallel & p_2\langle \sigma', \mathtt{let}\ r_j = v\ \mathtt{in}\ t_j \rangle \end{array}} \ \textsc{R-Sel-Sync}_3$$

$$\frac{g_i = \leftarrow c \qquad c_f[(\bot, \sigma'')] \qquad \sigma' = \sigma + \sigma''}{p\langle \sigma, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \ \rightarrow \ p\langle \sigma', \mathtt{let}\ r_i = \bot\ \mathtt{in}\ t_i \rangle} \ \textsc{R-Sel-Rec}_\bot$$

$$\frac{g_i = \mathtt{default} \qquad \neg \exists j.\ i \neq j.\ p\langle \sigma, \sum_j \mathtt{let}\ r_j = g_j\ \mathtt{in}\ t_j \rangle \parallel P \rightarrow p\langle \sigma', t' \rangle \parallel P'}{p\langle \sigma, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel P \ \rightarrow \ p\langle \sigma, t_i[()/r_i] \rangle \parallel P} \ \textsc{R-Sel-Def}$$

Figure 2.12: Operational semantics: Select statement

channels. Also take note of the additional structure σ, which is used to store thread-local information. The main thread starts with local state $\sigma_0 = \{E^0_{hb}, E^0_s\}$, where $E^0_{hb} = \{(m_0, x)\}$ and $E^0_s = \emptyset$. In other words, at the beginning of execution the main thread has: 1) a record of the shared variable's initialization in the happens-before set E^0_{hb}, and 2) no write event identifiers in the shadowed set E^0_s.

In the first reduction step, the main thread creates a new channel. Similar to the sequentially consistent semantics, the channel is composed of two queues; the forward queue q_f is initially empty while the backward queue q_b is initialized to two empty local states $\sigma_\bot, \sigma_\bot$ where $\sigma_\bot = (\emptyset, \emptyset)$. These local states represent the fact

$$
\begin{array}{l}
m_0(\!|x{:=}0|\!) \parallel p\langle \sigma_0, \mathtt{let}\ c\ = \mathtt{make\ (chan\ int,2)\ in} \\
\qquad\qquad\quad \mathtt{let}\ \_\ = \mathtt{go}\ \{x := 42;\ c \leftarrow 0\}\ \mathtt{in} \\
\qquad\qquad\quad\ \ \mathtt{let}\ \_\ =\ \leftarrow c\ \mathtt{in\ load}\ x\rangle
\end{array}
$$

$$
\begin{array}{llllll}
\xrightarrow{1} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_0, \mathtt{let}\ \_\ = \mathtt{go}\ \{x := 42;\ c \leftarrow 0\}\ \mathtt{in} \\
 & & & & \qquad\qquad\quad \mathtt{let}\ \_\ =\ \leftarrow c\ \mathtt{in\ load}\ x\rangle \\
\xrightarrow{2} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_0, \mathtt{let}\ \_\ =\ \leftarrow c\ \mathtt{in\ load}\ x\rangle \parallel \\
 & & & & \ \ p_s\langle \sigma_0, x := 42;\ c \leftarrow 0\rangle \\
\xrightarrow{3} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_0, \mathtt{let}\ \_\ =\ \leftarrow c\ \mathtt{in\ load}\ x\rangle \parallel \\
 & & & \ \ m_1(\!|x{:=}42|\!) & \parallel p_s\langle \sigma_1, c \leftarrow 0\rangle \\
\xrightarrow{4} & c_f[(0,\sigma_1)] & \parallel c_b[\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_0, \mathtt{let}\ \_\ =\ \leftarrow c\ \mathtt{in\ load}\ x\rangle \parallel \\
 & & & \ \ m_1(\!|x{:=}42|\!) & \parallel p_s\langle \sigma_1, \rangle \\
\xrightarrow{5} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_2, \mathtt{let}\ \_\ =\ 0\ \mathtt{in\ load}\ x\rangle \parallel \\
 & & & \ \ m_1(\!|x{:=}42|\!) & \parallel p_s\langle \sigma_1, \rangle \\
\xrightarrow{6} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_2, \mathtt{load}\ x\rangle \parallel \\
 & & & \ \ m_1(\!|x{:=}42|\!) & \parallel p_s\langle \sigma_1, \rangle \\
\xrightarrow{7} & c_f[] & \parallel c_b[\sigma_\bot,\sigma_\bot] & \parallel m_0(\!|x{:=}0|\!) & \parallel p\langle \sigma_2, 42\rangle \parallel \\
 & & & \ \ m_1(\!|x{:=}42|\!) & \parallel p_s\langle \sigma_1, \rangle
\end{array}
$$

Figure 2.13: Reduction of a simple program according to the weak operational semantics.

that the channel has capacity two and is currently empty. In the second reduction step, `main` forks a new thread p_s. According to rule R-GO, the new thread inherits the parent's local state σ_0. After that, the main thread blocks attempting to receive from an empty channel.

Next, p_s writes 42 to x. According to R-WRITE, this creates a new write event with a fresh name, m_1, and modifies p_s's local state to $\sigma_1 = (E^1_{hb}, E^1_s)$. Naturally, as p_s is aware of its own writing to x, the write event m_1 is recorded in p_s's happens-before set, meaning $(m_1,x) \in E^1_{hb}$. The initial value of x is no longer visible from p_s's perspective, since it has been overwritten by the more recent write event m_1. Therefore, the write event m_0 associated with x's initialization is placed in p_s's shadow set, meaning $m_0 \in E^1_s$. Therefore, $\sigma_1 = (\{(m_0,x),(m_1,x)\},\{m_0\})$.

Note that, at this point, the write event m_1 is not yet recorded into the main thread's local state. Note that, according to the read rule, R-READ, the write event m_1 *is observable* from `main`'s perspective. So is the initial write event m_0. As we will see in the Discussion section, this superposition of values is known in the memory-model literature as the coRR relaxation. When it comes to this particular example, the main thread is blocked, thus it is not able to read from x and the coRR behavior does not emerge.

Next, the setup thread sends a message onto the shared channel; see step 4. According to R-SEND, the message's value is placed into the channel along with the sender's current local state. Then, in step 5, `main` receives the message and updates its local state. The new local state, σ_2 reflects the fact that `main` is now aware of the events that took place (according to the sender's perspective) when the message was put onto the channel. In other words, `main`'s local state is the old local state σ_0 augmented by the state σ_1 received through channel communication:

$$\sigma_2 = \sigma_0 + \sigma_1 = (\{(m_0, x), (m_1, x)\}, \{m_0\})$$

The communication served to synchronize the actions of the setup thread from the perspective of the main thread. At this point, `main` is also not able to observe the initial value of the shared variable $x = 0$. The only observable write event is m_1; therefore, the `load` x reduces to 42 in step 6.

It is worth to note that, without channel communication, synchronization would not have been possible. For example, if instead of sending a message, `setup` and `main` tried to synchronize by writing to a shared variable (as shown in Listing 2.1), then `main`'s local state would not be updated to reflect the actions performed by `setup`. The program would contain a data race in this case.

2.6 Relating the strong and the weak semantics

This section describes the relationship between the strong and the weak semantics. After some preliminary definitions, Section 2.6.1 covers the easy direction: the weak semantics subsumes the strong one. The converse direction does not hold in general; it holds only when excluding race condition. This is established in Section 2.6.2. Additional intermediate lemmas are relegated to the appendix, in particular Appendix A.2.

Let us recall the definition of simulation [70] relating states of labeled transition systems. The set of transition labels and the information carried by the labels may depend on the specific steps or transitions done by a program and/or the observations one wishes to attach to those steps. This design choice leads to a distinction between internally and externally visible steps. Let us write α for arbitrary transition labels. Later we will use a for visible labels and τ as the label of invisible or internal steps.

Definition 6 (Simulation). *Assume two labeled transition systems over the same set of labels and with state sets S and T. A binary relation $\mathcal{R} \subseteq S \times T$ is a simulation relation between the two transition systems if $s_1 \xrightarrow{\alpha} s_2$ and $s_1 \mathrel{\mathcal{R}} t_1$ implies $t_1 \xrightarrow{\alpha} t_2$ for some state t_2. Diagrammatically:*

$$\begin{tikzcd}
s_1 \arrow[r, dash, "R" description] \arrow[d, "\alpha"] & t_1 \arrow[d, dotted, "\alpha"] \\
s_2 \arrow[r, dotted, dash, "R" description] & t_2
\end{tikzcd}$$

A state t simulates *s, written $t \gtrsim s$, if there exists a simulation relation $\mathcal{R}$ such that $s\,\mathcal{R}\,t$.*

We use formulations like "*s* is simulated by *t*" interchangeably, and $\lesssim$ as the corresponding symbol. Also, we subscript the operational rules for disambiguation; for example, R-READ$_s$ refers to the strong version of the read while R-WRITE$_w$ to the weak version of the write operation. The rules of the strong semantics are simplifications of the weak rules given in Section 2.5. More concretely, in the strong semantics, write events are unique per variable, goroutines do not have a local state σ, and channels do not carry local state information

The operational semantics is given as unlabeled global transitions $\rightarrow$. To establish the relationship between the strong and the weak semantics, we make the steps of the operational semantics more "informative" by labeling them appropriately: For read steps by rule R-READ$_s$ and R-READ$_w$, when reading a value v from a variable z, the corresponding step takes the form $\xrightarrow{(z?v)}$. All other steps, $\rightarrow$ as well as $\rightsquigarrow$ steps, are treated as invisible and noted as $\xrightarrow{\tau}$ in the simulation proofs. We make use of the following "alternative" labeling for the purpose of defining races and for some of the technical lemmas: we label write and read steps with the identity of the goroutine responsible for the action and the affected shared variable. Additionally, we sometimes mention as part of the label the identity n of the concerned *write* event. The labeled transitions are thus of the form $\xrightarrow{n(z!)_p}$ or $\xrightarrow{n(z?)_p}$. When not needed in the formulation of a property or a proof, we omit mentioning irrelevant parts of the transition labels. We often use subscripts when distinguishing the strong from the weak semantics; *e.g.* $\xrightarrow{(z!)p}_w$ and $\xrightarrow{(z!)p}_s$. We write $\Rightarrow$ for $\xrightarrow{\tau}^*$ and $\overset{a}{\Rightarrow}$ for $\xrightarrow{\tau}^*\xrightarrow{a}\xrightarrow{\tau}^*$.

2.6.1 The weak semantics simulates the strong

Lemma 2 (Simulation)**.** *Let S_0 and P_0 be a strong, resp. a weak initial configuration (for the same program with the same initial values for the global variables). Then $P_0 \gtrsim S_0$.*

The proof is given in Appendix A.1.

2.6.2 The strong semantics conditionally simulates the weak one

It should be intuitively clear and expected that the weak semantics "contains" the sequentially consistent strong one as special case. In other words, we expect the weak semantics to be able to simulate the strong one. Equally clear is that the opposite direction —the strong semantics simulates the weak— does *not* hold in general. If a simulation relation would hold in both directions, the two semantics would be equivalent,[5] thus obviating the whole point of a weak or relaxed memory model.

Simulation of the weak semantics by the strong one can only be guaranteed "conditionally." The standard condition is that the program is "well-synchronized." We take that notion to represent the absence of data races, where a data race is a situation in which two different threads access the same shared variable, at least one of the accesses is a write, and the accesses are not ordered by the happens-before relation. The definition is used analogously for the weak semantics.

From the fact that the weak semantics simulates the strong one, we have that every race condition in the strong semantics can be exhibited in the weak. The converse, however, is not true: the weak semantics has races not present in the strong one. The *new* races in the weak semantics come from the fact that *once a race is reachable*, the weaker version of the semantics allows values to be read which are unobservable to the corresponding sequentially consistent configuration. Therefore, the *first* race condition is what leads the weak semantics to behaviors not present in the strong one. Naturally, if a program is race free from the strong semantics' perspective, it must be race free from the weak's perspective as well. In other words, when checking for race freedom, it suffices to observe behavior under the strong semantics, which is arguably simpler.

This is, of course, an informal discussion. Next we prove that the weak semantics upholds the DRF-SC guarantee. The proof will be another simulation result: the strong semantics conditionally simulates the weak one; the condition requires programs to be data race free.

2.6.2.1 General invariant properties

Let us introduce some general properties of the weak semantics (*i.e.,* without assuming race freedom) that will be useful later in conditional simulation proof. The proofs of the lemmas presented next are mostly relegated to Appendix A.2.2.1.

Definition 7 (Observable and concurrent writes)**.** *Let W_P stand for the set of all write events $m(\!|z{:=}v|\!)$ in a weak configuration P and let $W_P(z)$ stand for the set of identifiers of writes events to the variable z:*

$$W_P(z) = \{m \mid m(\!|z{:=}v|\!) \in W_P\} \;. \tag{2.7}$$

Given a well-formed configuration P, the sets of writes that happens-before*, that are* concurrent*, and that are* observable *by process p for a variable z are defined as*

[5] A simulation in both directions, *i.e.,* the relation $\gtrsim \cap \lesssim$, does not technically correspond to bisimulation, but expresses a form of equivalence nonetheless.

follows:

$$
\begin{aligned}
W_P^{\mathsf{hb}}(z@p) &= E_{hb}(z@p) && (2.8)\\
W_P^{|||}(z@p) &= W_P(z) \setminus E_{hb}(z@p) && (2.9)\\
W_P^{\circ}(z@p) &= W_P(z) \setminus E_s(z@p)\ . && (2.10)
\end{aligned}
$$

We also use notations like $W_P^{\circ}(\_@p)$ to denote the set of observable write events in P for any shared variable.

Lemma 3 (Invariants about write events)**.** *The weak semantics has the following invariants.*

1. *For all local states (E_{hb}, E_s) of all processes, $E_s \subset E_{hb}(z)$.*
2. *$W_P^{|||}(z@p) \subseteq W_P^{\circ}(z@p)$.*
3. *$W_P^{|||}(z@p) \neq W_P^{\circ}(z@p)$.*
4. *$W_P^{\mathsf{hb}}(z@p) \cap W_P^{\circ}(z@p) \neq \emptyset$.*

As $W_P^{\circ}(z@p)$ is a proper superset of $W_P^{|||}(z@p)$ by part (2) and (3), each thread can observe at least one value held by a variable. This means, unsurprisingly, that no thread will encounter an "undefined" variable. More interesting is the following generalization, namely that at each point and for each variable, some value is *jointly* observable by all processes. The property holds for arbitrary programs, race-free or not. Under the assumption of race-freedom, we will later obtain a stronger "consensus" result: not only is a consensus possible, but there is *exactly one* possible observable write, not more.

Lemma 4 (Consensus possible)**.** *Weak configurations obey the following invariant*

$$\bigcap\nolimits_{p \in P} W_P^{\circ}(z@p) \neq \emptyset\ . \qquad (2.11)$$

2.6.2.2 Race-free reductions

Next, we present invariants that hold specifically for race-free programs but not generally. These invariants will be needed to define the relationship between the strong and weak semantics via a bisimulation relation. More concretely, the following properties are ultimately needed to establish that the relationship connecting the strong and the weak behavior of a program is well-defined.

Lemma 5 (No concurrent writes when it counts)**.** *Let P be a reachable configuration in the weak semantics, i.e., $P_0 \rightarrow_w^* P$ where P_0 is the initial configuration derived from program P.*

1. *Assume P has no read-write race. If* $P \xrightarrow{(z?)p}_w$, *then* $W_P^{|||}(z@p) = \emptyset$.
2. *Assume P has no write-write race. If* $P \xrightarrow{(z!)p}_w$, *then* $W_P^{|||}(z@p) = \emptyset$.

The following lemma, resp. the subsequent corollary express a welcome invariant concerning the observability of write events for a given variable z and seen from the perspective of a thread doing the next read or write step. At the point specified by the lemma, there is *exactly* one write event for z, which is observable by p, and actually its *commonly* observable by sets of threads that includes the thread in question. As one consequence, each read-step by a thread in a configuration of race-free program observes exactly *one* value as opposed to choosing non-deterministically.

Lemma 6 (Race-free consensus when it counts)**.** *Assume* $P_0 \rightarrow_w^* P$ *with* P_0 *race-free. If* $P \xrightarrow{(z?)p}_w$ *or* $P \xrightarrow{(z!)p}_w$, *then there exists a write event* $m(\!|z{:=}v|\!)$ *such that*

$$\bigcap_{p_i} W_P^{\circ}(z@p_i) = \{m\} \ , \tag{2.12}$$

where the intersection ranges over an arbitrary set of processes which includes p.

Corollary 7 (Locally deterministic read)**.** *Assume* $P_0 \rightarrow_w^* P$ *with* P_0 *race-free. Then* $P \xrightarrow{n_1(?)p}_w$ *and* $P \xrightarrow{n_2(?)p}_w$ *implies* $n_1 = n_2$.

Lemma 8 (Race-free consensus)**.** *Weak configurations for race-free programs obey the following invariant*

$$\bigcap_{p_i \in P} W_P^{\circ}(z@p_i) = \{m\} \tag{2.13}$$

for some write event $m(\!|z{:=}v|\!)$.

Definition 8 (Well-formedness for race-free programs)**.** *A weak configuration P is* well-formed *if*

1. *write-event references and channel references are unique, and*
2. *equation (2.13) from Lemma 8 holds.*

We write $\vdash_w^{rf} P : ok$ *for well-formed configurations P.*

We need to relate the weak and strong configurations via a simulation relation in order to establish the connection between the race-free behaviors of the weak and strong semantics. We will do so by the means of an erasure function from the weak to the strong semantics.

Definition 9 (Erasure)**.** *The erasure of a well-formed weak configuration P, written* $\lfloor P \rfloor$, *is defined as* $\lfloor P \rfloor^{\emptyset}$ *where* $\lfloor P \rfloor^{R}$ *is given in Table 2.1 and R is a set of write event identifiers. On the queues* q_1 *and* q_2 *in the last case, the function simply jettisons the* σ*-component in the queue elements.*

$$\lfloor \bullet \rfloor^R = \bullet \tag{2.14}$$

$$\lfloor p\langle \sigma, t\rangle \rfloor^R = \langle t\rangle \tag{2.15}$$

$$\lfloor m(\!|z{:=}v|\!) \rfloor^R = \begin{cases} \bullet & \text{if } m \in R \\ (\!|z{:=}v|\!) & \text{otherwise} \end{cases} \tag{2.16}$$

$$\lfloor P_1 \parallel P_2 \rfloor^R = \lfloor P_1 \rfloor^R \parallel \lfloor P_2 \rfloor^R \tag{2.17}$$

$$\lfloor \nu n\, P \rfloor^R = \begin{cases} \lfloor P \rfloor^R & \text{if } \forall p \in P.\ n \in W_P^\circ(\_@p) \\ \lfloor P \rfloor^{R \cup \{n\}} & \text{otherwise} \end{cases} \tag{2.18}$$

$$\lfloor c[q_1, q_2] \rfloor^R = c[\lfloor q_1 \rfloor^R, \lfloor q_2 \rfloor^R] \tag{2.19}$$

Table 2.1: Definition of the erasure function $\lfloor P \rfloor^R$

Note that $\lfloor P \rfloor$ is not necessarily a well-formed strong configuration. In particular, $\lfloor P \rfloor$ may contain two different write events $(\!|z{:=}v_1|\!)$ and $(\!|z{:=}v_2|\!)$ for the same variable. Besides, it is not *a priori* clear whether $\lfloor P \rfloor$ could remove all write events for a given variable (thus leaving its value undefined) and the configuration ill-formed.

Lemma 9 (Erasure and congruence). *$P_1 \equiv P_2$ implies $\lfloor P_1 \rfloor \equiv \lfloor P_2 \rfloor$.*

Lemma 10 (Erasure preserves well-formedness). *Let P be a race-free reachable weak configuration. If $\vdash_w P : ok$ then $\vdash_s \lfloor P \rfloor : ok$.*

Theorem 11 (Race-free simulation). *Let S_0 and P_0 be a strong, resp. a weak initial configuration for the same thread t and representing the same values for the global variables. If S_0 is data-race free, then $S_0 \gtrsim P_0$.*

Proof. Assume two initial race-free configurations P_0 and S_0 from the same program and the same initial values for the shared variables. To prove the $\gtrsim$-relationship between the respective initial configurations we need to establish a simulation relation, say $\mathcal{R}$, between well-formed strong and weak configurations such that P_0 and S_0 are in that relation.

Let P and S be well-formed configurations reachable (race-free) from P_0 resp. S_0. Define $\mathcal{R}$ as relation between race-free reachable configurations as

$$P\,\mathcal{R}\,S \quad \text{if} \quad S \equiv \lfloor P \rfloor \tag{2.20}$$

using the erasure from Definition 9. Note that by Lemma 9, $P_1\ \mathcal{R}\ S$ and $P_1 \equiv P_2$ implies $P_2\ \mathcal{R}\ S$.

Case: R-WRITE$_w$: $p\langle\sigma, z := v; t\rangle \rightarrow_w \nu m\,(p\langle\sigma', t\rangle \parallel m(\!|z{:=}v|\!))$, where $\sigma = (E_{hb}, E_s)$ and $\sigma' = (E'_{hb}, E'_s) = (E_{hb} + (m,z), E_s + E_{hb}(z))$. By the concurrent-writes Lemma 5(2), $W_P^{|||}(z@p) = \emptyset$, *i.e.,* there are no concurrent write events from the perspective of p. This implies that for all write events $m'(\!|z{:=}v'|\!)$ in P, we have $m' \in E_{hb}$. If $m' \in E_s$, then $m \in E'_s$ as well. If $m' \in E_{hb} \setminus E_s$, then $m' \in E'_s$ as well. Either way, *all* write events to z contained in P prior to the step are shadowed in p after the step.

Now for the new write event m in P': clearly $m \in W^{\circ}_{P'}(z@p_i)$, *i.e.,* the event is observable for all threads. By the race-free consensus Lemma 8, we have that this is the only event that is observable by all threads, *i.e.,*

$$\bigcap_{p_i} W^{\circ}_{P'}(z@p_i) = \{m\}\ . \tag{2.21}$$

That means for the erasure of P' that $\lfloor P' \rfloor \equiv \ldots \parallel p\langle t\rangle \parallel (\!|z{:=}v|\!)$ where $(\!|z{:=}v|\!)$ is the result of applying $\lfloor\_\rfloor$ to the write event $m(\!|z{:=}v|\!)$ of P. In particular, equation (2.21) shows that the write event m is not "filtered out" (cf. the cases of equation (2.16) and (2.18) in Definition 9) and furthermore that all other write events for z in P' *are* filtered out.[6] It is then easy to see that by R-WRITE$_s$, $\lfloor P \rfloor \rightarrow_s \lfloor P' \rfloor$.

The remaining cases are similar. □

2.7 Implementation

We have implemented the strong and the weak semantics in $\mathbb{K}$, a rewrite-based executable semantics framework [51, 87]. Concretely, the implementation helped us work through corner cases in the semantics. In addition, we believe the code can help the interested reader assimilate the reduction rules and explore alternatives by making modifications to the sources available online [29]. We have made use of $\mathbb{K}$'s built-in types and data-structures (`Set`, `Map`, and `List`), which we believe facilitated the work. The code is modular. In fact, most of the implementation (*i.e.* rules related to local steps, goroutine creation, channel communication) is reused between the weak and strong semantics. The implementation of the weak and strong semantics differ only when it comes to the treatment of memory.

To give a flavor of the rewriting rules, we start by looking at part of the implementation of the R-RECEIVE rule in Figure 2.11. The code, given in Figure 2.14, involves a `goroutine` receiving a value from a `chan`. The condition under "`requires`" stipulates additionally that the value being read from the channel must not be the special end-of-transmission marker (the act of attempting to receive from a previously closed channel is handled by a different rewrite rule).

A term to the left of => is rewritten to the term on the right. In this particular case, the receive reduces to V (line 2) corresponding to the head of the forward

[6] The latter is indirectly clear already as we have established that $\lfloor\rfloor$ preserves well-formedness under the assumption of race-freedom (Lemma 10).

queue (line 12). The receiving goroutine's local state is updated. In specific, its happens-before and shadowed information (HMap and SSet on lines 4 and 5) are rewritten to take into account the happens-before and shadow information in the forward queue (HMapDp and SSetDp on lines 13 and 14 resp.). The received entry is removed from the forward queue (lines 12-14) and the receiver's local state is added to the channel's backward queue (lines 15 and 16).

```
1  rule <goroutine>
2        <k> <- channel(Ref:Int) => V ... </k>
3        <sigma>
4          <HB> HMap:Map => mergeHB(HMap, HMapDP) </HB>
5          <S>  SSet:Set => SSet SSetDP </S>
6        </sigma>
7        <id> _ </id>
8      </goroutine>
9      <chan>
10      <ref> Ref </ref>
11      <type> _ </type>
12      <forward> ListItem( ListItem(V)
13                ListItem(HMapDP)
14                ListItem(SSetDP) ) => .List </forward>
15      <backward> BQ:List => ListItem( ListItem(HMap)
16                                ListItem(SSet)) BQ </backward>
17     </chan>
18     requires notBool( V ==K $eot )
```

Figure 2.14: Snippet from the implementation of the channel receive rule in 𝕂

The implementation gives us the ability to execute programs and observe their output. At the start of execution, the runtime configuration has the format shown in Figure 2.15, with goroutines held inside <G>...</G>, write events inside <W>...</W>, and channels inside <C>...</C>. The initial configuration features a single goroutine whose id is 1 (line 4). Initially, this goroutine holds no happens-before or shadowed information (lines 7 and 8 resp.). The tokens $PGM:Pgm are a placeholder for a syntactically valid program that gets filled by 𝕂 when execution starts. If the program declares shared variables, the implementation initializes them to 0.

As execution progresses, meaning, as write events are recorded and additional goroutines and channels are created, the configuration is expanded. Take the example given in Section 2.2, where a simple setup function is called asynchronously from main. The example, rewritten in the proposed syntax, is shown in Listing 2.4. Coordination is achieved through a shared channel. A message indicates that the setup is complete and, according to the semantics of channel communication, the receiver can no longer read the initial shared variable's value and will instead read the value updated by the setup function.

```
<mmgo>
  <G>
    <goroutine>
      <id> 1 </id>
      <k> $PGM:Pgm </k>
      <sigma>
        <HB> .Map </HB>
        <S> .Set </S>
      </sigma>
    </goroutine>
  </G>
  <W> .Map </W>
  <C> .ChanCellBag </C>
</mmgo>
```

Figure 2.15: The initial runtime configuration

Figure 2.16 shows the end configuration[7] for a run of the example. In it, there are two goroutines: the "main" goroutine (whose id is 1) terminates in state "42" (line 4) corresponding to the value read from the shared variable. The "setup" goroutine terminates in the state `unit` (line 11), which is the value resultant from executing its last instruction, namely `c<-0`. Note that there are two write events recorded in the final configuration. One coming from the initialization of `x` to 0 and another corresponding to the write of 42 into `x` by the "setup" goroutine. Note also the presence of a channel inside `<C>`, which was created by "main" to coordinate with "setup."

2.8 Discussion

This section positions our work in a wider context, revisiting notions from *axiomatic* semantics of memory models and using litmus tests to highlight similarities and differences between our semantics and a well-formulated axiomatic one [3]. In the axiomatic semantics of memory models, the execution of a given program (*i.e.,* the manifestation of a particular control flow and thread interleaving) gives rise to *candidate executions.* Candidate executions are graphs that help define and illustrate behavior accepted or rejected by the semantics; see [10] as example. The graphs are composed of events (nodes) representing memory operations and relations (edges) over events. In this section we use $(n{:}\mathsf{R}x = v)_p$ and $(n{:}\mathsf{W}x = w)_p$ for read and write events of a value v on a shared variable x, where n is the unique identifier and p is the identifier of the thread responsible for the event. The thread identifier is omitted when it can be deduced from the context.

[7] An end configuration is a configuration to which no further rewrite rules apply.

```
<mmgo>
  <G>
    <goroutine> <id> 1 </id>
      <k> 42 </k>
      <sigma>
        <HB> x |-> ( SetItem ( 3 ) SetItem ( 7 ) ) </HB>
        <S> SetItem ( 3 ) </S>
      </sigma>
    </goroutine>
    <goroutine> <id> 5 </id>
      <k> $unit </k>
      <sigma>
        <HB> x |-> ( SetItem ( 3 ) SetItem ( 7 ) ) </HB>
        <S> SetItem ( 3 ) </S>
      </sigma>
    </goroutine>
  </G>
  <W> x |-> ( 3 |-> 0 7 |-> 42 ) </W>
  <C>
    <chan>
      <ref> 4 </ref>
      <type> int </type>
      <forward> .List </forward>
      <backward>
        ListItem(ListItem( x |-> SetItem(3) ) ListItem(.Set))
        ListItem(ListItem(.Map) ListItem(.Set)) </backward>
    </chan>
  </C>
</mmgo>
```

Figure 2.16: Sample output from running Listing 2.4 on the weak semantics

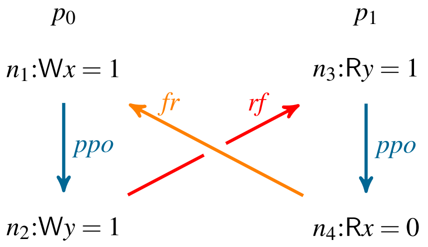
Aspects of a memory model are often captured by litmus tests, which are tailor-made code snippets that highlight features of a memory model. As illustration, on the left of Figure 2.17 is the well-known litmus test for *message passing* (mp) and, on the right, a corresponding candidate execution. The code snippet shows process p_0 sending data to p_1 via x and using a write to y as signal that the data is “ready.” For this simple form of synchronization to work, the observation $r_1 = 1$ and $r_2 = 0$ must be *forbidden*. The underlying assumptions, in this case, are that 1) the order of reads by p_1 reflects the order in which the writes are effected by p_0, and 2) the writes by p_0 respect program order.

The candidate execution of Figure 2.17b gives a justification for the impossibility of the observation $r_1 = 1$ and $r_2 = 0$ which violates the mp pattern. The edge $n_2 \rightarrow_{\mathsf{rf}} n_3$ of the “read-from” relation $\rightarrow_{\mathsf{rf}}$ expresses the fact that n_3 reads the value written by n_2. More complex is the “from-read” relation: the edge $n_4 \rightarrow_{\mathsf{fr}} n_1$ stipulates that n_4 “reads-from” some write event left unmentioned and for which n_1 comes “after.” More precisely, it abbreviates $n_0 \rightarrow_{\mathsf{rf}} n_4$ for some write event n_0 with

p_0	p_1
`x := 1;`	`r1 := y;`
`y := 1;`	`r2 := x;`

not ($r_1 = 1$ and $r_2 = 0$)

(a) Litmus test



(b) Candidate execution

Figure 2.17: Message passing (mp)

$n_0 \rightarrow_{\mathsf{co}} n_1$ and where $\rightarrow_{\mathsf{co}}$ represents the *coherence order*, which is a total order of writes over the same memory location. In the example, n_0 is the write event setting x to its initial value 0; by convention, such initialization events are often left out of candidate executions. Using the mentioned coherence order, the from-read relation captures the intuition that a read observes a value written *prior* to subsequent write. In contrast to the concept of coherence order, our model does not employ the notion of a total order of writes on a location. Instead, information about which writes are observable by a read is kept *local* per thread and past writes events are considered unordered.

Note from the mp example that the *preserved* program order edges $n_1 \rightarrow_{\mathsf{ppo}} n_2$ and $n_3 \rightarrow_{\mathsf{ppo}} n_4$ disallow out-of-order execution of the two writes and, also, of the two reads. The preservation of program order is characteristic for strong memory models such as the semantics presented in Section 2.4. In weaker settings, the $\rightarrow_{\mathsf{ppo}}$-edges may be replaced by $\rightarrow_{\mathsf{po}}$-edges. For example, both our model and PSO-style memory models with per-location write buffers allow the observation $r_1 = 1$ and $r_2 = 0$ in the mp litmus test of Figure 2.17. From our perspective, however, the treatment of the writes is best not seen as "buffering" since, after all, the value of a write becomes *immediately* observable in our operational semantics. In our weak memory model, it is the *negative* information of being *unobservable* that is not immediately available to all observers. To percolate through the system, this negative information requires synchronization via channel communication.

Another aspect of our semantics is that, from an observer thread's perspective, writes from different threads never invalidate each other. In the absence of synchronization, writes from other threads remain observable indefinitely. A litmus test typifying that kind of behavior is known as coRR,[8] shown in Figure 2.18.

The fact that repeated reads by the same thread give different seemingly incoher-

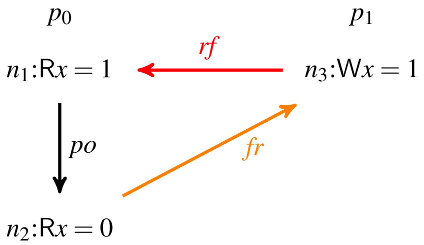
[8] In general, coherence tests coXY involve an access of kind X and an access of kind Y with X and Y standing for either R (read) or W (write).

p_0	p_2

`r1 := x;`
`r2 := x;`
	`x := 1;`

$r_1 = 1, r_2 = 0$

(a) Litmus test



(b) Candidate execution

Figure 2.18: coRR

ent values can be interpreted as a form of oscillation: when reads and writes happen "at the same time," *i.e.,* in a racy way without proper synchronization, the memory can be perceived as oscillating. Conceptually, in the example of Figure 2.18, x oscillates between the old value 0 and the new value of 1 indefinitely.

This behavior is allowed by our proposed semantics. As a matter of fact, it is also allowed by Sparc RMO [48] and pre-Power4 machines [96]. Many other models, though, including the axiomatization by [3], disallow the coRR behavior.

Load buffering is a relaxation which complements write buffering. Its effect is often illustrated by the litmus test of Figure 2.19. The candidate execution graph shows a run which justifies $r_1 = 1$ and $r_2 = 1$ as follows: the load or read of event n_1 is buffered, thereby taking effect after the write event n_4. This causes the instructions n_3 and n_4 to be executed *out-of-order*. For p_0, however, the read cannot be postponed until after the write, as the value of the write *depends*, via r_1, on the value being read. Program order has to be preserved due to a data dependence, indicated by a $\rightarrow_{\mathsf{ppo}}$-edge. The circumstances in which program order is preserved depends on the programming language semantics and/or the given hardware memory model. For example, various forms of special fence instructions (*e.g.* light-weight fences, full fences, control fences), which directly affect ordering, may be available on a given platform.

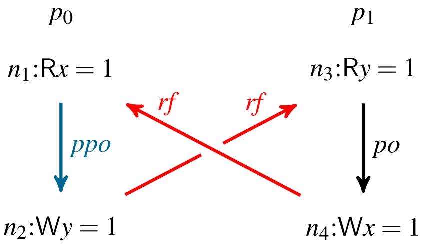
In contrast to writes, our semantics treats reads in a "strong," unbuffered way. Load buffering is conceptually more challenging than write buffering. Thinking operationally, dispatching an "asynchronous" write instruction is like "fire-and-forget." When executing an "asynchronous" read, however, the corresponding process continues regardless of whether the value it wishes to read has been obtained. This non-blocking nature is particularly problematic if it is assumed (as in our model) that reading is done *without* any synchronization. Subsequent code may *depend* on the value being read; the dependency may not only be a data-dependency (as the write to y in Figure 2.19a), but also a control flow dependency. Control flow depen-

p_0	p_1

`r1 := x;`
`y := r1;`
	`r2 := y;`
	`x := 1;`

$r_1 = 1, r_2 = 1$

(a) Litmus test



(b) Candidate execution

Figure 2.19: Load buffering (lb)

dency on values not yet available are common. When the reads are “synchronous,” these dependencies are not an issue: execution is stalled until the value is available. Difficulties emerge when the reads are “asynchronous;” in these cases, a decision has to be made regardless of whether the value is resolved. Only later, when the actual value is present, can the decision be revised. It could be the case that the decision is later deemed acceptable and execution continues as usual. It could also be the case that the branch decision leads to an impossibility, in which case execution needs to be back-tracked and an alternate path explored. It could also be the case that the branching decision is justified given a circular argument. As we will see next, circular reasoning is often deemed undesirable in a memory model.

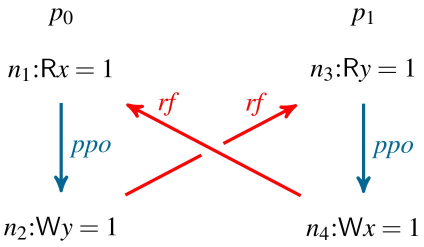
One important aspect in connection with load buffering is illustrated in Figure 2.20. It closely resembles the previous case from Figure 2.19. The crucial difference is an additional data dependency in p_1: the write statement has a data dependency on the preceding read event. This dependency is reflected in the graph by a $\rightarrow_{\mathsf{ppo}}$-edge, as opposed to a $\rightarrow_{\mathsf{po}}$-edge as in Figure 2.19b.

p_0	p_1

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

$r_1 = 1,\ \ r_2 = 1$ (out-of-thin-air)

(a) Litmus test

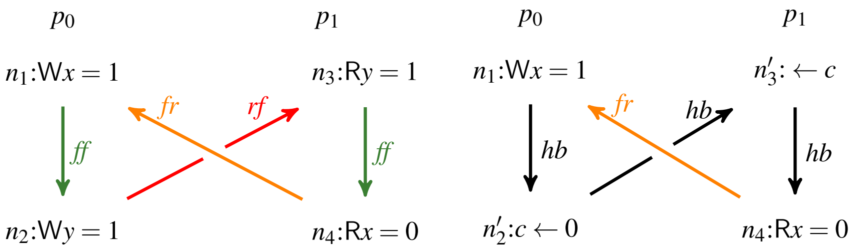


(b) Candidate execution

Figure 2.20: lb+ppos

The outcome $r_1 = 1 = r_2$ could be justified in that n_1 reads the value 1 written by n_4, subsequently used in the write n_2, which in turn is read by n_3, and used in the write event n_4 (see the candidate execution). This involves a circular argument and produces a value, the number 1, that does not even appear in the program text. Such behavior is termed "out-of-thin-air" and is generally, though not universally, considered illegal. In other words, the *candidate* graph of Figure 2.20b is ruled out by many memory models, for example [3]. Our operational semantics, given the absence of load buffering, also does not exhibit out-of-thin air behavior. Note, however, that in the informal happens-before Go memory model [40], out-of-thin-air behavior of this kind is allowed, as there are no statements or mechanisms which forbid the behavior. The Go model operates with the plain notion of program order $\rightarrow_{\mathsf{po}}$, stipulating that $\rightarrow_{\mathsf{po}} \subseteq \rightarrow_{\mathsf{hb}}$. Therefore, in the situation of Figure 2.20, with $\rightarrow_{\mathsf{po}}$ instead of $\rightarrow_{\mathsf{ppo}}$-edges, the out-of-thin-air observation is perfectly acceptable.[9]

Finally, we go back to the message passing pattern from Figure 2.17 to illustrate the role of channel communication. The assured ordering of the reads, resp. writes, represented by $\rightarrow_{\mathsf{ppo}}$-edges in Figure 2.17b can also be enforced by various fences. A properly synchronized message passing protocol would require, in many relaxed memory models, adding for example two full fences between the write resp. read instructions. These fences are shown as $\rightarrow_{\mathsf{ff}}$-edges in Figure 2.21a (cf. also [3]). The candidate execution illustrates the impossibility of the observation $r_1 = 1$ and $r_2 = 2$ to the litmus test from Figure 2.17a with added fences. Channel communication is the only synchronization primitive in our setting and, as we will see next, the effects of the fences can be achieved through sends and receives.



(a) mp+ffences

(b) mp+chan

Figure 2.21: mp with synchronization

In Figure 2.21b, p_0 updates the value of x, thereby *shadowing* its old value from p_0's local perspective. The thread then sends a message on a channel.[10] Since negative observability information (*i.e.,* a thread's shadow set) travels along channels,

[9] That is not to say that Go implementations will exhibit that behavior, just that it is consistent with the specification.

[10] At this point the reader may be wondering why write to x and then send a message on a channel

the receiving thread p_1 cannot read the stale value of x and will read the updated value 1 instead. The example also showcases how our model leads us to think about synchronization as *restriction on observability*. Rather than having write events percolating through a memory hierarchy composed of buffers and caches, in our semantics writes become visible immediately.

2.9 Limitations and future work

As seen, the semantics currently covers asynchronous writes, but only synchronous reads. Load buffering, however, accounts for an important form of relaxation that is present in many memory models, including that of Go. Therefore, the operation model presented here is less relaxed than the one of Go. We are currently working on adding read relaxation, which involves allowing control flow dependencies on read events "in-transit," meaning, branching on values that have not been retrieved from memory yet. Thus, the semantics will have a flavor of speculative execution similar to modern hardware. This will complicate the proof of conditional simulation.

On the other hand, our current model does support coRR behavior as illustrated in the example of Section 2.5.4 and discussed in Section 2.8. This is in line with the informal description of the Go memory model, even if coRR behavior may not be exhibited by actual Go compilers. The fact that the semantics allows for coRR behavior is not a problem to compiler writers. Complications can arise when the language semantics is more restrictive than the underlying architecture, but typically not the other way around. When emitting code to an architecture more relaxed than the language, the compiler must insert synchronization primitives in order to support its contract with the application programmer.

We believe that, once load buffering is incorporated, the augmented memory model will be lax enough to be a superset of Go's.[11] It will be interesting, then, to formally establish that relationship. As a mater of fact, one avenue of future work involves analyzing the proposed memory model as a basis for compiler verification. Similar to what CakeML is to ML [54], we envision the proposed semantics (once load buffering has been incorporated) as a high level specification with a chain of simulation relations towards more concrete operational semantics, all the way down to an actual compiler implementation.

instead of simply sending the value of x itself over the channel? In general, the shared resource may not be a single variable but a complex data structure. Take the example of a graphics pipeline with threads operating on a frame buffer. The buffer can reside in shared memory while threads coordinate the work by sending and receiving tokens on a channel.

[11] Perhaps the relation between Go's memory model and our operational semantics can be solidified by first translating the English specification to an axiomatic semantics and then proving a correspondence between this semantics and the operational one.

2.10 Related work

There are numerous proposals for and investigations of weak and relaxed memory models [1, 65, 3]. One widely followed approach, called *axiomatic*, specifies allowed behavior by defining various ordering relations on memory accesses and synchronizing events. Go's memory model [40] gives an informal impression of that style of specification. Less frequent are *operational* formalizations.

Boudol and Petri [17] investigate a relaxed memory model for a calculus with locks relying on concepts of rewriting theory. Unlike the presentation here, writes are buffered in a hierarchy of fifo-buffers reflecting the syntactic tree structure of configurations: immediately neighboring processors share one write buffer, neighbors syntactically further apart share a write buffer closer to the shared global memory located at the root. The position of a redex in the configuration is used as thread identifier and determines which buffers are shared. Consequently, parallel composition cannot be commutative and, therefore, terms cannot be interpreted up-to congruence $\equiv$ as in our case.

Zhang and Feng [104] use an abstract machine to operationally describe a happens-before memory model. Different from us, they make use of event *buffers*. Similar to us, they keep "older" write events to account for more than one observable variable value. The paper does not, however, deal with channel communication. Another operational semantics that uses histories of time-stamped, past read/write events is given by Kang et al. [52]. In this semantics, threads can promise future writes, and a reader acquires information on the writer's view of memory. Fences then synchronize global time-stamps on memory with thread-local information. Bisimulation proofs mechanized in Coq show the correctness of compilation to various architectures.

Pichon-Pharabod and Sewell [79] investigate an operational representation of a weak memory model that avoids problems of the axiomatic candidate-execution approach in addressing out-of-thin-air behavior. The semantics is studied in a calculus featuring locks as well as relaxed atomic and non-atomic memory accesses. Guerraoui et al. [45] introduce a "relaxed memory language" with an operational semantics to enable reasoning about various relaxed memory models. Their aim is to allow correctness arguments for software transactional memories implemented on weak-memory hardware. Another operational semantics is that of Flanagan and Freund [37], who present a weak memory model used as the basis for a race checker. The model is not as weak as the official Java Memory Model (JMM) but weaker than standard Java Virtual Machine implementations.

Much effort has been placed on Java and the JMM. In [62], Lochbihler points out how several features of Java, including dynamic memory allocation, thread spawns and joins, the wait-notify mechanisms, interruption, and infinite executions, interact in subtle ways with the language's memory model. Even though these features have been studied in their own right, Lochbihler's was the first paper to take their combined effect into account. Many of the complications analyzed in the paper arise from Java's security architecture. It has been known that security can be compromised when out-of-thin-air behavior is allowed. For example, out-of-thin-air may be leveraged to forge a pointer to `String's` underlying `char` array, which is assumed to be immutable for security reasons. Lochbihler shows, however, that security can be compromised by data races even after eliminating out-of-thin-air behavior. In contrast, the Go memory model does not preclude out-of-thin air behavior.

Demange et al. [26] formalize a weak semantics for Java using buffers. The semantics is quite less relaxed than the official JMM specification, the goal being to avoid the intricacies of the happens-before JMM and offer a firmer ground for reasoning. The model is defined axiomatically and operationally and the equivalence of the two formalizations is established. Jagadeesan et al. [49] present an operational semantics for a relaxed memory model for a concurrent, object-oriented language. The formalization is consistent with the official Java memory model JMM for data-race free programs. The semantics deviates from JMM though; it is weaker in that it allows more optimizations. Unlike our semantics, [49] allows *speculative* executions while at the same time still avoiding *out-of-thin* observations.

Alrahman et al. [4] formalize a relaxed total-store order memory model with fence and wait operations. They provide an implementation in Maude, a rewriting-based executable framework that precedes $\mathbb{K}$, and explore ways to mitigate state-space explosion. Lange et al. [58] define a small calculus, dubbed MiGo or mini-Go, featuring channels and thread creation. The formalization does not cover weak memory. Instead, the paper uses a behavioral effect type system to analyze channel communication.

2.11 Conclusion

This chapter presents an *operational* specification for a weak memory model with channel communication as the prime means of synchronization. In it, we prove the central guarantee that race-free programs behave sequentially consistently. The our semantics is accompanied by an implementation in the $\mathbb{K}$ framework and by several examples and test cases [29]. We plan to use the implementation towards the verification of program properties such as data-race freedom. Also, as the semantics is further relaxed, additional complications in the DRF-SC proof are likely to arise. At that point, we expect the implementation in $\mathbb{K}$ to help us manage the proof.

The current weak semantics remembers past write events as part of the run-time configuration, but does not remember read events. In Chapter 5 we present further relaxations to the model by treating read events similar to the representation of writes. This will allow us to accommodate load buffering behavior common to relaxed memory models, including that of Go.

3 Data-race detection and the Go language

Data races are often discussed in the context of lock acquisition and release, with race-detection algorithms routinely relying on *vector clocks* as a means of capturing the relative ordering of events from different threads. In this chapter, we present a data-race detector for a language with channel communication as its sole synchronization primitive, and provide a semantics directly tied to the *happens-before* relation, thus forging the notion of vector clocks.

3.1 Introduction

One way of dealing with complexity is by partitioning a system into cooperating subcomponents. When these subcomponents compete for resources, coordination becomes a prominent goal. One common programming paradigm is to have threads cooperating around a pool of shared memory. In this case, coordination involves, for example, avoiding conflicting accesses to memory. Two concurrent accesses constitute a *data race* if they reference the same memory location and at least one of the accesses is a write. Because data races can lead to counterintuitive behavior, it is important to detect them.

The problem of data-race detection in shared memory systems is well studied in the context of lock acquisition and release. When it comes to message passing, the problem of concurrent accesses to *channels*, in the absence of shared memory, is also well studied—the goal, in these cases, is to achieve determinism rather than race-freedom [24, 25, 97]. What is less prominent in the race-detection literature is the study of channel communication as the synchronization primitive for shared memory systems. In this chapter, we present exactly that; a dynamic data-race detector for a language in the style of Go, featuring channel communication as means of coordinating accesses to shared memory.

We fix the syntax of our calculus in Section 3.3 and present a corresponding operational semantics. The configurations of the semantics keep track of memory events (*i.e.,* of read and write accesses to shared variables) such that the semantics can be used to detect races. A proper book-keeping of events also involves tracking *happens-before* information. In the absence of a global clock, the happens-before relation is a vehicle for reasoning about the relative order of execution of different threads [56]. We describe the race detection task and present a framework, called GRACE [31], that is based on what we call *happens-before sets*. Different from other race detectors, which often employ vector clocks (VCs) as a mechanism for capturing the happen-before relation, we tie our formalization more closely to the concept of happens-before. The proposed approach, based on happens-before sets, allows for garbage collection of “stale” memory access information that would otherwise be tracked. Although, in the worst case, the proposed detector requires a larger footprint when compared to VC-based implementations, we conjecture the existence of a hybrid approach that can offer benefits from both worlds.

Our race detector is built upon the work of Chapter 2, where we formalize a weak memory model inspired by the Go specification [40]. The core of the work was a proof of the DRF-SC guarantee, meaning, we proved that the proposed relaxed memory model behaves Sequentially Consistently (SC) when running Data-Race Free (DRF) programs. The proof hinges on the fact that, in the absence of races, all threads agree on the contents of memory. The scaffolding used in the proof contains the ingredients for the race detector presented in this chapter. We should point out, however, that the operational semantics presented here and used for race detection is *not* a weak semantics.[1] Apart from the additional information for race detection, the semantics is “strong” in that it formalizes a memory guaranteeing *sequential consistency.* To focus on a form of strong memory is not a limitation. Since we have established that a corresponding weak semantics enjoys the crucial DRF-SC property [34], the strong and weak semantics agree up to the first encountered race condition. Given that even racy program behaves sequentially consistently up to the point in which the first data-race is encountered, a complete race detector can safely operate under the assumption of sequential consistency.

The remainder of the chapter is organized as follows. Section 3.2 presents background information on data races and synchronization via message passing that are directly related to the formalization of our approach to race detection. Section 3.3 formalizes race detection in the context of channel communication as sole synchronization mechanism. We turn our attention to the issue of efficiency in Section 3.4. Section 3.5 gives a detailed comparison of our algorithm and VC-based algorithms for the acquire-release semantics. Section 3.6 puts our work in the perspective of trace theory. Section 3.7 examines related work. Section 3.8 provides a conclusion and touches on future work.

[1] Note that while the mentioned semantics of Chapter 2 differs from the one presented here, both share some commonalities. Both representations are based on appropriately recording information of previous read and write events in their run-time configuration. In both versions, a crucial ingredient of the book-keeping is connecting events in happens-before relation. The purpose of the book-keeping of events, however, is different: in Chapter 2, the happens-before relation serves to operationally formalize the weak memory model (corresponding roughly to PSO) in the presence of channel communication. In the current chapter, the same relation serves to obtain a race detector. Both versions of the semantics are connected by the DRF-SC result, as mentioned.

3.2 Background

Read and write conflicts. Memory accesses conflict if they target the same location and at least one of the accesses is a *write*—there are no read-read conflicts. A data race constitutes of conflicting accesses that are unsynchronized.

Listing 3.1: Program with race condition. [40]

```
var a string

func main() {
        go func() { a = "hello" }()
        print(a)
}
```

Take the Go code of Listing 3.1 as an example. There, the main function invokes an anonymous function; this anonymous function sets the global variable "a" to "hello". Note, however, that the call is prepended with the keyword **go**. When this keyword is present in a function invocation, Go spawns a new thread (or goroutine), and the caller continues execution without waiting for the callee to return. The main and the anonymous functions access the same shared variable in a conflicting manner (*i.e.,* one of the accesses is a write). Since both the main and the anonymous functions run in parallel and no synchronization is used (as evidenced by the lack of channel communication), the two accesses are also concurrent. This allows us to conclude that this program has a race.

A data race *manifests* itself when an execution step is immediately followed by another and the two steps are conflicting. This definition is the closest one can get to a notion of simultaneity in an operational semantics, where memory interactions are modeled as instantaneous atomic steps. While manifest races are obvious and easy to account for, races in general can involve accesses that are arbitrarily far apart in a linear execution. A "memory-less" detector can fail to report races, for example non-manifest races, that could otherwise be flagged by more sophisticated race detectors. The ability to flag non-manifest data-races is correlated with the amount of information kept and the length in which this information is kept for. In general, recording more information and storing it for longer leads to higher degrees of "completeness" at the expense of higher run-time overheads.[2]

We break down the notions of read-write and write-write conflicts into a more fine-grained distinction. Inspired by the notion of data hazards in the computer architecture literature, we break down read-write conflicts into read-after-write (RaW) and write-after-read (WaR) conflicts. To keep consistent with this nomenclature, we

[2] It should go without saying that observing an execution as being race free is not enough to assert that the program is race free. Completeness can at best be expected with respect to alternative schedules or linearizations of a given execution.

refer to write-write conflicts as write-after-write (WaW).[3] Going back to the example in Listing 3.1, there are two possible executions: one in which the spawned goroutine writes "hello" to the shared variable after the main function prints it, and another execution in which the print occurs after the writing of the variable. The first execution illustrates a write-after-read race, while the first illustrates a read-after-write. Note that this example does not contain a write-after-write race.

We make the distinction between the detection of after-write races and the detection of write-after-read ones. As we will see in Section 3.3.3, the detection of after-write races can be done with little overhead. The detection of after-read, however, cannot.

When reading or writing a variable, it must be checked that conflicting accesses *happened-before* the current access. The check must take place from the perspective of the thread attempting the access. In other words, the question of whether an event occurred in the "definite past" (*i.e.,* whether an event is in happened-before relation with "now") is *thread-local*; threads can have different views on whether an event belongs to the past. This thread-local nature is less surprising than it may sound: if one threads executes two steps in sequence, the second step can safely assume that the first has taken effect; after all, that is what the programmer must have intended by sequentially composing instructions in the given program order. Such guarantees hold locally, which is to say that the semantics *respects program order within a thread*. It is possible, however, for steps to not take effect in program order. A compiler or hardware may rearrange instructions, and they often do so in practice. What must remain true is that these reorderings cannot be observable from the perspective of a single thread. When it comes to more than one thread, however, agreement on what constitutes the past cannot be achieved without synchronization. Synchronization and consensus are integrally related.[4] Specifically, given a thread t, events from a different thread t' are not in the past of t unless synchronization forces them to be.

Synchronization via bounded channels. In the calculus presented here, channel communication is the only way in which threads synchronize. Channels can be created dynamically and closed; they are also first-class data, which means channel identifiers can be passed as arguments, stored in variables, and sent over channels.

[3] The mentioned "temporal" ordering and the use of the word "after" refers to the occurrence of events in the trace or execution of the running program. It is incorrect to conflate the concept of happens-before with the ordering of occurrences in a trace. For instance, in a RaW situation, the read step occurs after a write in an execution, *i.e.,* the read is mentioned after the write in the linearization. This order of occurrence does *not* mean, however, that the read happens-after the write or, conversely, the write happens-before the read. Actually, for a RaW race (same as for the other kinds of races), the read occurs after the write but the accesses are *concurrent*, which means that they are *unordered* as far as the happens-before relation is concerned.

[4] In the context of channel communication and weak memory, the connection between synchronization and consensus is discussed in a precise manner in our previous work; see the *consensus lemmas* in Section 2.6.

Send and receive operations are central to synchronization. Clearly, a receive statement is synchronizing in that it is potentially blocking: a thread blocks when attempting to receive from an empty channel until, if ever, a value is made available by a sender. Since channels here are bounded, there is also potential for blocking when sending, namely, when attempting to send on a channel that is full.

We can use a channel c to eliminate the data race in Listing 3.1 as follows: the anonymous function sends a message to communicate that the shared variable has been set. Meanwhile, the main thread receives from the channel before printing the shared variable.

Listing 3.2: Repaired program.

```go
var a string
var c = make(chan bool, 1);

func main() {
        go func() { a = "hello";  c <- true }()
        <- c
        print(a)
}
```

The happens-before memory model stipulates, not surprisingly, a causal relationship between the communicating partners [40]:

A send on c happens-before the corresponding receive from c completes. (3.1)

Given that channels have finite capacity, a thread remains blocked when sending on a full channel until, if ever, another process frees a slot in the channel's buffer. In other words, the sender is blocked until another thread receives from the channel. Correspondingly, there is a happens-before relationship between a receive and a subsequent send on a channel with capacity k [40]:

The i^{th} receive from c happens-before the $(i+k)^{th}$ send on c completes. (3.2)

Interestingly, because of this rule, a causal connection is forged between the sender and *some previous* receiver who is otherwise unrelated to the current send operation. When multiple senders and receivers share a channel, rule (3.2) implies that it is possible for two threads to become related (via happens-before) without ever directly exchanging a message.[5]

The indirect relation between a sender and a prior receiver, postulated by rule (3.2), allows channels to be used as locks. In fact, free and taken binary

[5] Communication means sending a message to or receiving a message from a channel; messages are not addressed to or received from specific threads. Thus, sharing the channel by performing sends and receives does not necessarily make two threads "communication partners." Two threads are partners when one receives a message deposited by the other.

locks are analogous to empty and full channels of capacity one. A process takes and releases locks for the purpose of synchronization (such as assuring mutually exclusive access to shared data) without being aware of “synchronization partners.” In the (mis-)use of channels as locks, there is also no inter-process communication. Instead, a process “communicates” with itself: In a proper lock protocol, the process holding a lock (*i.e.,* having performed a send onto a channel) is the only one supposed to release the lock (*i.e.,* performing the corresponding receive). Thus, a process using a channel as lock receives its own previously sent message—there is no direct inter-process exchange. Note, however, synchronization still occurs: subsequent accesses to a critical region are denied by sending onto a channel and making it full. See Section 3.3.5.2 for a more technical elaboration.

To establish a happens-before relation between sends and receives, note the distinction, between a channel operation and its *completion* in the formulation of rules (3.1) and (3.2). The order of events in a concurrent system is partial; not only that, it is strictly partial since we don’t think of an event as happening-before itself. A strict partial order is an irreflexive, transitive, and asymmetric relation. In the case of synchronous channels, if we were to ignore the distinction between an event and its completion, according to rule (3.1), a send would then happen-before its corresponding receive, and, according to rule (3.2), the receive would happen-before the send. This cycle breaks asymmetry. Asymmetry can be repaired by interpreting a send/receive pair on a synchronous channel as a single operation; indeed, it can be interpreted as a *rendezvous*.

The distinction between a channel operation and its completion is arguably more impactful when it comes to buffered channels. For one, it prevents sends from being in happens-before with other sends, and receives from being in happens-before with other receives. To illustrate, let $\texttt{sd}^i$ and $\texttt{rv}^i$ represent the i^{th} send and receive on a channel. If we remove from rules (3.1) and (3.2) the distinction between an operation and its completion, the i^{th} receive would then happens-before the $(i+k)^{th}$ send—based on rule (3.2)—and the $(i+k)^{th}$ send would happens-before the $(i+k)^{th}$ receive—based on rule (3.1):

$$\texttt{rv}^i \rightarrow_{\textsf{hb}} \texttt{sd}^{i+k} \rightarrow_{\textsf{hb}} \texttt{rv}^{i+k}$$

By transitivity of the happens-before relation, we would then conclude that the i^{th} receive happens-before the $(i+k)^{th}$ receive, which would happen-before the $(i+2k)^{th}$ receive and so on. As a consequence, a receive operation would have a lingering effect through-out the execution of the program—similarly for send operations. This accumulation of effects can be counterintuitive for the application programmer, who would be forced to reason about arbitrarily long histories.

3.3 Data-race detection

We start in Section 3.3.1 by presenting the abstract syntax of our calculus and, in Section 3.3.2, an overview of the operational semantics used for data-race detection. The race detector itself is introduced incrementally. We start in Section 3.3.3 with a simple detector that has a small footprint but that is limited to detecting after-write races. We build onto this first iteration of the detector in Section 3.3.4, making it capable of detecting after-write as well as after-read races. The detector's operation is illustrated by examples in Section 3.3.5. Later, in Section 3.4, we turn to the issue of efficiency and introduce "garbage collection" as a means to reduce the detector's footprint. These race detectors can be seen as augmented versions of an underlying semantics without additional book-keeping related to race checking. This "undecorated" semantics, including the definition of internal steps and a notion of structural congruence, can be found in Section 2.4 .

3.3.1 A calculus with shared variables and channel communication

We formalize our ideas in terms of an idealized language shown in Figure 3.1 and inspired by the Go programming language. The syntax is basically unchanged from the previous chapter. *Values* v can be of two forms: r denotes local variables or registers; n is used to denote references or names in general and, in specific, p for processes or goroutines, m for memory events, and c for channel names. We do not explicitly list values such as the unit value, booleans, integers, etc. We also omit compound local expressions like $e_1 + e_2$. Shared variables are denoted by x, z, etc., `load` z represents reading the shared variable z into the thread, and $z := v$ denotes writing to z. References are dynamically created. A new channel is created by `make` $(\texttt{chan}\, T, v)$, where T represents the type of values carried by the channel and v a non-negative integer specifying the channel's capacity. Sending a value v over a channel c and receiving a value as input from a channel are denoted respectively as $c \leftarrow v$ and $\leftarrow c$. After the operation `close`, no further values can be sent on the specified channel. Attempting to send values on a closed channel leads to a panic.

$$
\begin{array}{rcll}
v & ::= & r \mid \underline{n} & \text{values} \\
e & ::= & t \mid v \mid \texttt{load}\, z \mid z := v \mid \texttt{go}\, t & \text{expressions} \\
 & \mid & \texttt{if}\, v\, \texttt{then}\, t\, \texttt{else}\, t & \\
 & \mid & \texttt{make}\, (\texttt{chan}\, T, v) \mid \leftarrow v \mid v \leftarrow v \mid \texttt{close}\, v & \\
g & ::= & v \leftarrow v \mid \leftarrow v \mid \texttt{default} & \text{guards} \\
t & ::= & \texttt{let}\, r = e\, \texttt{in}\, t \mid \sum_i \texttt{let}\, r_i = g_i\, \texttt{in}\, t_i & \text{threads}
\end{array}
$$

Figure 3.1: Abstract syntax

Starting a new asynchronous activity, called goroutine in Go, is done using the go-keyword. In Go, the go-statement is applied to function calls only. We omit function calls, asynchronous or otherwise, as they are orthogonal to the memory model's formalization. The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches (or communication clauses in Go-terminology). These branches act as guarded threads. General expressions in Go can serve as guards. Our syntax requires that only communication statements (*i.e.,* channel sending and receiving) and the `default`-keyword can serve as guards. This does not reduce expressivity and corresponds to an A-normal form representation [89]. At most one branch is guarded by `default` in each select-statement. The same channel can be mentioned in more than one guard. "Mixed choices" [77, 78] are also allowed, meaning that sending- and receiving-guards can both be used in the same select-statement. We use `stop` as syntactic sugar for the empty select statement; it represents a permanently blocked thread. The `stop`-thread is also the only way to syntactically "terminate" a thread, meaning that it is the only element of t without syntactic sub-terms.

The `let`-construct `let` $r = e$ `in` t combines sequential composition and scoping for local variables r. After evaluating e, the rest t is evaluated where the resulting value of e is handed over using r. The let-construct acts as a binder for variable r in t. When r does not occur free in t, `let` boils down to *sequential composition* and, therefore, is more conveniently written with a semicolon. See also Figure 2.3 on page 21 for syntactic sugar.

3.3.2 Overview of the operational semantics

To capture the notion of ordering of events between threads, an otherwise unadorned operational semantics, equation (2.5), is equipped with additional information: each thread and memory location tracks the events it is aware of as having happened-before—these events are stored in what we call the happens-before set, E_{hb}. Depending on the capabilities of the race detector, slightly different information is tracked as having happened-before (*i.e.,* stored in a happens-before set).

3.3.2.1 After-write races

When detecting after-write races (*i.e.,* RaW and WaW), in order to know whether a subsequent access to the same variable occurs without proper synchronization, one has to remember additional information concerning past write-events. Specifically, it must be checked that all write events to the same variable *happened-before* the current access. The happens-before set is then used to store information pertaining to write events; read events are not tracked. Also, terms representing a memory location have a different shape when compared to the undecorated semantics. In the undecorated semantics, the content v of a variable z is written as a pair $(\!| z{:=}v |\!)$. When after-write races come into play, it is not enough to store the last value written to each variable; we also need to identify write events associated with the variable. Thus, an entry in memory takes the form $(\!|E_{hb}, z{:=}v|\!)$ where E_{hb} holds identifiers m, m', etc. that uniquely identify write events to z—contrast the run-time configurations in equation (2.5), and (3.3). The number of prior write events that need to be tracked can be reduced for the sake of efficiency, in which case the term representing a memory location takes the form $m(\!|E^r_{hb}, z{:=}v|\!)$ where m is the identifier of the most recent write to z. See equation (3.4).

3.3.2.2 Write-after-read races

Besides the detailed coverage of RaW and WaW races in Section 3.3.3, we describe the detection of *write-after-read* races in Section 3.3.4. When it comes to WaR, the race checker needs to remember information about past reads in addition to past write events. Abstractly, a read event represents the fact that a load-statement has executed. Thus, the set E_{hb} of an entry $(\!|E_{hb}, z{:=}v|\!)$ in memory holds identifiers of both read and write events.

In the strong semantics, a read always observes one definite value which is the result of one particular write event. Therefore, the configuration contains entries of the form $m(\!|E^r_{hb}, z{:=}v|\!)$ where m is the identifier of the "last" write event and E^r_{hb} is a set of identifiers of read events, namely those that accumulated after m. Note that "records" of the form $m(\!|E^r_{hb}, z{:=}v|\!)$ can be seen as $n+1$ recorded events, one write event together with $n \geq 0$ read-events. This definition of records with one write per variable stands in contrast to a weak semantics, where many different write events may be observable by a given read [34].

3.3.2.3 Synchronization

Channel communication propagates happens-before information between threads, and thus, affects synchronization. In the operational rules, each channel c is actually realized with *two* channels, which we refer to as *forward*, c_f, and *backward*, c_b—see Figure 3.4. The forward part serves to communicate a value transmitted from a sender to a receiver; it also stipulates a causal relationship between the communicating partners [40]—see rule (3.1) of page 57. To capture this relationship in the context of race checking, the sender also communicates its current information about the happens-before relation to the receiver. The communication of happens-before information is accomplished by the transmission of E_{hb} over channels; see rule R-REC in Figure 3.4.

The memory model also stipulates a happens-before relationship between a receive and a subsequent send on a channel with capacity k—see rule (3.2) of page 57. While we refer to the *forward channel* as carrying a message from a sender to a receiver, the backward part of the channel is used to model the indirect connection between some prior receiver and a current sender; see R-SEND in Figure 3.4.

The interplay between forward and backward channels can also be understood as a form of flow control. Entries in the backward channel's queue are not values deposited by threads. Instead, they can be seen as tickets that grant senders a free slot in the communication channel, *i.e.,* the forward channel.[6] Thus, the number of "messages" in the backward channel capture the notion of fullness: a channel is full if the backward channel is empty. See rule R-SEND in Figure 3.4 or Figure 2.8 for the underlying semantics without race checking. When a channel of capacity k is created, the forward queue is empty and the backward queue is initialized so that it contains dummy elements $E_{hb\perp}$ (cf. rule R-MAKE). The dummy elements represent the number of empty or free slots in the channel. Upon creation, the number of dummy elements equals the capacity of the channel.

As discussed in Section 3.2, there is a distinction between a synchronization operation and its completion. A send/receive pair on a synchronous channel can be seen as a rendezvous operation; captured in our semantics by the R-REND reduction rule of Figure 3.4. When it comes to asynchronous communication, the distinction between a channel operation and its completion is handled by the fact that send and receive operations update a thread's local state but do not immediately transmit the updated state onto the channel—see rules R-SEND and R-REC in Figure 3.4.

3.3.3 Detecting read-after-write (RaW) and write-after-write (WaW) races

To detect "after-write" races, run-time configurations are given following syntax:

$$R ::= p\langle E_{hb}, t\rangle \mid (\!| E^z_{hb}, z{:=}v |\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R\,. \tag{3.3}$$

Configurations are considered up-to structural congruence, with the empty configuration $\bullet$ as neutral element and $\parallel$ as associative and commutative—similar to the configurations described on Sections 2.4 and 2.5.

In the configurations, a triple $(\!| E^z_{hb}, z{:=}v |\!)$ not only stores the current value of z but also records the unique identifiers m, m', etc of every write *event* to z in E^z_{hb}.[7] A

[6] In the case of lossy channels, backward channels are sometimes used for the purpose of error control and regulating message retransmissions, where the receiver of messages informs the sender about the successful or also non-successful reception of a message. Here, channels are assumed non-lossy and there is no need for error control. In that sense, the term "backward" should not be interpreted as communication *back* to the receiver in the form of an acknowledgment.

[7] We will later use the term "event" also when talking about histories or traces. There, events carry slightly different information. For instance, being interested in the question whether a history contains evidence of a race, it won't be necessary to mention the actual value being written in the write event in the history. Both notions of events, of course, hang closely together. It should be clear from the context whether we are referring to events as part of a linear history or recorded as part of the configuration. When being precise, we refer to a configuration event as *recorded* event. Since recorded events in the semantics are uniquely labeled, we also allow ourselves to use words like "event m" even if m is just the identifier for the recorded event $m(\!| z{:=}v |\!)$.

write to memory updates a variable's value and also generates a fresh identifier m. In order to record the write event, the tuple $(m, !z)$ is placed in the happens-before set of the term representing the memory location that has been written to. The initial configuration starts with one write-event per variable and the semantics maintains this uniqueness as an invariant. In effect, the collection of recorded write events behave as a mapping from variable to values.[8]

A thread t is represented as $p\langle E_{hb}, t\rangle$ at run-time, with p serving as identifier. To be able to determine whether a next action should be flagged as race or not, a goroutine keeps track of happens-before information corresponding to past write events. An event mentioned in E_{hb} is an event of the past, as opposed to being an event that simply occurred in a prior step. An event is "concurrent" if it occurred in a prior step but is not in happens-before relation with the current thread state. Concurrent memory events are potentially in conflict with a thread's next step. More precisely, if the memory record $(\!| E^z_{hb}, z{:=}v |\!)$ is part of the configuration, then it is safe for thread $p\langle E_{hb}, t\rangle$ to write to z if $E^z_{hb} \subseteq E_{hb}$. Otherwise, there exists a write to z that is not accounted for by thread p and a WaW conflict is raised. Similar when reading from a variable.

Data-races are marked as a transition to an exception E—see the derivation rules of Figure 3.3, and, when write-after-read races are considered, Figure 3.7. The exception takes as argument a set containing the prior memory operations that conflict and are concurrent with the attempted memory access.

$$\dfrac{E^z_{hb} \subseteq E_{hb} \qquad fresh(m') \qquad E'_{hb} = \{(m', !z)\} \cup E_{hb} \qquad E'^z_{hb} = \{(m', !z)\} \cup E^z_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to p\langle E'_{hb}, t\rangle \parallel (\!| E'^z_{hb}, z{:=}v' |\!)}\ \textsc{R-Write}$$

$$\dfrac{E^z_{hb} \subseteq E_{hb}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to p\langle E_{hb}, \texttt{let}\ r =\ v\ \texttt{in}\ t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!)}\ \textsc{R-Read}$$

Figure 3.2: Operational semantics augmented for RaW and WaW race detection

Goroutines synchronize via message passing, which means that channel communication must transfer happens-before information between goroutines. Suppose a goroutine p has just updated variable z thus generating the unique label m. The tuple $(m, !z)$ is placed in the happens-before set of both the thread p and the memory record associated with z. At this point, p is the only goroutine whose happens-before set contains the label m associated with this write-record. No other goroutine can read or write to z without causing a data-race. When p sends a message onto a channel, the information about m is also sent. Suppose now that a thread p' reads from the channel and receives the corresponding message before p makes any fur-

[8] The fact that memory behaves like a mapping is consistent with the strong memory assumption.

$$\frac{E^{z}_{hb} \not\subseteq E_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^{z}_{hb}, z{:=}v |\!) \to \mathsf{E}\left(E^{z}_{hb} - E_{hb}\right)}\ \text{R-WRITE-E}_{WaW}$$

$$\frac{E^{z}_{hb} \not\subseteq E_{hb}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!| E^{z}_{hb}, z{:=}v |\!) \to \mathsf{E}\left(E^{z}_{hb} - E_{hb}\right)}\ \text{R-READ-E}_{RaW}$$

Figure 3.3: Exception conditions for RaW and WaW data-race detection

ther modifications to z. The tuple $(m, !z)$ is added to p''s happens-before set, so both p and p' are aware of z's most recent write to z. The existence of m in both goroutine's happens-before sets implies that either p or p' are allowed to update z's value. The rules for channel communication are given in Figure 3.4. They will remain unchanged when we extend the treatment to RaW conflicts. The exchange of happens-before information via channel communication is also analogous to the treatment of the weak semantics in Chapter 2.

As Chapter 2, "the R-CLOSE rule closes both sync and async channels. Executing a receive on a *closed* channel results in receiving the end-of-transmission marker $\perp$ (cf. rule R-REC$_{\perp}$) and updating the local state E_{hb} in the same way as when receiving a properly sent value. The "value" $\perp$ is not removed from the queue, so that all clients attempting to receive from the closed channel obtain the communicated happens-before synchronization information."

Finally, goroutine creation is a synchronizing operation where the child, who is given a unique identifier p', inherits the happens-before set from the parent—see the R-GO rule in Figure 3.5.

3.3.4 Detecting write-after-read (WaR) races

In the previous section, the detection of read-after-write and write-after-write races required happens-before sets to contain write labels only. The detection of write-after-read races requires recording read labels, as well. A successful read of variable z causes a fresh read label, say m', to be generated. The pair $(m', ?z)$ is added to the reader's happens-before set as well as to the record associated with z in memory—see the rule R-READ of Figure 3.6.

In order for a write to memory to be successful, the writing thread must not only be aware of previous write events to a given shared variable, but must also account for all accumulated reads to the variable. A write-after-read data-race is raised when a write is attempted by a thread and the thread is unaware of some previous reads to z. In other words, there exist some read-label in the happens-before set associated with the variable's record, say $r \in E^{z}_{hb} \downarrow_{?}$, that is not in the thread's happen-before set, $r \notin E_{hb}$. The projection $\downarrow_{?}$ essentially filters out write

$$\frac{q = [E_{hb\perp}, \ldots, E_{hb\perp}] \qquad |q| = v \qquad \mathit{fresh}(c)}{p\langle E_{hb}, \texttt{let}\ r = \texttt{make}\ (\texttt{chan}\ T, v)\ \texttt{in}\ t\rangle \;\rightarrow\; \nu c\ (p\langle E_{hb}, \texttt{let}\ r = c\ \texttt{in}\ t\rangle \parallel c_f[] \parallel c_b[q])}\ \textsc{R-Make}$$

$$\frac{\neg \mathit{closed}(c_f[q_2]) \qquad E'_{hb} = E_{hb} + E''_{hb}}{c_b[q_1 :: E''_{hb}] \parallel p\langle E_{hb}, c \leftarrow v; t\rangle \parallel c_f[q_2] \;\rightarrow\; c_b[q_1] \parallel p\langle E'_{hb}, t\rangle \parallel c_f[(v, E_{hb}) :: q_2]}\ \textsc{R-Send}$$

$$\frac{v \neq \perp \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{llll} c_b[q_1] \parallel & p\langle E_{hb}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle & \parallel c_f[q_2 :: (v, E''_{hb})] & \rightarrow \\ c_b[E_{hb} :: q_1] \parallel & p\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t\rangle & \parallel c_f[q_2] & \end{array}}\ \textsc{R-Rec}$$

$$\frac{E'_{hb} = E_{hb} + E''_{hb}}{p\langle E_{hb}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \parallel c_f[(\perp, E''_{hb})] \;\rightarrow\; p\langle E'_{hb}, \texttt{let}\ r = \perp\ \texttt{in}\ t\rangle \parallel c_f[(\perp, E''_{hb})]}\ \textsc{R-Rec}_{\perp}$$

$$\frac{E'_{hb} = E_{hb1} + E_{hb2}}{\begin{array}{lllll} c_b[] \parallel & p_1\langle E_{hb1}, c \leftarrow v; t\rangle & \parallel p_2\langle E_{hb2}, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t_2\rangle & \parallel c_f[] & \rightarrow \\ c_b[] \parallel & p_1\langle E'_{hb}, t\rangle & \parallel p_2\langle E'_{hb}, \texttt{let}\ r = v\ \texttt{in}\ t_2\rangle & \parallel c_f[] & \end{array}}\ \textsc{R-Rend}$$

$$\frac{\neg \mathit{closed}(c_f[q])}{p\langle E_{hb}, \texttt{close}\ (c); t\rangle \parallel c_f[q] \;\rightarrow\; p\langle E_{hb}, t\rangle \parallel c_f[(\perp, E_{hb}) :: q]}\ \textsc{R-Close}$$

Figure 3.4: Operational semantics augmented for race detection: channel communication

$$\frac{\mathit{fresh}(p')}{p\langle E_{hb}, \texttt{go}\ t'; t\rangle \;\rightarrow\; \nu p'\ (p'\langle E_{hb}, t'\rangle) \parallel p\langle E_{hb}, t\rangle}\ \textsc{R-Go}$$

Figure 3.5: Operational semantics augmented for race detection: thread creation

events from the happens-before set. Under these circumstances, the precondition $E^z_{hb} \downarrow_? \not\subseteq E_{hb}$ of the R-WRITE-E$_{WaR}$ rule is met and a race is reported.

Compared to the detector of Section 3.3.3, rule R-WRITE-E$_{WaW}$ is augmented with the precondition $E^z_{hb} \downarrow_? \subseteq E_{hb}$. Without this precondition, there would be nondeterminism when reporting WaW and WaR races.[9] Note, however, that when both

[9] Consider the scenario in which p writes to and then reads from the shared variable z. Say the write to z generates a label w and the read generates r. If a thread p' attempts to write to z without first communicating with p, p' will not be aware of the prior read and write events. In other words, the happens-before set of p' will contain neither $(w, !z)$ nor $(r, ?z)$. Both rules R-WRITE-E$_{WaW}$ and R-WRITE-E$_{WaR}$ are

$$\frac{E^z_{hb} \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', !z)\} \cup E_{hb} \qquad E'^z_{hb} = \{(m', !z)\} \cup E^z_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb},\, z{:=}v |\!) \to p\langle E'_{hb}, t\rangle \parallel (\!| E'^z_{hb},\, z{:=}v' |\!)}\ \textsc{R-Write}$$

$$\frac{E^z_{hb} \downarrow_! \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', ?z)\} \cup E_{hb} \qquad E'^r_{hb} = \{(m', ?z)\} \cup E^r_{hb}}{p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!| E^z_{hb},\, z{:=}v |\!) \to p\langle E'_{hb}, \mathtt{let}\ r = \ v\ \mathtt{in}\ t\rangle \parallel (\!| E'^z_{hb},\, z{:=}v |\!)}\ \textsc{R-Read}$$

Figure 3.6: Operational semantics augmented for data-race detection

$$\frac{E^z_{hb} \not\subseteq E_{hb} \qquad E^z_{hb} \downarrow_? \subseteq E_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb},\, z{:=}v |\!) \to \mathsf{E}\left(E^z_{hb} - E_{hb}\right)}\ \textsc{R-Write-}\mathsf{E}_{WaW}$$

$$\frac{E^z_{hb} \downarrow_? \not\subseteq E_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb},\, z{:=}v |\!) \to \mathsf{E}\,(E^r_{hb} - E_{hb})}\ \textsc{R-Write-}\mathsf{E}_{WaR}$$

$$\frac{E^z_{hb} \downarrow_! \not\subseteq E_{hb}}{p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!| E^z_{hb},\, z{:=}v |\!) \to \mathsf{E}\left(E^z_{hb} - E_{hb}\right)}\ \textsc{R-Read-}\mathsf{E}_{RaW}$$

Figure 3.7: Exception conditions for WaR data-race detection

WaW and WaR apply, the read in the WaR race happens-after the write involved in the WaW race. We favor to resolve this non-determinism and to report the most recent conflict.

The detector presented here can flag all races: read-after-write, write-after-write, and write-after-read. In Section 3.4 we also make the detector efficient by "garbage collecting" stale information. But before then, let us look at a couple of examples that illustrate the detector's operation.

3.3.5 Examples

We will look at two examples of properly synchronized programs. The first is a typical usage of channel communication; one in which an action is placed in the past of another. The second example relies on mutual exclusion instead. In this case, we know that actions are not concurrent, but we cannot infer an order between

enabled in this case. However, the read happens-after the write that generated $(w, !z)$.

them. By contrasting the two examples in Section 3.3.5.3, we derive observations related to determinism and constructivism.

3.3.5.1 Message passing

Message passing, depicted in Figure 3.8, involves a producer writing to a shared variable and notifying another thread by sending a message onto a channel. A consumer receives from the channel and reads from the shared variable.

$$
\begin{array}{l}
p_1 \langle E_{hb1}, z := 42;\ c \leftarrow 0 \rangle \\
p_2 \langle E_{hb2}, \leftarrow c;\ \texttt{load}\ z \rangle
\end{array}
$$

Figure 3.8: Message passing example.

The access to the shared variable is properly synchronized. Given the operational semantics presented in this chapter, we can arrive at this conclusion as follows. A fresh label, say m, is generated when p_1 writes to z. The memory record involving z is updated with this fresh label, and the pair $(m, !z)$ is placed into p_1's happens-before set, thus yielding $E_{hb}{}'_1$. A send onto c sends not only the message value, 0 in this case, but also the happens-before set of the sender, $E_{hb}{}'_1$, see rule R-SEND. The act of receiving from c blocks until a message is available. When a message becomes available, the receiving thread receives not only a value but also the happens-before set of the sender at the time that the send took place, see rule R-REC. Thus, upon receiving from c, p_2's happens-before set is updated to contain $(m, !z)$. Receiving from the channel places the writing to z by p_1 into p_2's definite past. The race-checker makes sure of this fact by inspecting p_2's happens-before set when p_2 attempts to load from z. In other words, the race-checker checks that the current labels associated with z in the configuration are also present in the happens-before set of the thread performing the load.

The message passing example illustrates synchronization as imposing of an order between events belonging to different threads. The message places the producer's write in the past of the consumer's read. Next, we will look into an example in which synchronization is achieve via mutual exclusion. Two threads, p_1 and p_2, are competing to write to the same variable. We will not be able to determine which write happens-before the other. Even though we cannot infer the order, we can determine that a happens-before order exists and, therefore, that the program is properly synchronized.

3.3.5.2 Mutual exclusion

Figure 3.9 shows a typical mutual exclusion scenario. It involves two threads writing to a shared variable z. Before writing, a thread sends a message onto a channel c

which capacity $|c| = 1$. After writing, it receives from c.[10]

$$p_1\langle c \leftarrow 0;\ z := 17;\ \leftarrow c\rangle$$
$$p_2\langle c \leftarrow 0;\ z := 42;\ \leftarrow c\rangle$$

Figure 3.9: Mutual exclusion example.

A send and its corresponding receive do not directly contribute to synchronization in this example. The send is matched by a receive from the same thread; nothing new is learned from this exchange. To illustrate this point, which may come as a surprise, let us look at an execution. Say p_1 is the first to send 0 onto c. Then p_1's happens-before set E_{hb1} is placed onto the channel along with the value of 0. The thread then proceeds to write to z, which generates a fresh label, say m'; the pair $(m', !z)$ is placed on p_1's happens-before set. When receiving from c, p_1 does not learn anything new! It receives the message 0 and a "stale" happens-before set E_{hb1}. The receiver's happens-before set, $E_{hb}{}'_1$, is updated to incorporate the stale happens-before set, but this "update" causes no effective change:

$$\begin{aligned} E_{hb}{}'_1 \cup E_{hb1} &= (E_{hb1} \cup \{(m', !z)\}) \cup E_{hb1} \\ &= E_{hb1} \cup \{(m', !z)\} \\ &= E_{hb}{}'_1 \end{aligned}$$

The explanation for why the program is synchronized, in this case, is more subtle. It involves reasoning about the channel's capacity. Recall that, according to rule (3.2) on page 57, the i^{th} receive from a channel with capacity k happens before the $(i+k)^{th}$ send onto the channel completes. Since channel capacity is 1 in our example, rule (3.2) implies that the first receive from the channel happens-before the second send completes. If p_1 is the first to write to z, then p_1 is also the first to receive from c. Receiving from c places p_1's happens-before set onto the backward channel (see rule R-REC). This happens-before set contains the entry $(m', !z)$ registering p_1's write to z. Upon *sending* onto c, p_2 receives from the backward channel and learns of p_1's previous write. Thus, by the time p_2 writes to z, the write by p_1 has been-placed onto p_2's definite past. Since no concurrent accesses exist, the race checker does not flag this execution as racy.

[10] Note that the channel is being used as a semaphore [27]. Sending on the channel is analogous to a semaphore `wait` or `P` operation. Receive is analogous to `signal` or `V`. The `wait` decrements the value of the semaphore and, if the new value is negative, the process executing the `wait` is blocked. A `signal` increments the value of the semaphore variable, thus allowing another process (potentially coming from the pool of previously blocked processes) to resume. Similarly, a send operation decrements the number of available slots in the channel's queue, while a receive increments it. Sending on a channel with capacity 1 can only take place if the channel is empty; meaning, all previous sends are matched with a corresponding receive.

Similarly, p_2 could first send onto c and write to z. The argument for the proper synchronization of this alternate run would proceed in the same way. Therefore, even though it is not possible to infer who, among p_1 and p_2, writes to z first, we know that one of the writes is in a happens-before relation with the other. This knowledge is enough for us to conclude that the program is properly synchronized.

This example shows that channels are excessively powerful when it comes to implementing mutual exclusion, as evidenced by the fact that the forward queue associated with the channel is not utilized. When it comes to mutual exclusion, a more parsimonious synchronization mechanism suffices. Indeed, the *acquire* and *release* semantics associated with locks is a perfect fit. When acquiring a lock, a thread *learns* about the memory operations that precede the lock's release. In other words, memory operations preceding a lock's release are put in happens-before with respect to a thread that acquires the lock. Assuming a lock l starts with empty happens-before information, say $l[\emptyset]$, the rules ACQUIRE and RELEASE capture a lock's behavior.

$$\frac{E'_{hb} = E_{hb} \cup E''_{hb}}{p\langle E_{hb}, \texttt{acq}\ (l);t\rangle \parallel l[E''_{hb}] \ \rightarrow\ p\langle E'_{hb}, t\rangle \parallel l[]}\ \textsc{Acquire}$$

$$\frac{E'_{hb} = E_{hb} \cup E''_{hb}}{p\langle E_{hb}, \texttt{rel}\ (l);t\rangle \parallel l[] \ \rightarrow\ p\langle E_{hb}, t\rangle \parallel l[E'_{hb}]}\ \textsc{Release}$$

Note that an acquired lock, represented by $l[]$, cannot be re-acquired without a prior release, and that a released lock, meaning $l[E_{hb}]$, cannot re-released without a prior acquire.[11] While a thread's happens-before is updated on both sends and receives, with locks, only the acquisition updates a thread's happens-before information. Surrounding code with a call to acquire at the beginning and release at the end is sufficient for ensuring mutual exclusion. The full generality of channels is not required.

3.3.5.3 Determinism, confluence, and synchronization

In the message passing example of Section 3.3.5.1, we are able to give a constructive proof-sketch of the synchronization between p_1 and p_2; the "proof" puts an event from p_1 in the past of p_2. In the mutual exclusion example of Section 3.3.5.2, no such guarantee is possible. Instead, we give a non-constructive "proof" that p_1 and

[11] When releases are matched by an prior acquire from the same thread, then happens-before information accumulates monotonically, meaning, a thread learns about all previous releases, not just the most recently occurring one.

p_2 are synchronized by arguing that either p_1's actions are in the past of p_2's or vice versa. The *law of excluded middle* is used in this non-constructive argument.

The absence of constructivism is tied to the absence of determinism. While in the message passing example the program is deterministic, in the mutual exclusion example it is not. There is no *data* race in the mutual exclusion example, but there is still a "race" insofar as the two threads compete for access to a shared resource. The resource, in this case, is the channel, which is being used as a lock. The two threads race towards acquiring the lock (*i.e.,* sending onto the channel) first. The initial configuration has two transitions, one in which p_1 acquires the lock first and one in which p_2 does. These transitions are non-confluent.

When it comes to reasoning about programs that model hardware, the lack of constructivism and the non-confluence in the use of channels as locks is a hindrance. Deterministic languages and constructive logics are needed in order to rule out scenarios in which two logic gates attempt to drive the same *via* with different logic values (*i.e.,* a short circuit) [12]. In the case of channel communication and in the absence of shared memory, determinism can be achieved by enforcing ownership on channels; for example, by making sure a single thread can read and a single thread can write on a given channel at any given point in the execution [**?**]. It is possible for the ownership on channels to be passed around the threads in a way that preserves determinism [97].

The examples show that the absence of absence of data races is not enough to ensure determinism. In general, however, determinism is not a requirement. Many applications require "only" data-race freedom.

3.4 Efficient data-race detection

We have been gradually introducing a data-race checker. In Section 3.3.3, we presented a simple checker that flags after-write races (WaW and RaW) but is not equipped for write-after-read (WaR) detection. In Section 3.3.4, we augmented the detector to handle WaR. Here, we discuss how these detectors can be implemented efficiently; where efficiency is gained by employing "garbage collection" to reduce the detector's memory footprint. Note that keeping one record per variable is already a form of efficiency gain. In a relaxed memory model, since there may be more than one value associated with a variable at any point in the execution, one might keep one record per memory event [34]. The first step towards a smaller footprint is to realize that, if the underlying memory model supports the DRF-SC guarantee, a data-race detector can be built assuming sequential consistency. The reason being that, when a data race is flagged, execution stops at the point in which the weak and strong memory models' executions would diverge.

Knowing that memory events can overtake each other, in this section we discuss how stale or redundant information can be garbage collected. More precisely, we show how to garbage collect the data structures that hold happens-before information, that is, the thread-local happens-before set and the per-memory-location one.

3.4.1 Most recent write

Terms representing a memory location have taken different shapes when compared to the undecorated semantics. In the undecorated semantics, the content v of a variable z is written as a pair $(\!|z{:=}v|\!)$. For after-write race detection, an entry in memory took the form of $(\!|E_{hb}, z{:=}v|\!)$ with E_{hb} holding information about prior write events. Our first optimization comes from realizing that we do not need to keep a set of prior write events. We can record only the most recent write and still be able to flag all after-write racy *executions*. With this optimization, we may fail to report all *accesses* involved in the race, but we will still be able to report the execution as racy and to flag the most recent conflicting write event. This optimization is significant; it reduces the arbitrarily large set of prior write events to a single point.

An intuitive argument for the correctness of the optimization comes from noticing that a successful write to a variable can be interpreted as the writing thread taking *ownership* of the variable. Suppose a goroutine p has just updated variable z. At this point, p is the only goroutine whose happens-before set contains the label, say m, associated with this write-record. The placement of the new label into p's happens-before set can be seen as recording p's *ownership* of the variable: a data-race is flagged if any other thread attempts to read or write to z without first synchronizing with p—see the check $(m, !z) \in E_{hb}$ in the premise of the R-WRITE and R-READ rules of Figure 3.10.

When p sends a message onto a channel, the information about m is also sent. Suppose now that a thread p' reads from the channel and receives the corresponding message before p makes any further modifications to z. The tuple $(m, !z)$ containing the write-record's label is added to p''s happens-before set. Now both p and p' are aware of z's most recent write to z. The existence of m in both goroutine's happens-before sets imply that either p or p' are allowed to update z's value. We can think of the two goroutines as *sharing* z. Among p and p', whoever updates z first (re)gains the *exclusive* rights to z.

It may be worth making a parallel with hardware and cache coherence protocols. Given the derivation rules, we can write a race detector as a state machine. Compared to the Modified-Exclusive-Shared-Invalid protocol (MESI), our semantics does not have the *modified* state: all changes to a variable are immediately reflected in the configuration, there is no memory hierarchy in the memory model. As hinted above, the other states can be interpreted as follows: If the label of the most recent write to a variable is only recorded in one goroutine's happens-before set, then we can think of the goroutine as having *exclusive* rights to the variable. When a number of goroutines contain the pair $(m, !z)$ in their happen-before set with m being the label of the most recent write, then these goroutines can be thought to be *sharing* the variable. Other goroutines that are unaware of the most recent write can be said to hold *invalid* data.

3.4.2 Runtime configuration and memory related reduction rules

Given the "most recent write" optimization above, and, if we were satisfied with after-write conflicts, an entry in memory would take the form of $m(\!|z{:=}v|\!)$, with the label m uniquely identifying the event associated with v having been stored into z. Being able to flag after-write but not write-after-read races may be an adequate trade-off between completeness and efficiency. By not having to record read events, a simplified detector tailored for after-write race detection has a much smaller footprint than when read-after-write conflicts are also taken into account. Besides, a write-after-read race that is not flagged in an execution may realize itself as a read-after-write race in another run, and then be flagged by the simplified detector.[12]

In contrast, the detection of write-after-read races requires more book-keeping: we need read- in addition to write-labels. This addition is required because a WaR conflict can ensue between an attempted write and *any* previous unsynchronized read to the same variable. Therefore, the race-checker is made to remember all such potentially troublesome reads.[13] The runtime configuration is thus modified, this time as to contain entries of the form $m(\!|E^r_{hb}, z{:=}v|\!)$. The label m identifies of the most recent write event to z and the set E^r_{hb} holds-read event identifiers, namely, the identifiers of reads that accumulated after m.

$$R ::= p\langle E_{hb}, t\rangle \mid m(\!|E^r_{hb}, z{:=}v|\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R\,. \tag{3.4}$$

Note that *records* of the form $m(\!|E^r_{hb}, z{:=}v|\!)$ can be seen as $n+1$ recorded events: one write together with $n \geq 0$ read events.

The formal semantics maintains the following invariants. First, the happens-before information E^r_{hb} in $m(\!|E^r_{hb}, z{:=}v|\!)$ contains information of the form $(m', ?z)$ only, *i.e.,* there are no write events and all read-events concern variable z. Also, the event labels are unique for both reads and writes. In an abuse of notation, we may refer to m being in E^r_{hb} and write $m \in E^r_{hb}$ meaning, more precisely, $(m, ?z) \in E^r_{hb}$.

[12] Intuitively, say $S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} S_n$ is a run starting from an initial configuration S_0. Let $\bowtie$ be an independence relation on events, meaning, given $S_i \xrightarrow{e_i} S_{i+1} \xrightarrow{e_{i+1}} S_{i+2}$, we say that $e_i \bowtie e_{i+1}$ if there exist S' such that $S_i \xrightarrow{e_{i+1}} S' \xrightarrow{e_i} S_{i+2}$. The independence relation induces an equivalence relation on traces, namely, traces are equivalent if they can be derived from one another via the permutation of independent events. It can be shown that if $S_0 \xrightarrow{h} S_n$ is a run containing a write-after-read race, the exist an equivalent run in which the race materializes as a read-after-write race.

[13] Since depending on scheduling, a WaR data-race can manifest itself as RaW race, one option would be not add instrumentation for WaR race detection and, instead, hope to flag the RaW manifestation instead. Such practical consideration illustrates the trade-off between completeness versus run-time overhead.

$$\frac{(m, !z) \in E_{hb} \qquad E^{r}_{hb} \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow_z)}{p\langle E_{hb}, z := v'; t\rangle \parallel m(\!| E^{r}_{hb}, z{:=}v |\!) \rightarrow p\langle E'_{hb}, t\rangle \parallel m'(\!| \emptyset, z{:=}v' |\!)} \text{ R-WRITE}$$

$$\frac{(m, !z) \in E_{hb} \qquad \mathit{fresh}(m') \qquad \begin{array}{l} E'^{r}_{hb} = \{(m', ?z)\} \cup \left(E^{r}_{hb} - E_{hb} \downarrow_z\right) \\ E'_{hb} = \{(m', ?z)\} \cup (E_{hb} - E_{hb} \downarrow_z) \cup \{(m, !z)\} \end{array}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!| E^{r}_{hb}, z{:=}v |\!) \rightarrow p\langle E'_{hb}, \texttt{let}\ r = \ v\ \texttt{in}\ t\rangle \parallel m(\!| E'^{r}_{hb}, z{:=}v |\!)} \text{ R-READ}$$

Figure 3.10: Operational semantics augmented for efficient data-race detection

3.4.3 Garbage collection of happens-before sets

Knowledge of past events contained in a happens-before set E_{hb} is naturally monotonically increasing. For example, each time a goroutine learns about happens-before information, it adds to its pool of knowledge. In particular, events that are known to have "happened-before" cannot, by learning new information, become "concurrent." An efficient semantics, however, does not accumulate happens-before information indiscriminately; instead, it purges redundant information. We say "redundant" for the purpose of flagging racy executions, but leaving out conflicting accesses that have been overtaken by more recent memory events.

3.4.3.1 Garbage collection on *writes*

For a thread t to successfully write to z, all previously occurring accesses to z must be in happens-before with the thread's current state. One optimization comes from realizing that we can purge all information about prior accesses the variable z from the happens-before set of the writing thread t. We call these prior accesses *redundant* from the point of view of flagging racy executions. The reason for the correctness of this optimization is as follows: All future access of t to z are synchronized with the redundant accesses, after all, the accesses are recorded in t's happens-before set. Therefore, from the perspective of t, these accesses do not affect data-race detection. For the same reason, if a thread t' synchronizes with t, there is no race to report if and when t' accesses memory—the absence of these redundant accesses from t''s happens-before is, therefore, inconsequential. Finally, if t' does not synchronize with t, then an access to z is racy because it is unsynchronized with t's most recent write, regardless of the redundant prior accesses. Note that this optimization allows us to flag all racy *executions* even if we fail to report some of the accesses involved in the race.

Rule R-WRITE of Figure 3.10 embodies this discussion. Before writing, the rule checks that the attempted write happens-after all previously occurring accesses to z. This check is done by two premises: premise $(m,!z) \in E_{hb}$ makes sure that the most recent write to z, namely, the one that produced event $(m,!z)$, is in happens-before with the current thread state E_{hb}. As per discussion in Section 3.4.1, being synchronized with the most recent write means the thread is synchronized with all writes up to that point in the execution. The other premise, $E^r_{hb} \subseteq E_{hb}$, makes sure that the attempted write is in happens-after read accesses to z. If these two premises are satisfied, the write can proceed and prior accesses to z are garbage collected from the point of view of t. The filtering of redundant accesses is done by subtracting $E_{hb} \downarrow_z$ in

$$E'_{hb} = \{(m',!z)\} \cup (E_{hb} - E_{hb} \downarrow_z)$$

where $\downarrow_z$ projects the happens-before set down to operations on variable z. Finally, the write rule also garbage collects the in-memory record E^r_{hb} by setting it to $\emptyset$,[14] meaning that no read event have accumulated after the write yet.

3.4.3.2 Garbage collection on *reads*

We also garbage collect on load operations. Say t reads from z, thus generating event $(m',?z)$. Let us call *redundant* the memory accesses to z in t's happens-before set at the time event $(m',?z)$ takes place, with the exception of $(m,!z)$. A read operation can only conflict with a future write; there are not read-read conflicts. For a future write to take place, the writing thread will need to synchronize with a thread that "knows" about the read m'.[15] Any thread that knows of m' would also know about the redundant access to z and know of $(m,!z)$. In other words, m' and m subsume all happened-before accesses of z from the perspective of t. Therefore, we can garbage collect all such accesses by filtering them out of the thread's happen-before set, as in

$$E'_{hb} = \{(m',?z)\} \cup (E_{hb} - E_{hb} \downarrow_z) \cup \{(m,!z)\}.$$

These redundant accesses are also filtered out of the in-memory happens-before set:

$$E'^r_{hb} = \{(m',?z)\} \cup (E^r_{hb} - E_{hb} \downarrow_z).$$

[14] As per discussion in Section 3.4.1, a term representing a memory location $m(\!|E^r_{hb}, z{:=}v|\!)$ records in E^r_{hb} all the reads to z that have accumulated after the write that generated the write label m. When a new write m' of value $z := v'$ ensues, we update the memory term to record this new write and we reset its corresponding E^r_{hb} to $\emptyset$.

[15] "Knowing about the read m'" is a necessary condition for a thread to successfully write to z, but it is not a sufficient one. There may exist other reads, say m'', m''', etc that are concurrent with m'. A thread needs to synchronize with all such concurrent reads before it can successfully write to z.

3.4.3.3 Off-line garbage collection and channel communication

The garbage collector rules of Figure 3.11 can be run non-deterministically during the execution of a program. Rule R-GC eliminates stale entries from the happens-before set of a thread. It can be sensible to perform garbage collection also *after* a thread interacts with a channel, as happens-before information communicated via channels are likely to become stale. For example, suppose a thread, whose happens-before set does not contain stale entries, sends onto a channel and continues executing. By the time a receive takes place, the happens-before set transmitted via the channel may have become stale. Similarly for happens-before transmitted between receives and prior sends via the backward channel. Alternatively, we may choose an implementation in which the happens-before of in-flight messages are also garbage collected, in which case we would process the happens-before sets in a channel's forward and backward queues.

$$\frac{\begin{aligned} E'_{hb} = E_{hb} \quad & - \{(\hat{m}, !z) \mid (\hat{m}, !z) \in E_{hb} \ \wedge\ \hat{m} \neq m\} \\ & - \{(\hat{m}, ?z) \mid (\hat{m}, ?z) \in E_{hb} \ \wedge\ (\hat{m}, ?z) \notin E^r_{hb}\} \end{aligned}}{p\langle E_{hb}, t\rangle \parallel m(\!|E^r_{hb}, z{:=}v|\!) \to p\langle E'_{hb}, t\rangle \parallel m(\!|E^r_{hb}, z{:=}v|\!)}\ \text{R-GC}$$

Figure 3.11: Off-line garbage collection

3.5 Comparison with vector-clock based race detection

Vector clocks (VCs) are a mechanism for capturing the happen-before relation over events emanating from a program's execution [67]. A *vector clock* V is a function `Tid`→`Nat` which records a clock, represented by a natural number, for each thread in the system. "VCs are partially-ordered ($\sqsubseteq$) in a pointwise manner, with an associated join operation ($\sqcup$) and minimal element ($\bot_V$). In addition, the helper function inc_t increments the t-component of a VC" [36].

$$\begin{aligned} \mathrm{V}_1 \sqsubseteq \mathrm{V}_2 \ \text{ iff } & \forall t.\ \mathrm{V}_1(t) \leq \mathrm{V}_2(t) \\ \mathrm{V}_1 \sqcup \mathrm{V}_2 \ &= \ \lambda t.\ max(\mathrm{V}_1(t), \mathrm{V}_2(t)) \\ \bot_V \ &= \ \lambda t.\ 0 \\ inc_t(\mathrm{V}) \ &= \ \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ \mathrm{V}(u) + 1\ \texttt{else}\ \mathrm{V}(u) \end{aligned}$$

Using vector clocks, Pozniansky and Schuster [80] proposed a data-race detection algorithm referred to as DJIT$^+$. Their algorithm works as follows. Each thread t is associated with a vector clock C_t. Entry $\mathrm{C}_t(t)$ stores the current time at t, while

$\mathrm{C}_t(u)$ for $u \neq t$ keeps track of the time of the last operation "known" to t as having been performed by thread u.

The algorithm also keeps track of memory operations. Each memory location x has two vector clocks, one associated with reads, R_x, and another with writes, W_x. The clock of he last read from variable x by thread t is recorded in $\mathrm{R}_x(t)$; similar for $\mathrm{W}_x(t)$ and writes to x by t. When it comes to reading from memory, a race is flagged when a thread t attempts to read from x while being "unaware" of some write to x. Precisely, a race is flagged when t attempts to read from x and there exists a write to x by thread u, $\mathrm{W}_x(u)$, that is not accounted for by t, meaning $\mathrm{W}_x(u) \geq \mathrm{C}_t(u)$, or, equivalently, $\mathrm{W}_x \not\sqsubseteq \mathrm{C}_t$. If t succeeds in reading from x, then $\mathrm{R}_x(t)$ is updated to the value of $\mathrm{C}_t(t)$. When it comes to writing to memory, a race is flagged when t attempts to write to x while being unaware of some read or write to x, meaning either $\mathrm{R}_x \not\sqsubseteq \mathrm{C}_t$ or $\mathrm{W}_x \not\sqsubseteq \mathrm{C}_t$. If t succeeds in writing to x, then $\mathrm{W}_x(t)$ is updated to $\mathrm{C}_t(t)$.

A thread's clock is advanced when the thread executes synchronization operations, which have bearing on the happens-before relation. The algorithm was proposed in the setting of locks; each lock m is associated with a vector clock L_m. When a thread t acquires m, then C_t is updated to $\mathrm{C}_t \sqcup \mathrm{L}_m$. Acquiring a lock is analogous to receiving from a channel with buffer size one: the receiving thread updates its vector clock by incorporating the VC previously "stored" in the lock. When a thread t releases a lock m, the vector clock L_m is updated to C_t and thread's clock is advanced, meaning $\mathrm{C}_t := inc_t(\mathrm{C}_t)$. We can think of lock release as placing a message, namely the vector clock associated with the releasing thread, into a buffer of size one. Thus, in comparison with the approach presented here, lock operations are a special case of buffered channel communication. Our approach deals with channels of arbitrary size and their capacity limitations.

A significant difference between our approach and DJIT$^+$ is that we dispense with the notion of vector clocks. Vector clocks are a conceptual vehicle to capturing partial order of events. Instead of relying on VCs, our formalization is tied directly to the concept of happens-before. Vector clocks are expensive. VCs require $O(\tau)$ storage space and common operations on VCs consume $O(\tau)$ time where τ is the number of entries in the vector. In the case of race detection, τ is the number of threads spawn during the execution of a program. It turns out that not all uses of VCs in DJIT$^+$ are strictly necessary. In fact, Flanagan and Freund [36] introduce the concept of *epoch*, which consists of a pair $c@t$ where c is a clock and t a thread identifier. They then replace W_x, the vector clock tracking writes to x, with a single epoch. This epoch captures the clock and thread identity associated with the most recent write to x. Similarly, in our approach, a memory location is associated with the identifier of only the most-recent write to that location. Any thread who is "aware" of this identifier is allowed to read from the corresponding variable.

FASTTRACK also reduces the dependency on vector clocks by replacing R_x with the epoch of the most recent read to x. However, since reads are not totally ordered, FASTTRACK dynamically switches back to a vector clock representation when needed. Similar to FASTTRACK, we record the most recent (unordered) reads which, in the best case, involves an $O(1)$-memory footprint and $O(\tau)$ at the worst.

When it comes to per-thread memory consumption, however, our approaches look very different. While DJIT$^+$'s and FASTTRACK's worst-case memory consumption per thread is $O(\tau)$, our is $O(\nu\tau)$ where ν is the number of shared variables in a program.[16] Vector clocks' memory efficiency, when compared to happens-before sets, come from VC's ability to succinctly capture the per-thread accesses that take place in between advances of a clock. A thread's clock is advanced when the thread releases a lock.[17] All accesses made by a thread t in a given clock c are captured by the clock: if another thread u "knows" the value c of t's clock, then u is in happens-after with *all* accesses made by t—that is, all accesses up to when t's clock was advanced to $c+1$. In contrast, the happens-before set representation is much more coarse. We keep track of individual accesses, as opposed to lumping them together into a clock number. This coarseness explains the extra factor of ν in the worst-case analysis of the happens-before set solution. Although being a disadvantage in the worst case scenario, it does provide benefits, as we discuss next.

Note that the vector clocks associated with threads and locks grow monotonically. By *growing monotonically* we do not mean that time marches forward to increasing clock values. Instead, we mean that the number of clocks in a vector grows without provisions for the removal of entries. This growth can lead to the accumulation of "stale" information, where by stale we mean information that is not useful from the point of view of race detection. This growth stands in contrast to our approach to garbage collection. Stale information is purged from happens-before sets, which means they can shrink back to size zero after having grown in size.

3.5.1 Stale information in the vector clock setting

We have explored the notion of stale race-detection information in the setting of happens-before sets and have hinted at how stale information also exists within vector clocks. Here we define what a stale VC entry is. Similar to HB-sets, a vector-clock entry is stale if the entry does not impact a race detector's judgment of a program and its execution. Such entries can thus be purged from VCs associated with threads and locks.

[16] We believe the worst case is a degenerate case unlikely to happen: it involves every thread reading from every shared variable and then exchanging messages as to inform everyone else about their read events.

[17] If channels were used instead of locks, the advance would take place when a thread sends onto or receives from a channel.

Definition 10. *(Stale VC entry) A vector clock entry $c@t$ belonging to a thread or lock is stale if, for all shared variables x,*

$$\begin{aligned} \mathrm{R}_x(t) \neq 0 \rightarrow c < \mathrm{R}_x(t) \qquad &\textit{and}, \\ \mathrm{W}_x(t) \neq 0 \rightarrow c < \mathrm{W}_x(t). \end{aligned}$$

Given a sound and complete race detector, a memory access will either succeed (if properly synchronized) or fail (in the case of a race) regardless of whether a stale entry is contained in the vector clock of the accessing thread. Stale entries can be removed from threads' and locks' VCs—potentially alleviating the memory pressure associated with race detection—without impacting the data-race detector's outcome.

Proof. When proving that a stale entry is irrelevant when it comes to data-race detection, let u be a thread attempting to access variable x and $c@t$ be a stale entry in C_u. We break the access into the following cases:

Case: The thread attempts a read access that is flagged as racy
Since the read is flagged as racy, there exists t' such that $\mathrm{C}_u(t') < \mathrm{W}_x(t')$. If $t = t'$, then removing $c@t$ from C_u does not change the inequality, which means the read is flagged as racy regardless of $c@t$ being contained in the thread's vector clock. If $t \neq t'$, then the access is flagged as racy irrespective of the clock value associated with t in C_u.

Case: The access is a read (not flagged as racy)
Since the read succeeds, then for all t', $\mathrm{W}_x(t') \leq \mathrm{C}_u(t')$. If $\mathrm{W}_x(t) = 0$, then $c@t$ does not gate the access, meaning that the access would have succeeded even if $c@t$ was not present in C_u. The case in which $\mathrm{W}_x(t) \neq 0$ leads to the following contradiction, meaning that the access cannot be gated by a stale entry:

- $\mathrm{C}_u(t) < \mathrm{W}_x(t)$ by the staleness of $c@t$ and $W_x(t) \neq 0$, and
- $\mathrm{W}_x(t) \leq \mathrm{C}_u(t)$ from the fact that the access succeeded.

Case: Write access
Write accesses (racy and non-racy) are handled similar to read accesses. The difference with write accesses is that they require inspecting R_x in addition to W_x.

□

Although stale entries can be removed from the vector clocks associated with threads and locks, it is impractical to traverse all shared memory in order to identify stale entries. Defining an algorithm that alleviates memory pressure without compromising runtime is an open research question.

Let us look at an example that illustrates the difference in treatment of stale information between current VC based algorithms and our detection mechanism based on happens-before set. Consider the producer/consumer paradigm, where a thread produces information to be consumed by other threads. Say p_0 produces information by writing to the shared variable z. The thread then notifies consumers, p_1 and p_2, by sending a message on channel c. The consumers read from z and signal the fact that they are done consuming by sending onto channel d. The producer p_0 writes to z again once it has received the consumers' messages.

Producer	Consumers	
p_0	p_1	p_2
$z := 42;$	$\leftarrow c;$	$\leftarrow c;$
$c \leftarrow 0;\ c \leftarrow 0;$	$\texttt{load}\ z;$	$\texttt{load}\ z;$
$\leftarrow d;\ \leftarrow d;$	$d \leftarrow 0$	$d \leftarrow 0$
$z := 43$		

Let us run this example against a prototype implementation [31] of our proposed race detector, called GRACE, and against FASTTRACK. Consider the point in the execution after p_0 has written to z, the consumers have read from z and notified p_0, and p_0 is about to write to z again. Below is the state of the detectors at this point. The information contained in the happens-before sets and the vector clocks is very similar. There are three entries for p_0, and two entries for p_1 and p_2 each.

Happens-before sets	Vector clocks
$E_{hb}^{p_0} = \{(m_0, !z), (m_1, ?z), (m_2, ?z)\}$	$\mathrm{C}_{p_0} = \bot[p_0 \mapsto 6, p_1 \mapsto 1, p_2 \mapsto 1]$
$E_{hb}^{p_1} = \{(m_0, !z), (m_1, ?z)\}$	$\mathrm{C}_{p_1} = \bot[p_0 \mapsto 2, p_1 \mapsto 2]$
$E_{hb}^{p_2} = \{(m_0, !z), (m_2, ?z)\}$	$\mathrm{C}_{p_2} = \bot[p_0 \mapsto 3, p_2 \mapsto 2]$

The happens-before set $E_{hb}^{p_0}$ show the reads by p_1 and p_2 as being in happens-before with respect to p_0, along with p_0's own write to z. It also shows p_1 and p_2 as being "aware" of p_0's write to z, as well as being "aware" of their own reads to z. The same information is captured by the vector clocks. Recall that the bottom clock, $\bot$, maps every process-id to the clock value of 0. Thus, the VC associated with p_0 contains p_0's clock (which happens to be 6) as well as the clock associated with the reads to z by p_1 and p_2. In this execution, p_0's clock was 2 when the thread wrote to z. Thus, the entry $p_0 \mapsto 2$ in C_{p_1} and the entry $p_0 \mapsto 3$ in C_{p_2} place the write to z by p_0 in p_1's and p_2's past.

The difference between our approach the VC based approach is evidenced in the next step of execution, when p_0 writes to z for the second time. This write subsumes all previous memory interactions on z. In other words, this write is in happens-after with respect to all reads and writes to z up to this point in the execution of the program. Therefore, it is sufficient for a thread to synchronize with p_0 before issuing a new read or write to z; also, it is no longer necessary to remember the original write to z and the reads from z by p_1 and p_2. Here are the happens-before sets and vector clocks in the next step of execution, meaning, after p_0 writes to z the second time:

Happens-before sets	Vector clocks
$E^{p_0}_{hb} = \{(m_3, !z)\}$	$\mathrm{C}_{p_0} = \bot[p_0 \mapsto 6, p_1 \mapsto 1, p_2 \mapsto 1]$
$E^{p_1}_{hb} = \{\}$	$\mathrm{C}_{p_1} = \bot[p_0 \mapsto 2, p_1 \mapsto 2]$
$E^{p_0}_{hb} = \{\}$	$\mathrm{C}_{p_2} = \bot[p_0 \mapsto 3, p_2 \mapsto 2]$

The happens-before sets are mostly empty; the only entry corresponds to the most recent write to z, which is known to p_0. Meanwhile, the vector clocks are unchanged. Note, however, that every entry with the exception of $p_0 \mapsto 6$ in C_{p_0} is stale. In other words, with the exception of $p_0 \mapsto 6$, the presence or absence of all other entries does not alter a thread's behavior. To illustrate this point, take entry $p_0 \mapsto 2$ in C_{p_1} as an example: if p_1 were to attempt to access z, a data race will ensue regardless of whether or not the entry $p_0 \mapsto 2$ is in p_1's vector clock. Therefore, ideally, we would want these stale entries purged from the vector clocks of p_0, p_1, and p_2. Concretely, we would want $\mathrm{C}_{p_0} = \bot[p_0 \mapsto 6]$ and $\mathrm{C}_{p_1} = \mathrm{C}_{p_2} = \bot$.

Similar unbounded growth occurs in the VCs associated with locks,[18] thus also leading to the accumulation of stale information. We conjecture that an approach that purges stale information from VCs, similar to our notion of garbage collection, would be highly beneficial. VC-based implementations are very efficient in managing the memory overhead associated with variables. For example TSan, a popular race-detection library based on vector clocks that comes with the Go tool chain, stores one write and a small number of reads per memory location (the number of reads stored is 4 in the current implementation) [44]. Capping the number of tracked read events leads to false negatives; the cap a fair compromise between *recall* and memory consumption. In order to further reduce the memory footprint of modern race detection implementations, we are thus left with devising approaches to managing threads' and locks' memory overhead.

Unfortunately, reducing memory pressure on vector clocks associated with threads and locks is arguably more difficult than reducing memory pressure on VCs associated with shared variables. On the one hand, if a variable does not "remember" a read or write to itself as having happened-before, then the variable becomes more permissive from the point of view of race detection; meaning, more threads would be able to interact with this variable without raising a data-race, even when races should have been reported. On the other hand, if a *thread* "forgets" about some prior read or write access that have taken place on a variable, a spurious

[18] The *acquire* grows the VC associated with the acquiring thread; the *release* sets the VC of the corresponding lock to the VC of the acquiring thread.

data race may be raised. Thus, while dropping clock entries in the VCs associated with variables can introduce false negatives, dropping clock entries from VCs associated with threads and locks introduce false positives. From a practical perspective, false negatives are acceptable and can even be mitigated,[19] however, being warned of non-existing races is overwhelming to the application programmer, which means false positives are generally not tolerated.

3.6 Connections with trace theory

Our operational semantics mimics the Go memory model in defining synchronization in terms of channel communication. Specifically, we abide by rules (3.1) and (3.2), which establish a happens-before relation between a send and the completion of its corresponding receive, and, due to the boundedness of channels, between a receive and the completion of a future send. However, these are not the only imposition by the semantics on the order of events. Channels act as FIFO queues in both Go [19] as well as in our operational semantics. However, neither Go nor our operational semantics establish a happens-before relation between consecutive sends or consecutive receives. For example, the i^{th} send on a channel c does not happens-before the $(i+1)^{th}$ send on c. Therefore, there exist events that are necessarily ordered, but that are not in happens-before relation.

It is tempting to think of happens-before in terms of observations, where a and b are in happens-before if and only if we observe a followed by b, and never the other way around. This intuition is captured by the following tentative definition:

Let `idx` (a,h) *be the index of event* a *in a run* h*. Given the set of runs* H *starting from an initial configuration, we say that event* a happens-before b *if-and-only-if, for all runs* $h \in H$ *such that* $a,b \in h$*,* `idx` $(a,h) <$`idx` (b,h).

When it comes to weak memory systems, there exist events that are ordered according to the above tentative definition but that are not in happens-before relation. Take the improperly synchronized message-passing example of Figure 3.12 as an example. In this example, a thread p_0 writes to a shared variable `z` and sets a flag; another thread, p_1, checks the flag reads from `z` if the flag has been set.

p_0	p_1
`z    := 42;` (A) `done := true;` (B)	`r = load done;` (C) `if r then` `    load z` (D)

Figure 3.12: Message passing example.

[19] Provided we run a program enough times, we can randomly evict entries from a VC or happens-before set associated with a variable such that we eventually flag all existing races of the program.

If A and B are the first and second instructions in thread p_0, and C and D are the loads of the flag and of the shared variable z in p_1, then program order gives rise to $A \rightarrow_{\mathsf{hb}} B$ and $C \rightarrow_{\mathsf{hb}} D$. We also have that the load of z in D only occurs if the value of the flag observed by thread p_1 is `true`, which means it was previously set by thread p_0 in B. Therefore, in all runs in which D is observed, B necessarily occurs earlier in the execution. This necessity does not, however, place B and D in happens-before relation. Under many flavors of weak memory, the memory accesses between the two threads are not synchronized. As the example shows, our tentative definition of happens-before as always-occurring-before or necessarily-occurring-before does not work for weak memory systems. How about for sequential consistent ones?

In the program of Figure 3.13, thread p_0 sends values 0 and 1 into channel c consecutively. Concurrently, thread p_1 writes 42 to a shared variable z and receives from the channel, while thread p_2 first receives from the channel and conditionally reads from z. From this program, we construct an example in which events are necessarily ordered but are not in happens-before—even if we assume sequential consistency. To illustrate this point, let us consider an execution of the program. Let $(o)_p$ be a trace event capturing the execution of operation o by threads p. Let also $z!$ and $z?$ represent a write and read operation on the shared variable z, and $\mathtt{sd}\ \ c$ and $\mathtt{rv}\ \ c$ represent send and receive operations on channel c. Assuming channel capacity $|c| \geq 2$, the sequence below is a possible trace obtained from the execution of the program. Note that the if-statement's reduction is interpreted as an internal or silent transition:

$$\begin{array}{l}
p_0\langle c \leftarrow 0; c \leftarrow 1\rangle \\
p_1\langle z := 42;\ \leftarrow c\rangle \\
p_2\langle \mathtt{let}\ r := \leftarrow c\ \ \mathtt{in}\ \mathtt{if}\ r = 1\ \ \mathtt{then}\ \mathtt{load}\ z\rangle
\end{array}$$

Figure 3.13: Conditional race example.

$$(\mathtt{sd}\ \ c)_{p_0}\ \ (\mathtt{sd}\ \ c)_{p_0}\ \ (z!)_{p_1}\ \ (\mathtt{rv}\ \ c)_{p_1}\ \ (\mathtt{rv}\ \ c)_{p_2}\ \ (z?)_{p_2} \tag{3.5}$$

Given that p_1 receives from c before p_2 does, the value received by p_2 must be 1 as opposed to 0. Therefore, p_2 takes the branch and reads from the shared variable z. Figure 3.14 shows the partial order on events for this execution. Program order is captured by the vertical arrows in the diagram; channel communication is captured by the solid diagonal arrows. As per discussion in Section 3.3.2.3, we make the distinction between a channel operation and its completion. A channel operations is depicted as two half-circles; the operation's completion is captured by the bottom half-circle. That way, a send (top of the half-circle) happens-before its corresponding receive completes (bottom half).

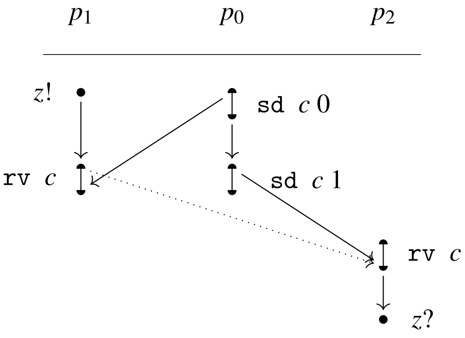


Figure 3.14: Partial order on conditional-race example.

Now, given that the send operations are in happens-before, meaning $(\texttt{sd}\ c\ 0)_{p_0} \rightarrow_{\mathsf{hb}} (\texttt{sd}\ c\ 1)_{p_0}$, and that channels are First-In-First-Out (FIFO), the reception of value 0 from c must occur before the reception of 1. This requirement is captured by the dotted arrow in the diagram. However, according to the semantics of channel communication (*i.e.,* rules (3.1) and (3.2) of page 57), this order does not impose a happens-before relation between the receiving events. In other words, there exist events that are necessarily ordered, but not in happens-before relation to one another.

The failure of our tentative definition of happens-before as necessarily-occurring-before, given early in this section, has subtle implications as discussed next.

3.6.1 Happens-before, traces, and commutativity of operations

Traces come from observing the execution of a program and are expressed as strings of events. In a concurrent system, however, events may not be causally related, which means that the order of some events is not pre-imposed. In reality, instead of sequences, events in a concurrent system form a partially ordered set (see Figure 3.14 for an example). As advocated by Mazurkiewicz [68], it is useful to combine sequential observations with a dependency relation for studying "the nonsequential behaviour of systems via their sequential observations." By defining an *independence relation* on events, it is possible to derive a notion of equivalence on traces: two traces are equivalent if it is possible to transform one into the other "by repeatedly commuting adjacent pairs of independent operations" [53].

One way to define independence is as follows: Given a run $R_i \xrightarrow{a} \cdot \xrightarrow{b} R$, we say that a and b are independent if $R_i \xrightarrow{b} \cdot \xrightarrow{a} R$, meaning,

- b is enabled at R_i,

- a is enabled at $R_i \xrightarrow{b} \cdot$, and
- there exists an R' such that $R_i \xrightarrow{b} R' \xrightarrow{a} R$.

Clearly, if a happens-before b, then a and b cannot be swapped in a trace. So, independence between two events means (at least) the absence of happens-before relation between them. But happens-before is not all that needs to be considered in the definition of independence.

When translating a partial order of events to a trace, not every linearization that respects the happens-before relation is a valid trace. Some linearizations of the partial order may not be "realizable" by the operational semantics. In other words, there can be traces that abide by the happens-before relation but that cannot be generated from the execution of a program. For example, we can obtain the following linearization given the partial order of Figure 3.14:

$$(\texttt{sd}\ c\ 0)_{p_0}\ \ (\texttt{sd}\ c\ 1)_{p_0}\ \ (\texttt{rv}\ c)_{p_2}\ \ (z!)_{p_2}\ \ (z?)_{p_1}\ \ (\texttt{rv}\ c)_{p_1}. \tag{3.6}$$

This linearization respects the partial order based on the happens-before relation: program order is respected, so is the relation between sends and their corresponding receives. However, this linearization breaks the first-in-first-out assumption on channels. FIFO is broken because, in order for p_2 to read from z, it must be that it received the value of 1 from the channel. But p_2 is the first thread to receive from the channel and, since 0 was the first value into the channel, it must also have been the first value read from the channel. Therefore, the linearization in Trace 3.6 is not "realizable" by the operational semantics. While happens-before restricts the commutation of trace operations, there exist other operations that are ordered (though not ordered by happens-before) and that, consequently, must not commute.

The difficulty in conciliating the commutativity of trace events with the happens-before relation remains counterintuitive today, even though its origins are related to an observation made years ago in a seminal paper by Lamport [56]. In the paper, Lamport points out that "anomalies" can arise when there exist orderings that are external to the definition of happens-before—see the "Anomalous Behavior" section of [56]. In order to avoid these anomalies, one suggestion from the paper is to expand the notion of happens-before so that, if a and b are necessarily ordered, then a and b are also in happens-before.

Let us analyze the consequences of rolling FIFO notions into the definition of happens-before. Given the example of Figure 3.13, since the sends are ordered in a happens-before relation, and the channel is FIFO, one can argue that the receive events should also be ordered by happens-before. According to this argument, we ought to promote the dotted line in Figure 3.14 to a solid $\rightarrow_{\mathsf{hb}}$ arrow. This modification would make the example well-synchronized. In one hand, given that the write to z by p_1 and the read from z by p_2 are *always* separated by events (by the two receive events in specific), interpreting the two memory accesses as being synchronized seems rather fitting: the two memory accesses cannot happen simultaneously, nor can they exist side-by-side in a trace.

There are downsides to this approach. For one, the resulting semantics deviates from Go's, but, more importantly, such a change does impact synchronization in counterintuitive ways. Specifically, making the dotted arrow a happens-before arrow would imply that a receiver (in this case p_2) can learn about prior events that are not known by the corresponding sender. If the dotted arrow is promoted to a synchronization arrow, the write $(z!)_{p_1}$ is communicated to p_2 via p_0 without p_0 itself being "aware" of the write. In other words, the write identifier is transmitted via p_0 but is not present in p_0's happens-before set.

We follow Go and allow for some events to always occur in order without affecting synchronization. Consequently, such ordered events are not considered to be in *happens-before* order. A less clear consequence, however, is that races can no longer be defined as simultaneous (or side-by-side) accesses to a shared variable. This point is explored next.

3.6.2 Manifest data races

Section 3.2 mentioned the concept of manifest data race; below we give a concrete definition.

Definition 11 (Manifest data race)**.** *A well-formed configuration R contains a* manifest data race *if either hold:*

$$R \xrightarrow{(z!)_{p_1}} \text{ and } R \xrightarrow{(z!)_{p_2}} \qquad \textit{(manifest write-write race on z)}$$

$$R \xrightarrow{(z?)_{p_1}} \text{ and } R \xrightarrow{(z!)_{p_2}} \qquad \textit{(manifest read-write race on z)}$$

for some $p_1 \neq p_2$.

Manifest data races can also be defined on traces.

Definition 12 (Manifest data race)**.** *A well-formed trace h contains a* manifest data race *if either*

$$(z!)_{p_1}\ (z!)_{p_2} \qquad \textit{(manifest write-after-write)}$$

$$(z!)_{p_1}\ (z?)_{p_2} \qquad \textit{(manifest read-after-write)}$$

$$(z?)_{p_1}\ (z!)_{p_2} \qquad \textit{(manifest write-after-read)}$$

are a sub-sequence of h and where $p_1 \neq p_2$).

While manifest races are obvious, races in general may involve accesses that are arbitrarily "far apart" in a linear execution. By bring conflicting accesses side-by-side, we could show irrefutable evidence of a race that, otherwise, may be obscured in a trace. Let $h \sqsubseteq h'$ represent the fact that h' is derivable from h by the repeated commutation of adjacent pairs of independent operations. If $h \sqsubseteq h'$ and h' contains a manifest data race, then we say h contains a data-race. This definition of races seems unequivocal. From here, soundness and completeness of a race detector may be defined as such:

Theorem 12. *(Soundness) If $S_0 \xrightarrow{h}$ is a run flagged by a data-race detector, then $h \sqsubseteq h_{dr}$ with h_{dr} containing a manifest data-race.*

Theorem 13. *(Completeness) Let $S_0 \xrightarrow{h}$ be a run such that $h \sqsubseteq h_{dr}$ and h_{dr} contains a manifest race. Then $S_0 \xrightarrow{h}$ is flagged by the data-race detector.*

Theorems 12 and 13 are also clear and unequivocal. More importantly, they link two world views: the view of races as unsynchronized accesses with respect to the happens-before relation and a view of races in terms of commutativity of trace events *à la* Mazurkiewicz. The problem with the concept of manifest data race and Theorems 12 and 13, however, is that when the definition of independence is made to respect FIFO order as well as the happens-before relation, the notion of manifest data race is no longer attainable. In other words, given a definition of independence which respects FIFO and happens-before, there exist racy traces from which a manifest data race is not derivable.

The program of Figure 3.13 on page 82 gives rise to such an example. The access to z by p_2 only occurs if p_2 receives the second message sent on the channel. In other words, the existence of event $(z?)_{p_2}$ in a trace is predicated on the order of execution of channel operations: p_2 only reads from z if the other thread, p_1, receives from c before p_2 does.[20] This requirement places the receive operations between the memory operations. Therefore, a trace in which $(z!)_{p_1}$ and $(z?)_{p_2}$ are side-by-side is not attainable. Yet, as discussed previously, the accesses to z are not ordered by happens-before, and, therefore, are concurrent. Since the accesses are also conflicting, they constitute a data race.

It seems that Mazurkiewicz traces are "more compatible" with confluence checking than data-race checking. In data-race checking, there are non-confluent runs that do not exhibit data races; these runs are non-confluent because they have "races on channels." In our example, the two receives from p_1 and p_2 are in competition for access to the channel. These receive operations are concurrent and non-confluent. Finally, the example also hints at the perhaps more fundamental observation: that races have little to do with *simultaneous* accesses to a shared variable but instead

[20] In this example, we use the value of the message received on a channel to branch upon. But since a receive from a channel changes a thread's "visibility" of what is in memory, it is possible to craft a similar example in which all message values are `unit` but in which a thread's behavior changes due to a change in the ordering of the receives.

with *unsynchronized* accesses. While simultaneous accesses are clearly unsynchronized, not all unsynchronized accesses may be made simultaneous.[21]

3.7 Related work

Race detection via the analysis of source code is an undecidable problem. Regardless, race detectors via the static analysis of source code [72, 100, 13] exist and have found application in industry. More recently, Blackshear et al. [13] implement a static analysis tool called RACERD to help the parallelization of previously sequential Java source code. The tool over approximates the behavior of programs and can, thereby, reject programs that turn out to be data-race free. This over approximation was not a hindrance, as even conservative parallelization efforts can lead to gains over purely sequential code.

By and large, however, instead of flagging races in a program as a whole, race detectors have resorted the analysis of particular *runs* of a program. To that end, detectors instrument the program so that races are either flagged during execution, in what is called *on-line* or *on-the-fly* race detection, or on logs captured during execution and analyzed *postmortem*. Even still, dynamic race detection is NP-hard [74] and many techniques have been proposed for detection at scale. Broadly, these techniques involve static analysis used to reduce the number of runtime checks [36][86], and heuristics that trade false-positive [90, 81, 20] or false-negative rates [66] for better space/time utilization. For example, by allowing races to sometimes go undetected, *sampling* race detectors let go of completeness in favor of lower overheads. One common heuristic, called the *cold region hypothesis*, is to sample more frequently from less executed regions of the program. This rule-of-thumb hinges on the assumption that faults are more likely to already have been identified and fixed if they occur in the hot regions of a program [66]. Alternatively, by going after a proxy instead of an actual race, imprecise race detectors let go of soundness. The prominent examples here are Eraser's LockSet [90] and Locksmith [81], which enforce a lock-based synchronization discipline. A violation of the discipline is a *code smell* but not necessarily a race. The amalgamation of different approaches have also been investigated, leading to *hybrid race detectors*. For example, O'Callahan and Choi [76] combined LockSet-based detection with happens-before information reconstructed from vector clocks; Choi et al. [20] extended LockSet to incorporate static analyses.

Another avenue of inquiry has lead to *predictive race detection* [93, 47], which attempts to achieve higher detection capabilities by extrapolating beyond individual runs. Huang et al. [47] incorporate abstracted control flow information and formulate race detection as a constraint solving problem. With the goal of observing more

[21] There may not exist a configuration from which two transitions are possible; transitions that involve conflicting memory accesses. Yet, it is possible for two access separated "in time" to be unsynchronized.

races per run, Smaragdakis et al. [93] introduce a new relation, called *causally-precedes*, which is a generalization of the happens-before relation.

A number of papers address race detection in the context of channel communication [24, 25, 97]. Some of the papers, however, do not speak of shared memory but, instead, define races as conflicting channel accesses. In that setting, the lack of conflicting accesses to channels imply determinacy. A different angle is taken by Terauchi and Aiken [97], who, among different kinds of channels, define a buffered channel whose buffer is overwritten by every write (*i.e.,* send) but never modified by a read (*i.e.,* receive). This kind of channel, referred to as a *cell*, behaves, in essence, as shared memory. The goal of Terauchi and Aiken [97] is, still, determinacy. Having conflated the concept of shared memory as a channel, determinacy is then achieved by ensuring the absence of conflicting accesses to channels. Our goal, however, is different: we aim to detect data-races but do not want to go as far as ensuring determinacy. Therefore, our approach allows "races" on channel accesses. From a different perspective, however, the work of Terauchi and Aiken [97] can be seen as complementary to ours: We conjecture that their type system can serve as the basis for a static data-race detector.

Among the dynamic data-race detection tools from industry, Banerjee et al. [8] discuss different race detection algorithms including one used by the Intel Thread Checker. The authors describe *adjacent conflicts*, which is similar to our notion of side-by-side or manifest data race. The paper also classifies races similar to our WaR, RaW, and WaW classification.

Go has a race detector integrated to its tool chain [42]. The `-race` command-line flag instructs the Go compiler to instrument memory accesses and synchronization events. The race detector is built on top of Google's sanitizer project [43] and TSan in particular [91, 44]. TSan is part of the LLVM's runtime libraries [92, 61]; it works by instrumenting memory accesses and monitoring locks acquisition and release as well as thread forks and joins. Note, however, that channel communication is the vehicle for achieving synchronization in Go. Even though locks exist, they are part of a package, while channels are built into language. Yet, the race detector for Go sits at a layer underneath. In this chapter we study race detection with channel communication taking a central role. Also, different from TSan, we employ propose a technique based on what we call *happens-before sets* as opposed to vector clocks. The consequences of this decision is discussed in detail on Section 3.5.

It is also relevant to point out that, in the absence of the DRF-SC guarantee, one may resort to finding data races involving weak memory behavior. Since the full C/C++11 memory model can harbor such races, and with the goal of finding data races in production level code, Lidbury and Donaldson [60] extend the ThreadSanitizer (TSan) [91, 44] to support a class of non-sequentially consistent executions.

3.8 Conclusion

We presented a dynamic data-race detector for a language in the style of Go: featuring channel communication as sole synchronization primitive. The proposed detector records and analyzes information locally and is well-suited for online detection.

Our race detector is built upon the operational semantics discussed in Chapter 2, where we formalize a weak memory model inspired by the Go specification [40]. In that setting, we recorded memory read- and write-events that were in happens-before relation with respect to a thread's present operation. This information was stored in a set called E_{hb} or the happens-before set of a thread, and it was used to regulate a thread's visibility of memory events. The core of the work was a proof of the DRF-SC guarantee, meaning, we proved that the proposed relaxed memory model behaves sequentially consistently in the absence of data races. The proof hinges on the fact that, in the absence of races, all threads agree on the contents of memory; see the *consensus lemmas* in Section 2.6. The scaffolding used in the proof of the consensus lemma contains the ingredients used of the race detectors presented in this chapter. Based on our experience, we conjecture that one may automatically derive a race detector given a weak memory model and its corresponding proof of the DRF-SC guarantee.

In the DRF-SC the proof of Chapter 2, we show that if a program is racy, it behaves sequentially consistent up to the point in which the first data-race is encountered. In other words, this first point of divergence sets in motion all behavior that is not sequentially consistent and which arise from the weakness in the memory model. With this observation, we argue that a race detector can operate under the assumption of sequential consistency. This is a useful simplification, as sequential consistent memory is conceptually much simpler than relaxed memories. If the data-race detector flags the first evidence of a data-race, then program behavior is sequentially consistent up to that point.

Avenues for future work abound. In contrast to data-race detectors based on vector clocks, our approach using happens-before sets does not provide as terse of a representation for the collection of memory events performed by a thread in between synchronization points. In effect, out approach has a larger footprint, which ought to be mitigated. On the other hand, our thorough expunging of stale information can serve as inspiration to vector clock based approaches, which allow for the accumulation of stale information—see Section 3.6. Another extension would be to statically analyze a target program with the goal of removing dynamic checks or ameliorating the detector's memory consumption. Here, we may be able to borrow from the research on static analysis for dynamic race-detection in the context of lock-based synchronization disciplines. Finally, by connecting our semantics and the work of Maarand [63], it may be possible to address some of the issues raised in Section 3.6 concerning a trace-theory based interpretation of data races.

Finding and fixing a mismatch between the Go memory model and data-race detector 4

Go is an open-source programming language developed at Google. In previous works, we presented formalizations for a weak memory model and a data-race detector inspired by the Go specification. In this chapter, we describe how our theoretical research guided us in the process of finding and fixing a concrete bug in the language. Specifically, we discovered and fixed a discrepancy between the Go memory model and the Go data-race detector implementation—the discrepancy led to the under-reporting of data races in Go programs. Here, we share our experience applying formal methods on software that powers infrastructure used by millions of people.

4.1 Introduction

Go is an open-source programming language designed for concurrency. Developed at Google, the language has gained traction in the areas of cloud computing [38], where it is used to implement various client-server applications and container management systems, such as Docker [69] and Kubernetes [18].

One of the language's main features are light-weight threads, called *goroutines*, which are spawned during function invocation. Any function can be made to execute asynchronously by simply prepending the keyword go to the function's name during invocation. Go's approach to synchronization also stands out. *Do not communicate by sharing memory; instead, share memory by communicating* [41]—is a catchphrase among Go programmers. The language's feature-mix encourages a style of programming where (1) variables are implicitly owned by goroutines, and (2) variables are shared when this ownership is transferred through direct communication. So, in contrast to locks, which favor synchronization via mutual exclusion, Go has channels, which typically enforce a happens-before relation [56] between a message sender and its receiver.

The discipline prescribed by Go's *share by communicating* slogan is not, however, enforced at compile time (as static data race detection has the potential of introducing a large number of false alarms). It is, therefore, possible for programs to harbor data races. Since data races often lead to counterintuitive behavior, the Go programming language comes with a data-race detector built into its toolchain.

The Go memory model is relaxed and its specification describes the behavior of well-synchronized programs. In Chapter 2, we gave a small-step operational semantics of a memory model inspired by Go's. There, we proved the DRF-SC guarantee, which states that data-race free (DRF) program executions behave sequentially consistently (SC) under the proposed model. Given the importance of flagging data races, in Chapter 3 we explore the use of our semantics for the sake of data-race detection. Armed with these formalisms, we turned our attention to Go's implementation. With that, we discovered that the Go data-race detector was not strictly abiding by the rules of the Go memory model specification. This oversight lead to the under-reporting of data races in Go programs. We then proposed and implemented a fix in conjunction with the Go community. Here, we discuss how the theoretical modeling of the language helped us find and address this issue.

In Sections 4.2 and 4.3, we will visit the Go memory model and explore examples of synchronization via channel communication. Having covered this background, we discuss how the Go data-race detector is built into the language (Section 4.4). In Section 4.4.1, we show that the detector's implementation inadvertently mismatched rules governing channel communication. We address the issue in Section 4.5 and share lessons we learned in Section 4.6.

4.2 Synchronization via channel communication

Two concurrent memory accesses constitute a data race if they reference the same memory location and at least one of the accesses is a *write*. Data races can be eliminated through synchronization, that is, the enforcement of an order between conflicting memory accesses. In Go, synchronization is performed via channel communication. Go channels assure FIFO communication from a sender to a receiver sharing the channel's reference. Channels can be dynamically created and closed—their type and finite capacity are fixed upon creation.

When attempting to receive from an empty channel, a thread blocks until, if ever, a value is made available by a sender. A thread also blocks when attempting to send on a channel that is full. According to the Go memory model specification [40], the following two main rules govern synchronization. Given a channel c with capacity C:

I. A send on c happens-before the corresponding receive from c completes.

II. The k^{th} receive from c happens-before the $(k+C)^{th}$ send on c completes.

The first rule establishes a causal relationship between a sender and its communicating partner. In contrast, the second rule establishes a relationship between a sender and some past receiver, without there being any message transmission between the two goroutines. Note also that the second rule accounts for channel capacity: a current sender is able to place a new message because some past receiver, by taking an older message out, has made space in the channel's buffer.

Figure 4.1a is an example of synchronization via rule (I), and Figure 4.1b is an example via rule (II). Throughout the chapter, we will follow the syntax in Chapter 2, which closely matches the syntax of a relevant subset of Go. The term $c \leftarrow v$, with the arrow pointing into c, stand for the sending of value v over channel c. Let $\leftarrow c$, with the arrow pointing away from c, stand for the reception of a value from the channel. Assuming a channel of capacity one, Figure 4.1a is the classic message passing example, while Figure 4.1b enforces mutual exclusion.

$T0$	$T1$
$z := 42$	$\leftarrow c$
$c \leftarrow 0$	$\texttt{load}\, z$

(a) Message passing example.

$T0$	$T1$
$c \leftarrow 0$	$c \leftarrow 0$
$z := 42$	$z := 43$
$\leftarrow c$	$\leftarrow c$

(b) Mutual exclusion example.

Figure 4.1: Synchronization via channel communication (channels of capacity one).

In the message passing example, the goroutine $T0$ writes to a shared variable z and, by sending a message over a channel, the routine transfers its implicit ownership of z. Goroutine $T1$ blocks until a message is ready to be received. Once a message has been received, $T1$ proceeds to load from z. This program is properly synchronized, which means $T1$ necessarily loads the value of 42 as opposed to potentially observing an uninitialized variable value. Using the happens-before (HB) rules of the Go memory-model specification, we can show that the memory accesses are properly synchronized as follows:

$$\begin{array}{rcll} z := 42 & \sqsubset_{hb} & c \leftarrow 0 & \text{via program order} \qquad (4.1)\\ c \leftarrow 0 & \sqsubset_{hb} & \leftarrow c & \text{via channel rule (I)} \qquad (4.2)\\ \leftarrow c & \sqsubset_{hb} & \texttt{load}\, z & \text{via program order} \qquad (4.3)\\ z := 42 & \sqsubset_{hb} & \texttt{load}\, z & \text{via transitivity of HB, (4.1), (4.2), (4.3).} \end{array}$$

While Figure 4.1a and rule (I) account for direct communication, Figure 4.1b relies on rule (II) and the use of channels as locks. The example in Figure 4.1b involves two threads attempting to write to the same shared variable. Before writing, a thread sends a message onto a channel. Because the channel has capacity one, all subsequent attempts to send again will block until the prior message is received. Therefore, it is not possible for $T0$ and $T1$ to execute their critical sections at the same time. The send is thus analogous to acquiring a lock, and the receive to releasing the lock. Again, we can use the Go memory model to reason about this example.

Without loss of generality, assume $T0$ sends its message first, then

$$\begin{aligned}
z := 42 \quad &\sqsubset_{hb} \quad \leftarrow c \text{ by } T0 && \text{via program order} && (4.4)\\
\leftarrow c \text{ by } T0 \quad &\sqsubset_{hb} \quad c \leftarrow 0 \text{ by } T1 && \text{via channel rule (II)} && (4.5)\\
c \leftarrow 0 \text{ by } T1 \quad &\sqsubset_{hb} \quad z := 43 && \text{via program order} && (4.6)\\
z := 42 \quad &\sqsubset_{hb} \quad z := 43 && \text{transitivity, (4.4), (4.5), (4.6).} &&
\end{aligned}$$

While mutual exclusion is ensured, we cannot ascertain the final value of z. If $T0$ sends a message before $T1$, then z equals 43; otherwise, $z = 42$. Note also that, in this example, rule (I) is obviated by the *program order;* therefore, the rule has no synchronization effect here.[1]

The Go memory model is described succinctly in plain English [40]. The word "completes," present in both rules (I) and (II), can easily be overlooked. By overlooking the distinction between an operation and its completion, the Go data-race detector over-synchronizes and fails to report certain races. The bug, which we describe in detail in the next section, is related to the following question: Is it possible for the detector to account for the mutex paradigm (Figure 4.1b) and, at the same time, observe the distinction between a channel operation and its completion?

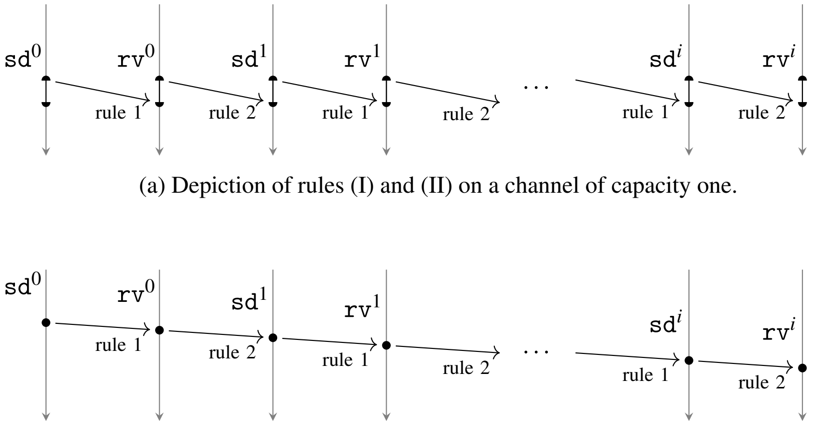
4.3 The Go memory model: Every word counts

The completion of a channel operation, in addition to the operation itself, is an important part of the Go memory model. In rule (I), it is not the case that a send happens-before the corresponding receive. Instead, the send happens-before the *completion* of the corresponding receive. Similar for rule (II), involving a past receive and the *completion* of a current send. To illustrate, consider Figure 4.2, where each vertical arrow represents the execution of a thread (flowing from top to bottom). Both the top and the bottom diagrams depicts consecutive send `sd` and receive `rv` operations on a channel of capacity one—the operations are indexed as to show their order of execution.

According to the Go memory model, channel operations are related as shown in Figure 4.2a. The operations are broken into two halves of a circle: the top is the operation and the bottom its completion. The arrows in the diagram represent the happens-before relation—arrows are labeled with the memory-model rule that justify their existence. According to rule (I), the 0^{th} send happens-before the completion of the 0^{th} receive—this relation is captured by the arrow starting at the top half-circle on the far left ($\mathtt{sd}^0$) and ending at the bottom half-circle to the right

[1] The relation below can be derived by both program-order as well as by rule (I). Similar for the send and receive operations performed by $T1$.

$$c \leftarrow 0 \text{ by } T0 \quad \sqsubset_{hb} \quad \leftarrow c \text{ by } T0$$



(a) Depiction of rules (I) and (II) on a channel of capacity one.

(b) Alternate formulation of rules (I) and (II) with no distinction between an operation and its completion.

Figure 4.2: The Go memory model specification, *every word counts*.

(completion of rv^0). The next arrow establishes the happens-before relation between receive rv^0 and send sd^1 according to rule (II), and so forth. Note from Figure 4.2a that an operation is related to its immediate predecessor. There is no chain of happens-before starting from the "distant" past. For example, although sd^0 is related to the completion of rv^0, and rv^0 is related to the completion of sd^1, it is not the case that sd^0 and sd^1 are related to each other.

Figure 4.2b captures an alternative formulation of the happens-before rules (I) and (II) where the word "completes" is left out. Sends and receives are not split into the operation and the operation's completion. Instead, sends and receives happen-before each other. This formulation leads to a chain starting at the very first send, and connecting every send and receive operation ever performed onto the channel. From an application programmer's perspective, this chain leads to an accumulation of happens-before information: after interacting with a channel, a goroutine's behavior is now dependent, not only on its communicating partner but, on every thread that has previously interacted with the channel. From the point of view of data races, this alternate formulation leads to over-synchronization.

The Go data-race detector's implementation matches the behavior of Figure 4.2b and, therefore, deviates from the Go memory model specification. Note that the over-synchronization on the part of the detector is not the result of careful deliberation, for example when false-negatives are accepted in exchange for lower runtime overheads. Rather, the implementation springs from an interpretation of synchronization from the perspective of locks rather than of channels. As will be discussed in Section 4.5, addressing this issue not only eliminates false-negatives but also yields lower runtime overhead.

4.4 The Go data-race detector

By adding `-race` to the command line, a Go program can be compiled and run with data-race detection enabled. The Go data-race detector is based on TSan, the Thread Sanitizer library [44]. The library is part of the LLVM infrastructure [59] and was originally designed to find races in C/C++11 applications.

When data-race detection is enabled, each thread becomes associated with an additional data structure. This data structure keeps track of the operations that are in happens-before from a thread's point of view. In most data-race detectors, including TSan, this data structure is a *vector clock* (VC) [56]. Vector clocks offer a compact representation of the relative order of execution between threads. With this bookkeeping, data-race detectors are able to find synchronization issues in programs—where synchronization means the transfer of happens-before information between threads.

In the setting of locks, a thread performs an *acquire* operation in order to "learn" the happens-before information stored in a lock. By performing a *release* operation, a thread deposits its happens-before information onto a lock. In the setting of channels, we can think of happens-before as being transferred via sends and receives.

Figure 4.3 contains snippets from Go's implementation of the send and receive operations. Unsurprisingly, Go implements a channel of capacity C as an array of length C. This array is contained in a struct called `hchan`. Struct member `sendx` is the index where a new message is to be deposited, while `recvx` is the index of the next message to be retrieved. Function `chanbuf` takes a channel struct and an index—the function returns a pointer to the channel's array at the given index. Note from lines 20 to 23 that a channel array is treated as a circular buffer.

When data-race detection is enabled, each channel array entry becomes associated with a vector clock. Also, when detection is enabled, a send operation (Listing 4.1) generates calls to acquire and release—lines 11 to 14. The acquire causes the sender to "learn" the happens-before (HB) information associated with the channel entry at `c.sendx`. The release causes the thread's HB information to be stored back into that entry.[2] The receive operation is similarly implemented and shown in Listing 4.2.

In light of the implementation described above, we now revisit the message passing and mutual exclusion examples of Section 4.2. In the case of message passing, a thread sends a message onto a channel of capacity one, then another thread receives this message before accessing a shared resource—see Figure 4.1a. According to the

[2] In the implementation of the send operation, a message is moved from the sender's buffer to a receiver's buffer `ep` on line 16. The index `c.sendx` is incremented in line 20 and the increment wraps around based on the length of the array—lines 21 to 23. The number of elements in the array is incremented, the lock protecting the channel is unlocked and the function returns—lines 24 to 26.

Listing 4.1: Send.

```
1   func chansend(c *hchan,
2                 ep unsafe.Pointer,
3                 block bool,
4                 callerpc uintptr)
5                 bool {
6     ...
7     lock(&c.lock)
8     ...
9     if c.qcount < c.dataqsiz {
10      qp := chanbuf(c, c.sendx)
11      if raceenabled {
12        raceacquire(qp)
13        racerelease(qp)
14      }
15
16      typedmemmove(c.elemtype,
17                   qp, ep)
18
19
20      c.sendx++
21      if c.sendx == c.dataqsiz {
22        c.sendx = 0
23      }
24      c.qcount++
25      unlock(&c.lock)
26      return true
27    }
28    ...
29  }
```

Listing 4.2: Receive.

```
1   func chanrecv(c *hchan,
2                 ep unsafe.Pointer,
3                 block bool)
4                 ( selected ,
5                 received bool) {
6     ...
7     lock(&c.lock)
8     ...
9     if c.qcount > 0 {
10      qp := chanbuf(c, c.recvx)
11      if raceenabled {
12        raceacquire(qp)
13        racerelease(qp)
14      }
15      if ep != nil {
16        typedmemmove(c.elemtype,
17                     ep, qp)
18      }
19      typedmemclr(c.elemtype, qp)
20      c.recvx++
21      if c.recvx == c.dataqsiz {
22        c.recvx = 0
23      }
24      c.qcount--
25      unlock(&c.lock)
26      return true, true
27    }
28    ...
29  }
```

Figure 4.3: Snippets of Go's send and receive operations from `runtime/chan.go`.

data-race detector's implementation, the channel array entry at index 0 observes an *acquire* followed by *release* on behalf of the sender. Then, again, a sequence of *acquire* followed by *release* on behalf of the receiver. In effect, the happens-before information of the sender is transferred to the receiver: specifically, the *release* by $T0$ followed by the *acquire* by $T1$ places $T0$'s write operation in happens-before relation with respect to $T1$'s read operation. The message passing example of Figure 4.1 is thus deemed properly synchronized by the Go data-race detector.

We can reason about the mutual exclusion example of Figure 4.1b in similar terms. A thread sends onto a channel, accesses a shared resource, and then receives from the channel. With the receive operation, this thread deposits its happens-before information onto the channel—line 13 of Listing 4.2. The second thread then acquires this happens-before information when it sends onto the channel—line 12 of Listing 4.1. Again, the Go data-race detector's implementation correctly deems the example as properly synchronized.

4.4.1 The bug

Although the Go data-race detector behaves correctly on the message-passing and mutual-exclusion examples, the detector's implementation does not reflect the Go memory model specification. The acquire/release sequence performed on behalf of send and receive operations follows the typical lock usage. Channel programming is, however, different from lock programming. The current implementation of the detector leads to an accumulation of happens-before information associated with channel entries. This monotonic growth of happens-before information, however, is not prescribed by the Go memory model.

In the example that follows, we illustrate the mismatch between (1) the implementation of the data-race detector and (2) the memory model specification. We show how this mismatch leads to over-synchronization and the under reporting of data races.

$T0$	$T1$	$T2$
$c \leftarrow 0$	$c \leftarrow 0$	$\leftarrow c$
$z := 42$		$\texttt{load}\, z$
$\leftarrow c$		

Figure 4.4: Example that highlights a mismatch between the Go memory model and the Go data-race detector implementation. (Capacity of channel c equals one).

Let c in Figure 4.4 be a channel of capacity one. The example is then a mix of mutual exclusion and message passing: $T0$ is using the channel as a lock in an attempt to protect its access to a shared variable,[3] and we can interpret $T1$ as using the same channel to communicate with $T2$.[4] Now, consider the interleaving in which $T0$ runs to completion, followed by $T1$, then $T2$—shown in Trace 4.7. Is the write to z by $T0$ in a race with the read of z by $T2$?

$$(c \leftarrow 0)_{T0} \quad (z := 42)_{T0} \quad (\leftarrow c)_{T0} \quad (c \leftarrow 0)_{T1} \quad (\leftarrow c)_{T2} \quad (\texttt{load}\, z)_{T2} \tag{4.7}$$

The original Go data-race detector does not flag these accesses as racy:[5] $T0$ releases its happens-before (HB) by sending on the channel. This HB is stored in the vector clock associated with c's 0^{th} array entry. The send by $T1$ performs an acquire followed by a release, at which point the VC associated with the entry contains the union of $T0$'s and $T1$'s happens-before. Finally, the receive by $T2$ performs an acquire and a release, causing $T2$ to learn the happens-before of $T0$ and $T1$. Formally,

[3] The send operation by $T0$ is analogous to acquire and the receive to release.

[4] Recall that the mutual exclusion and message passing patterns were introduced in Figure 4.1 and discussed in Section 4.2.

[5] GitHub issue https://github.com/golang/go/issues/37355

the data-race detector derives a happens-before relation between the write and the read as follows:

$$\begin{array}{rcll} z := 42 & \sqsubset_{hb} & \leftarrow c \text{ by } T0 & \text{via program order} \\ \leftarrow c \text{ by } T0 & \sqsubset_{hb} & c \leftarrow 0 \text{ by } T1 & \text{release by } T0\text{, acquire by } T1 \\ c \leftarrow 0 \text{ by } T1 & \sqsubset_{hb} & \leftarrow c \text{ by } T2 & \text{release by } T1\text{, acquire by } T0 \\ \leftarrow c \text{ by } T2 & \sqsubset_{hb} & \texttt{load}\, z & \text{via program order} \\ z := 42 & \sqsubset_{hb} & \texttt{load}\, z & \text{via transitivity of HB} \end{array}$$

According to the Go memory model specification, however, the receive from c in $T0$ is not in happens-before relation to the send in $T1$. Instead, the receive is in happens-before relation to the *completion* of the send! Information about the write to z by thread $T0$ is transmitted to $T1$, but this information is only incorporated into $T1$ *after* the thread has transmitted its message to $T2$. Therefore, $T2$ does not receive $T0$'s happens-before information. In other words, according to the Go memory model, there is no chain of happens-before connecting $T0$ to $T2$. The trace captured by equation (4.7) is thus racy, with the race depicted in Figure 4.5. Specifically, the race is captured by the absence of a path between the write to z in $T0$ and the load of z in $T2$.

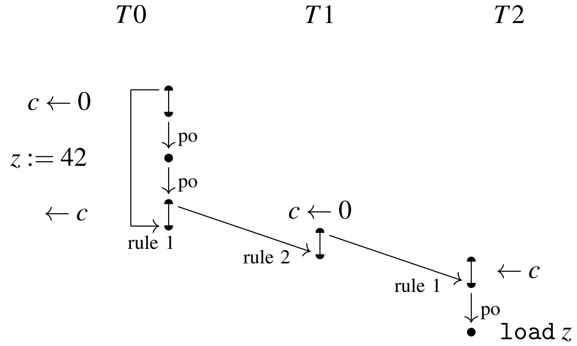


Figure 4.5: Partial order on events according to the Go memory model. The HB relation is represented by arrows labeled with the Go memory model rule justifying the arrow's existence. The top part of the half-circle corresponds to a channel operation and the bottom to its completion.

The Go memory model calls for a mix between channel communication as described by Lamport [56] and lock programming. Lamport [56] was studying distributed systems in the absence of shared memory: the shared resources were the channels themselves, and the absence of races (of channel races) was related to determinism. In contrast, Go employs channels as a primitive for synchronizing *memory accesses*. In Go, some happens-before relations are forged solely between communicating partners—these relations are derived from rule (I), which is also present in [56]. Similar to lock programming, some happens-before relations are the result of an accumulation of effects from past interactions on a channel. This accumulation occurs when we incorporate rule (II), which is not present in [56]. So, while the language favors a discipline where an implicit notion of variable ownership is transferred via direct communication, as prescribed by rule (I), by incorporating rule (II), Go also supports the use of channels as locks.

4.5 The fix: capturing the semantics of channels

The repair to the Go data-race detector's deviation from the memory model specification comes from acknowledging that a primitive, different from acquire and release, can better fit the semantics of synchronization via channel communication. We propose the primitive depicted in Figure 4.6, which we call *release-acquire-exchange* or `rea`. Let T_t be the happens-before information of thread t, m be the channel entry where a message will be deposited or retrieved, and C_m be the happens-before information associated with m. The primitive is implemented with a thread releasing onto a place-holder and then acquiring from C_m. The happens-before in C_m is then overwritten with the HB information from the place-holder.[6] We added this new synchronization primitive into TSan, the data-race detection library that powers the Go data-race detector. With the new primitive in place, changes to the Go sources became trivial:[7] it involved changing sequences of acquire/release calls with a call to *release-acquire-exchange*.

$$\frac{\mathrm{T}'_t := \mathrm{T}_t \sqcup \mathrm{C}_m \qquad \mathrm{C}'_m := \mathrm{T}_t}{(\mathrm{T}_t, \mathrm{C}_m) \Rightarrow^{\mathtt{rea}(t,m)} (\mathrm{T}'_t, \mathrm{C}'_m)}$$

Figure 4.6: Semantics of "release-acquire-exchange," a new primitive added to TSan.

Given the addition of `rea` into TSan, let us revisit trace (4.7). While the original implementation of the Go data-race detector did not flag this trace as racy, the updated version does. Given the detector's updated implementation, we can reason about the race as follows. Let T_{T0}, T_{T1}, and T_{T2} be data-structures storing

[6] The place-holder is a variable local to a function in TSan, as opposed to an extra memory region allocated in Go.

[7] Changes in Go `https://golang.org/cl/220419` and TSan `https://reviews.llvm.org/D76322`

happens-before information of threads $T0$, $T1$, and $T2$. Let $\mathrm{C}_{c[0]}$ be the happens-before associated with the 0^{th} array entry of channel c. We denote the write event to z as $!z$ and, for simplicity, we represent happens-before information as a set of memory events. The race-detector state is then the tuple $[\mathrm{T}_{T0}, \mathrm{T}_{T1}, \mathrm{T}_{T2}, \mathrm{C}_{c[0]}]$, with initial state $[\{\}, \{\}, \{\}, \{\}]$. The data-race detector performs the following transitions as the program executes:

$$\begin{array}{lllll r}
\mathrm{T}_{T0} & \mathrm{T}_{T1} & \mathrm{T}_{T2} & \mathrm{C}_{c[0]} & & \\
[\,\{\}, & \{\}, & \{\}, & \{\}\,] & \Rightarrow^{(c\leftarrow 0)_{T0}} & \\
[\,\{\}, & \{\}, & \{\}, & \{\}\,] & \Rightarrow^{(z:=42)_{T0}} & (4.8) \\
[\{!z\}, & \{\}, & \{\}, & \{\}\,] & \Rightarrow^{(\leftarrow c)_{T0}} & (4.9) \\
[\{!z\}, & \{\}, & \{\}, & \{!z\}] & \Rightarrow^{(c\leftarrow 0)_{T1}} & (4.10) \\
[\{!z\}, & \{!z\}, & \{\}, & \{\}\,] & \Rightarrow^{(\leftarrow c)_{T2}} & (4.11) \\
[\{!z\}, & \{!z\}, & \{\}, & \{\}\,] & \Rightarrow^{(\texttt{load}z)_{T2}} & (4.12)
\end{array}$$

The write to z by $T0$ places $!z$ into T_{T0}—transition from equation (4.8) to (4.9). Sends and receives are interpreted according to their formal semantics in Figure 4.6. The receive by $T0$ places the write event into the channel-entry's happens-before—equations (4.9) and (4.10). The send by $T1$ places the write event into the thread's happens-before and overwrites the channel-entry's happens-before with the empty set—equations (4.10) and (4.11). The receive by $T2$ retrieves the empty happens-before information—equations (4.11) and (4.12). Therefore, at the time $T2$ loads from the shared variable, the write to z by $T0$ is not in happens-before with respect to $T2$. In conclusion, the execution is racy.

Note that the fix to the Go data-race detector does not invalidate the use of channels as locks. Without loss of generality, let the trace below be an execution of the mutual exclusion example of Figure 4.1b.

$$(c\leftarrow 0)_{T0} \quad (z:=42)_{T0} \quad (\leftarrow c)_{T0} \quad (c\leftarrow 0)_{T1} \quad (z:=43)_{T1} \quad (\leftarrow c)_{T1} \tag{4.13}$$

The detector's execution, from initial state $[\mathrm{T}_{T0}, \mathrm{T}_{T1}, \mathrm{C}_{c[0]}] = [\{\}, \{\}, \{\}]$, is

$$\begin{array}{llll}
\mathrm{T}_{T0} & \mathrm{T}_{T1} & \mathrm{C}_{c[0]} & \\
[\,\{\}, & \{\}, & \{\}\,] & \Rightarrow^{(c\leftarrow 0)_{T0}} \\
[\,\{\}, & \{\}, & \{\}\,] & \Rightarrow^{(z:=42)_{T0}} \\
[\{!z\}, & \{\}, & \{\}\,] & \Rightarrow^{(\leftarrow c)_{T0}} \\
[\{!z\}, & \{\}, & \{!z\}] & \Rightarrow^{(c\leftarrow 0)_{T1}} \\
[\{!z\}, & \{!z\}, & \{\}\,] &
\end{array}$$

Note that the event $!z$ capturing the write by $T0$ is contained in T_{T1} before $T1$ attempts to write to z. In other words, the writes are ordered by happens-before and the execution is properly synchronized. Thus, the answer to the question raised at the end of Section 4.2, *"is it possible to support the use of channels as locks (as in the mutex example) and still avoid over-synchronization?"* is yes.

We implement the new synchronization primitive `rea` in TSan with one pass, as opposed to two passes, over the data-structure storing happens-before information. Therefore, the updated data-race detector implementation provides better performance than the original sequence of *acquire* followed by *release*. Another consequence of our fix is a potential reduction in the memory footprint associated with data-race detection. This savings comes from the fact that vector clocks associated with channel entries no longer observe as large of an accumulation of happens-before information—this point was previously touched upon in Section 4.3, Figure 4.2. We provide a short experimental evaluation next.

Memory footprint Here we illustrate how our fix to the Go data-race detector leads to a smaller memory footprint. Consider an in-place parallel sorting algorithm where an array is recursively split, up to some depth, in approximately half. Each region of the array is assigned to a thread for sorting. When a thread completes sorting, it signals its parent. The parent merges, in-place, the consecutive array regions previously assigned to its children.

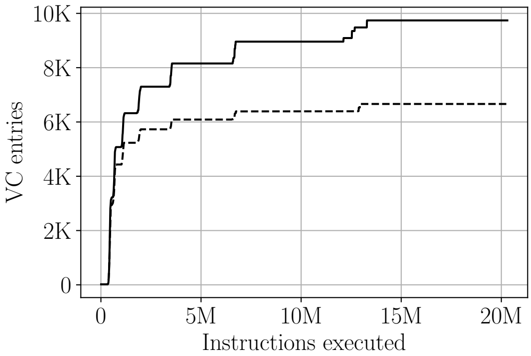


Figure 4.7: Number of VC entries associated with channels during the execution of an in-place parallel sorting algorithm: before (solid line) and after (dashed line) the introduction of release-acquire-exchange.

We tracked the number of entries in the vector clocks associated with channel array entries. Measurements of the number of VC entries were taken multiple times during the program's execution. For ease of collecting and plotting the data, we modified TSan to call out to a reference data-race detector implemented in

Python.[8,9] Figure 4.7 shows the number of VC entries before and after the fix to the data-race detector—meaning, with a race detector that performed an *acquire* followed by *release* versus a race detector that implements the *release-acquire-exchange* primitive. The x-axis is the number of instructions executed, the y-axis is the number of vector-clock entries consumed so far in the execution. As the program makes progress, more entries accumulate in the vector clocks associated with channel entries. This accumulation is much more accentuated before the fix to the data-race detector. In fact, for this workload, the fix lead to larger than 30% reduction in the number of VC entries after 12.5M instructions were executed.

4.5.1 From small-step operational semantics to `rea`

The *release-acquire-exchange* primitive comes from our previous formalizations of Go channels. It is conceptually useful to distinguish between happens-before information transmitted on behalf of rule (I) versus (II). In Chapter 2, our formalization split a channel c in two: a forward and a backward one. The forward channel c_f holds messages and thread-local information to be transmitted, as prescribed by rule (I), from a sender to its corresponding receiver. The backward channel c_b, which flows from a prior receiver to a current sender, captures rule (II) of the memory model.[10]

In Chapter 2, threads or goroutines $p\langle\sigma,t\rangle$ have a unique identifier p, contain thread-local information σ, and a term t corresponding to the program under execution. When it comes to data-race detection, thread-local information σ is composed of *happens-before* data. This data could be stored in a vector clock or, more simply, it could be a set of read- and write-events that are in happens-before with respect to the thread. Synchronization, therefore, entails the exchange of thread-local information σ via channel communication.

Configurations consist of the parallel composition of goroutines, memory events, and channels. The semantics of Chapter 2 is operational. We give the reduction rules for sends and receives in Figure 4.8—other rules are omitted and can be found in the original chapter. The `let`-construct in R-REC is a binder for the local variable r in a term t. In the case of R-REC, the `let`-construct allows t to refer to the value obtained when receiving from a channel.

According to reduction rule R-SEND, when a thread sends a value v, the thread's local state σ is placed on the forward channel alongside v. The rule captures the placement of the message (v,σ) onto the forward channel as follows: if q_2 is the

[8] `https://github.com/dfava/paper.go.mm.drd`

[9] Because of differences in how vector clocks are allocated and managed, the memory gains reported by the reference data-race detector may be different from TSan's.

[10] As noted in Chapter 3, "the interplay between forward and backward channels can also be understood as a form of flow control. Entries in the backward channel's queue are not values deposited by threads. Instead, [these entries] can be seen as tickets that grant senders a free slot in the communication channel."

$$\frac{\neg closed(c_f[q_2]) \qquad \sigma' = \sigma + \sigma''}{c_b[q_1 :: \sigma''] \parallel p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q_2] \;\rightarrow\; c_b[q_1] \parallel p\langle \sigma', t\rangle \parallel c_f[(v,\sigma) :: q_2]}\ \textsc{R-Send}$$

$$\frac{v \neq \bot \qquad \sigma' = \sigma + \sigma''}{\begin{array}{lll} c_b[q_1] \parallel & p\langle \sigma, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t\rangle & \parallel c_f[q_2 :: (v, \sigma'')] \;\rightarrow \\ c_b[\sigma :: q_1] \parallel & p\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle & \parallel c_f[q_2] \end{array}}\ \textsc{R-Rec}$$

Figure 4.8: Send and receive reduction rules in the calculus of Chapter 2.

content of the forward channel before transmission, $(v,\sigma) :: q_2$ are the contents after. The transmission of σ models rule (I) of the Go memory model: a receiver who receives (v,σ) will learn about the sender's actions up to the given send operation.

Besides transmitting, a sender also learns HB information in accordance to rule (II). Precisely, the $(k+C)^{th}$ sender obtains, from the backward channel, thread-local state from the k^{th} receiver. This is captured by the update $\sigma' = \sigma + \sigma''$ with the state σ'' coming from the backward channel. Thus, if the contents of the backward channel were $q_1 :: \sigma''$ before the send, the channel is left with q_1 after the send. Note that the update to the sender state occurs *on completion* of the send operation: the update "occurs after" the sender has deposited its message onto the forward channel—concretely, the send transmits the thread state σ as opposed to the updated thread state σ'.

When receiving, a goroutine obtains a value v as well as a state σ'' from a sender. As dictated by rule (I), the receiver updates its state given the corresponding sender state: $\sigma' = \sigma + \sigma''$. The sender also deposits its state onto the backward channel. Similar to R-SEND, the original thread state σ is deposited as opposed to the updated thread state σ'.

For both reduction rules R-SEND and R-REC, local thread state σ is deposited onto a channel as opposed to the update thread state σ'. This discipline creates a distinction between an operation and its completion. In effect, the reduction rules do not cause the over-synchronization observed by the Go data-race detector.

4.5.2 Why not *acquire* and *release*?

The formalization of Chapter 2 speaks of synchronization in terms of channel communication. Since TSan operates at the level of locks, we might be tempted to implement the reduction rules with acquire and release operations. The reduction rule R-SEND could be implemented with a thread releasing its happens-before information into the forward queue, and *then* acquiring happens-before information from the backward queue. Similarly, R-REC can be implemented with a release to the backward queue, followed by an acquire from the forward queue.

One invariant of the semantics of Chapter 2 is that the number of elements in the forward and backward queues equals the capacity of the channel. Since a thread must first release its HB into the channel before acquiring from the channel, there would be more than C entries in the queues while a send or receive is in-flight. In fact, when using acquire and release operations as primitives, the Go data-race detector would need to allocate an array of length $C+2$ for a channel of capacity C. Given such an array, sends and receives can be implemented with acquire/release operations as shown in Listings 4.3 and 4.4. Recall that `c.sendx` and `c.recvx` are the indices into the array where the next message is to be deposited and retrieved respectively. Recall also that `chanbuf` returns a pointer to a channel's array at a given index.

Listing 4.3: Implementation of send.

```
idx := c.sendx+1
if idx == C+2
  idx = 0
qf := chanbuf(c, c.sendx)
qb := chanbuf(c, idx)
racerelease (qf)
raceacquire (qb)
```

Listing 4.4: Implementation of receive.

```
idx := c.recvx−1
if c.recvx == 0
  idx = C+1
qf := chanbuf(c, c.recvx)
qb := chanbuf(c, idx)
racerelease (qb)
raceacquire (qf)
```

Although correct, there are major downsides to the solution of Listings 4.3 and 4.4. First, it requires additional memory allocation. Second, because the Go runtime expects a channel of capacity C to be implemented with an array of length C, the solution would require intrusive changes. Third, from a timing perspective, in order to implement a single channel operation, the solution performs two passes over the data-structure storing happens-before information—we want a solution that performs less passes.

Compared to *acquire* and *release*, the *release-acquire-exchange* primitive requires no additional allocation in Go, involves minimal changes to the Go runtime, and has lower overhead.

4.6 Lessons learned

When we started looking at TSan's source code, our goal was to improve Go's data-race detector by expressing synchronization in terms of channels as opposed to locks [35]. We began reading the source code of Go and TSan in November 2019. In January 2020, we started experimenting by compiling the projects from source and making modifications in order to gain experience and intuition. This tinkering lead us to find, in early and mid February, a small Go compiler bug[11] and a

[11] `https://github.com/golang/go/issues/37012`

small performance bug in TSan.[12] Shortly after, around late February, we found the bug described in this chapter.

Given our experience formalizing the calculus in Chapter 2, we could see similarities between our reduction rules and the Go implementation.[13] The implementations of send and receive, however, stood out. The bug was thus found by inspection. We created a test (Figure 4.4) to showcase what we believed was discrepancy between the detector and the memory model. From there, we filed an issue on GitHub and started interacting with the Go community. With this interaction, which went until May, an initial patch was iteratively improved until being accepted for release.[14]

In this section, we collect insights drawn from our experience in both (1) formalizing aspects of the Go programming language and (2) interacting with the TSan and Go communities.

Models do not have to be right, they have to be useful

In Chapter 2, we developed a memory model based on the Go specification. Before embarking on studying the Go source code, we found ourselves at cross-roads. Since our model is not as relaxed as Go's, more theoretical research remained to be done. We pondered whether to continue working on formalizations or whether to investigate how the current model fits the "real world." Both avenues are interesting to us. By taking, for now, the second avenue, we learned that *models do not have to be right, they have to be useful.* Our memory model formalization in Chapter 2 *is not* the memory model of Go, but it is close enough to allow us to reason about Go and its implementation.

Mind the gap

In one hand, we have the concept of a data-race according to the synchronization rules of the Go memory model specification. The specification is expressed in English. On the other hand, we have the Go data-race detector implementation, with thousands of lines of code spawning different projects and repositories and involving at least three languages (Go, C/C++, and assembly). These are two ends of a spectrum. Our model was useful, in part, because of where it sits in this spectrum. When developing the model in Chapter 2, we followed the English text of the Go memory model specification very closely. Our model, however, is expressed in structural operational semantics—its rules form an executable implementation. Our calculus,

[12] `https://reviews.llvm.org/D74831`

[13] For example, the closing of channels in both Chapter 2 and in Go cause happens-before information to be deposited onto the channel, regardless of whether the channel is full.

[14] See the release notes for Go 1.16 in `https://golang.org/doc/go1.16`

therefore, forms a bridge between source code and the specification expressed in natural language.[15]

Bad news is good news

The effort in formalizing and proving a nontrivial property of a software system is often high. Before finding the issue described in this chapter, we had been working on formalisms related to Go for over two years. This high barrier of entrance is both good and bad. It is good, less obviously so, because it opens opportunities for collaboration between industry and academia. While industry excels at delivering software, academia can provide artifacts, such as formalisms and proofs, which are still not as commonly produced in industry.[16]

4.7 Conclusion

The bug described in this chapter evaded skilled developers for about six years, nearly since the data-race detector was bolted onto the Go runtime. In this chapter, we share how formal methods played an integral role in bringing the issue to light, and giving it closure.

[15] Our observation about the representational different between specification and implementation is not new. The idea of bridging specification and implementation has been tackled by many fronts, for example [7].

[16] Because of stigma, the "formal" qualifier has been de-emphasized when disseminating formal methods in industry [75]. This stance has shifted dramatically [23].

5 Incorporating load buffering into the memory model

By allowing multiple writes to the same memory location to co-exist, the semantics introduced in Chapter 2 leads to a form of relaxation called *delayed writes*. Although fairly relaxed, the resulting memory model precludes load buffering. In this chapter, we explore mechanisms for supporting load buffering—which we refer to as *delayed reads*. We conjecture that a delayed read/write semantics, meaning, the combination of delayed reads with the memory model of Chapter 2, can precisely capture the Go memory model specification.

5.1 Introduction

In Chapter 2, we introduced a delayed-write memory model inspired by the Go programming language—where synchronization takes place via channel communication. Delayed writes are handled by allowing multiple stores to the same shared variable to coexist. Think of how there can be multiple copies of a "variable" in hardware: in a stale cache entry, as entries in a processor's write-buffers, and in memory. When it comes to our semantics, these copies exist as terms in a run-time configuration—see the runtime configuration in equation (2.6) on page 27 for the delayed-write run-time configuration.

In this chapter, we extend the memory model of Chapter 2 by introducing *delayed reads*, which account for load buffering. With delayed reads, a handle is obtained before a concrete value is finally observed. This relaxation introduces new behavior to the execution of a program: at the moment a read observes a concrete value, additional writes may be present and available to service the read operation—writes that would not have already been executed earlier when the read was issued. To illustrate, consider the following trace where threads p_1 and p_2 write values v_1 and v_2 to z, followed by p_0 issuing a read of z, p_3 executing a write of value v_3 to z, and p_0's read being serviced with a concrete value.

$$(z!v_1)_{p_1} \quad (z!v_2)_{p_2} \quad (z?\_)_{p_0} \quad (z!v_3)_{p_3} \quad (z?v)_{p_0}$$

Note that we make a distinction between *the read being issued*, $(z?\_)_{p_0}$, and *the read being serviced*, $(z?v)_{p_0}$. The underscore in $(z?\_)_{p_0}$ is representing a place holder that is to be filled at some point in the future—the place holder is filled with a concrete value when the read is serviced. In the delayed write semantics of Chapter 2, reads are not delayed; meaning that there exists no distinction between when a read operation is issued and when it is “concretized” or “serviced.” Therefore, according to the delayed write semantics, v_1 and v_2 are the only values available to service the read if p_0 executes before p_3. However, by delaying the concretization of the read until after the execution of p_3, the delayed read/write semantics introduces new possibilities. For example, the read can observe the value of v_3 in addition to the values of v_1 and v_2.

We start by introducing delayed reads to a language with only memory events. In Section 5.2, we show how delayed reads can introduce circular dependencies. These dependencies lead to out-of-thin-air (OOTA) behavior in axiomatic semantics. In the case of our executable semantics, circularity can lead to non-terminating reductions—even for programs that terminate under sequential consistency. In Section 5.3 we discuss conditionals and the possibility of branching on a symbolic (as opposed to concrete) value. The interaction of branching and delayed reads can lead to OOTA behavior in both axiomatic as well as in the semantics explored in Section 5.3. We are of the believe that an executable semantics, like hardware, can support delayed read/write events without introducing OOTA and non-termination. To that end, more research is needed and, in this chapter, we point out open questions along the way. We end the chapter by discussing channel operations and the sending of a symbolic value over a channel in Section 5.4 before concluding in Section 5.5.

5.2 Delaying reads in a setting without conditionals

Here we introduce load buffering (aka. delayed reads) to a very simple language. To help us gain intuition, we start with motivating examples. The examples differ in the extent that reads depend on prior writes. In the simplest case, there is no dependence.

Example 3 (Load buffering, no data dependence)**.** *The code of Listing 5.1 shows the simplest illustration of load buffering. There are two threads, each performs a load followed by a store. Note that there is no data dependence between the load and the store within a thread.*

$$\begin{array}{ll} T1 & T2 \\ r_1 := x & r_2 := y \\ y\ := 23 & x\ := 42 \end{array}$$

Listing. 5.1: Load buffering, no data dependence

Note also that the stores occur at the end of each thread, which means that write delays do not come into play. Indeed, both a sequentially consistent memory model and the model of Chapter 2 produce the same results. In particular, they *forbid the post-condition* $(r_1, r_2) = (42, 23)$. *This post-condition would imply that at least one load completed after the write within a thread: Observing, for instance* $r_1 = 42$ *implies that x must have been written before—see the read-from edge in the candidate execution graph of Figure 5.1, noting that* rf *stands for the* read-from *relation. Due to program order, this prior write would imply that* $r_2 = y$ *must have been read before. Since the writing to y comes after the read of x, then y must still be* 0*, which precludes the result* $(r_1, r_2) = (42, 23)$.

We would like, however, for a weak memory model to be able to produce $(42, 23)$. *The absence of a data dependency allows each thread to execute its instructions out-of-order. In terms of candidate executions, this example*[1] *is captured by Figure 5.1.*

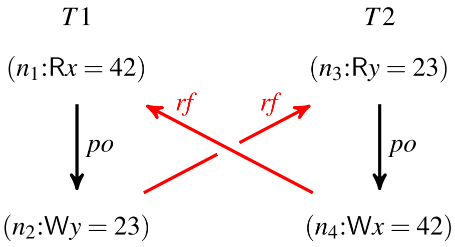


Figure 5.1: Candidate execution for load buffering with no data dependence

Example 4 (Load-buffering, non-circular data dependence)**.** *Listing 5.2 shows a slight generalization*[2] *of the LB example from Listing 5.1. Like in the previous example, the program of Listing 5.2 behaves the same under a sequentially consistent memory model as well as the delayed-write semantics of Chapter 2 . These memory models preclude* $r_2 = 42$ *and, instead, can produce the following results for* (r_1, r_2) *at termination:* $\{(42, 0), (0, 23), (0, 0)\}$.

$$\begin{array}{ll} T1 & T2 \\ r_1 := x & r_2 := y \\ y\ := r_1 & x\ := 42 \end{array}$$

Listing. 5.2: Load buffering, no circular dependence

[1] Figure 5.1 resembles Figure 5.2 except that, since there is no data dependence here, the ppo-edge is not present.

[2] The example corresponds to the LB example of Maranget et al. [65, Section 7]. The example is also mentioned in Valle [99] to illustrate how a "delayed-writes" model (such as the one in Chapter 2) is not as relaxed as we would like it to be.

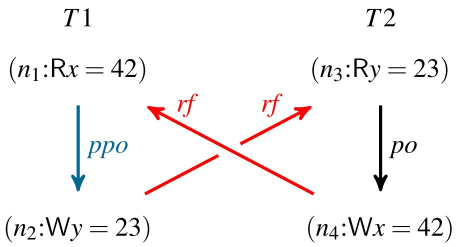

$T1$
$T2$
$(n_1{:}\mathsf{R}x = 42)$
$(n_3{:}\mathsf{R}y = 23)$
rf
rf
ppo
po
$(n_2{:}\mathsf{W}y = 23)$
$(n_4{:}\mathsf{W}x = 42)$

Figure 5.2: Candidate execution for load buffering with no circular dependence

Note that there is a data dependence between the read and the subsequent write in $T1$ and that there is no such dependence in $T2$. From a local perspective, $T2$'s instructions can be swapped because the swap does not alter the thread's behavior. The swap does, however, alter the behavior of the program, which is the composition of $T1$ and $T2$. In particular, the swap of $T2$'s instructions introduce $(42, 42)$ as a possible program outcome. The difference between $T2$'s dependency and $T2$'s lack of data dependency is captured by the $\rightarrow_{\mathsf{ppo}}$ (preserved program order) versus the $\rightarrow_{\mathsf{po}}$ (program order) arrows in Figure 5.2.

We can also justify $r_2 = 42$ by arguing that since there is no happens-before relation across the threads, each read can observe the write on the opposite thread. In particular, r_1 can read 42, then store it in y, and that can be read by the second thread into r_2.

Next, we explore the idea of delaying or buffering the load of y in $T2$.

Load-buffering and read events. The result $(42, \mathbf{42})$ can be obtained from Listing 5.2 by delaying the effect of $r_2 := y$ in the second thread, in other words, to execute that statement asynchronously. From the point of view of our semantics and its notation, such delay is achieved by adding *read events*, which are tuples of the form

$$n\llbracket ?y \rrbracket \, .$$

A reduction[3] showcasing how read events work is sketched in equation (5.1). For clarity, the reduction also uses labeled transitions with $\xrightarrow{n(z!v)_{p_2}}$ representing a write and $\xrightarrow{n(z?5)_{p_2}}$ a read. We do not always fill-in all components of a transition; for

[3] The avid reader may notice that in this chapter, we are interpreting $r := e;\ t$ as $\texttt{let}\ r := e\ \texttt{in}\ t$, which is a slight departure from the notation of Chapter 2.

instance, if we do not yet know the value of a variable that is being read, then the value is left out of the corresponding label.

$$\begin{aligned}
&\langle \sigma_1, r_1 := x; y := r_1; r_1\rangle \parallel \langle \sigma_2, \underline{r_2 := y}; x := 42; r_2\rangle && \xrightarrow{p_2(y?\_)_{n_2}}\\
&\langle \sigma_1, r_1 := x; y := r_1; r_1\rangle \parallel \langle \sigma_2, \underline{x := 42}; n_2\rangle \parallel n_2\llbracket ?y\rrbracket && \xrightarrow{p_2(x!42)_{m_1}}\\
&\langle \sigma_1, \underline{r_1 := x}; y := r_1; r_1\rangle \parallel \langle \sigma_2', n_2\rangle \parallel n_2\llbracket ?y\rrbracket \parallel m_1\llparenthesis x{:=}42\rrparenthesis && \xrightarrow{p_1(x?\_)_{n_1}}\\
&\langle \sigma_1, y := n_1; n_1\rangle \parallel \langle \sigma_2', n_2\rangle \parallel \underline{n_1\llbracket ?x\rrbracket} \parallel n_2\llbracket ?y\rrbracket \parallel \underline{m_1\llparenthesis x{:=}42\rrparenthesis} && \xrightarrow{deref}\\
&\langle \sigma_1, y := \underline{n_1}; \underline{n_1}\rangle \parallel \langle \sigma_2', n_2\rangle \parallel \underline{n_1\llbracket ?42\rrbracket} \parallel n_2\llbracket ?y\rrbracket \parallel m_1\llparenthesis x{:=}42\rrparenthesis && \rightarrow\\
&\langle \sigma_1, \underline{y := 42}; 42\rangle \parallel \langle \sigma_2', n_2\rangle \parallel n_1\llbracket ?42\rrbracket \parallel n_2\llbracket ?y\rrbracket \parallel m_1\llparenthesis x{:=}42\rrparenthesis && \xrightarrow{m_2(y!42)_{p_1}}\\
&\langle \sigma_1, 42\rangle \parallel \langle \sigma_2', n_2\rangle \parallel n_1\llbracket ?42\rrbracket \parallel \underline{n_2\llbracket ?y\rrbracket} \parallel m_1\llparenthesis x{:=}42\rrparenthesis \parallel \underline{m_2\llparenthesis y{:=}42\rrparenthesis} && \xrightarrow{deref}\\
&\langle \sigma_1, 42\rangle \parallel \langle \sigma_2', \underline{n_2}\rangle \parallel n_1\llbracket ?42\rrbracket \parallel \underline{n_2\llbracket ?42\rrbracket} \parallel m_1\llparenthesis x{:=}42\rrparenthesis \parallel m_2\llparenthesis y{:=}42\rrparenthesis && \rightarrow\\
&\langle \sigma_1, 42\rangle \parallel \langle \sigma_2', 42\rangle \parallel n_1\llbracket ?42\rrbracket \parallel n_2\llbracket ?42\rrbracket \parallel m_1\llparenthesis x{:=}42\rrparenthesis \parallel m_2\llparenthesis y{:=}42\rrparenthesis\ .
\end{aligned} \tag{5.1}$$

In the second configuration in equation (5.1), a read has been issued, leading to an event $n_2\llbracket ?y\rrbracket$. The local variable r_2 is now represented by the new name n_2. In fact, every occurrence of r_2 is replaced by n_2 in that thread.

The next step issues a write event. If the second thread would then perform its last step and dereference n_2, the thread would not find any non-shadowed write[4] to y other than y's initial value. The core of the example, however, is the fact that now the first thread is allowed to execute before this dereferencing takes place. The first thread starts by reading x. This read is also delayed, thus generating n_1. However, since the delay associated with n_1 is not interesting, we let the dereferencing take place right away: the read to x, as captured by n_1, is serviced by the write event that wrote 42 to x. In the next step, the write of 42 into y is issued, leading to the event $m_2\llparenthesis y{:=}42\rrparenthesis$.

At this point, the "earlier" read marked by n_2 is dereferenced. Note that we can now execute a step that was not possible in the delayed-write semantics of Chapter 2. The introduction of delayed reads created the possibility of servicing the read of y with the write $m_1\llparenthesis y{:=}42\rrparenthesis$. We are then left with $(r_1, r_2) = (42, 42)$.

The machinery involved in the above reduction introduces a certain amount of relaxation in the direction of load-buffering. The reduction above is still overly eager in the sense that there are no *symbolic writes*, meaning, there are no attempts to write a handle n. The eagerness lies in the treatment of the first thread. When p_1 executes the write $y := n_1$, the derivation dereferences n_1 to 42 opposed to writing the handle n_1. Nonetheless, we are still able to obtain the desired relaxed behavior despite the absence of symbolic writes.

[4] The concept of shadow percolates the thesis starting on Chapter 2.

Next, we craft a more complex example in order to show the need of symbolic writes.

Example 5 (Longer delay)**.** *Building on Listing 5.2, we create a situation where symbolic writes are called for. In a relaxed memory environment, it is desirable for the code in Listing 5.3 to produce an execution in which y contains* 42*. In terms of swapping, this result comes from performing the assignment of* $z := 42$ *before the read of x and the write to y. The candidate execution justifying* $y = 42$ *is given in Figure 5.3.*

$T1$	$T2$
$r_1 := x$	$r_2 := z$
$y := r_1$	$x := r_2$
$z := 42$	

Listing. 5.3: Longer delay

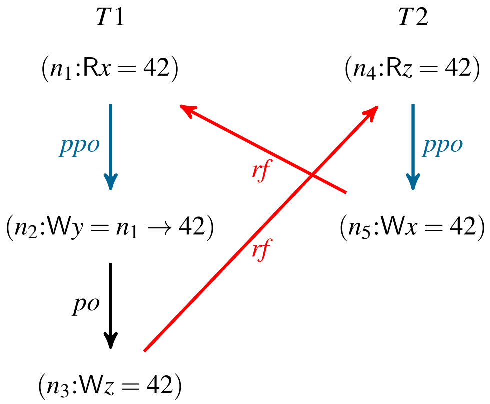

$T1$
$T2$
$(n_1{:}\mathsf{R}x = 42)$
$(n_4{:}\mathsf{R}z = 42)$
ppo
rf
ppo
$(n_2{:}\mathsf{W}y = n_1 \to 42)$
rf
$(n_5{:}\mathsf{W}x = 42)$
po
$(n_3{:}\mathsf{W}z = 42)$

Figure 5.3: Candidate execution for “longer delay” example

As shown in the reduction below, in order for y to contain 42, we need the write to y in p_1 to store a handle as opposed to an actual value.

$$\begin{array}{ll}
\langle \sigma_1, \underline{r_1 := x}; y := r_1; z := 42; r_1 \rangle \parallel \langle \sigma_2, r_2 := z; x := r_2; r_2 \rangle & \xrightarrow{p_1(x?\_)_{n_1}} \\
\langle \sigma_1, \underline{y := n_1}; z := 42; n_1 \rangle \parallel \langle \sigma_2, r_2 := z; x := r_2; r_2 \rangle \parallel n_1 \llbracket ?x \rrbracket & \xrightarrow{p_1(y!n_1)_{n_2}} \\
\langle \sigma_1', \underline{z := 42}; n_1 \rangle \parallel \langle \sigma_2, r_2 := z; x := r_2; r_2 \rangle \parallel n_1 \llbracket ?x \rrbracket \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis & \xrightarrow{p_1(z!42)_{n_3}} \\
\langle \sigma_1'', n_1 \rangle \parallel \langle \sigma_2, \underline{r_2 := z}; x := r_2; r_2 \rangle \parallel n_1 \llbracket ?x \rrbracket \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis & \xrightarrow{p_2(z?\_)_{n_4}} \\
\langle \sigma_1'', n_1 \rangle \parallel \langle \sigma_2, x := n_4; n_4 \rangle \parallel n_1 \llbracket ?x \rrbracket \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel \underline{n_3 \llparenthesis z{:=}42 \rrparenthesis} \parallel \underline{n_4 \llbracket ?z \rrbracket} & \xrightarrow{deref} \\
\langle \sigma_1'', n_1 \rangle \parallel \langle \sigma_2, x := \underline{n_4}; \underline{n_4} \rangle \parallel n_1 \llbracket ?x \rrbracket \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel \underline{n_4 \llbracket ?42 \rrbracket} & \rightarrow \\
\langle \sigma_1'', n_1 \rangle \parallel \langle \sigma_2, \underline{x := 42}; 42 \rangle \parallel n_1 \llbracket ?x \rrbracket \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel n_4 \llbracket ?42 \rrbracket & \xrightarrow{p_2(x!42)_{n_5}} \\
\langle \sigma_1'', n_1 \rangle \parallel \langle \sigma_2', 42 \rangle \parallel \underline{n_1 \llbracket ?x \rrbracket} \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel n_4 \llbracket ?42 \rrbracket \parallel \underline{n_5 \llparenthesis x{:=}42 \rrparenthesis} & \xrightarrow{deref} \\
\langle \sigma_1'', \underline{n_1} \rangle \parallel \langle \sigma_2', 42 \rangle \parallel \underline{n_1 \llbracket ?42 \rrbracket} \parallel n_2 \llparenthesis y{:=}n_1 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel n_4 \llbracket ?42 \rrbracket \parallel n_5 \llparenthesis x{:=}42 \rrparenthesis & \rightarrow \\
\langle \sigma_1'', 42 \rangle \parallel \langle \sigma_2', 42 \rangle \parallel \underline{n_1 \llbracket ?42 \rrbracket} \parallel \underline{n_2 \llparenthesis y{:=}n_1 \rrparenthesis} \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel n_4 \llbracket ?42 \rrbracket \parallel n_5 \llparenthesis x{:=}42 \rrparenthesis & \rightarrow \\
\langle \sigma_1'', 42 \rangle \parallel \langle \sigma_2', 42 \rangle \parallel n_1 \llbracket ?42 \rrbracket \parallel n_2 \llparenthesis y{:=}42 \rrparenthesis \parallel n_3 \llparenthesis z{:=}42 \rrparenthesis \parallel n_5 \llparenthesis x{:=}42 \rrparenthesis &
\end{array} \tag{5.2}$$

Execution starts with a read of x into r_1 and a write of r_1 into y. Unlike in the previous examples, the process is not yet in possession of a concrete value, so it stores a handle n_1 instead.[5] Once x has been written the value of 42, the delayed read of x can be concretized (*i.e.,* the last *deref* in the equation). At this point, the symbolic write n_2, which references n_1, can also be concretized. This leads to the result of $y = 42$ at the end of execution.

5.2.1 Regulating observability

Without proper bookkeeping, when reads and writes are delayed, the final configuration becomes a soup of unordered memory events. In this extreme scenario, reads can access any write event, as none of the writes shadow each other. Ignoring even program order, this scenario would allow for *prophetic reads*, meaning, the reading of values from "the future," such as the load of x observing the value of 42 from the subsequent write in $r = x; \cdots; x := 42$. In this extremely relaxed semantics, reads can only observe values that were written at some point—this makes it seem like the semantics precludes *out-of-thin-air* results.

Besides being a thought exercise, such an extremely relaxed (and quite unrealistic) semantics can be a departing point for design. What we need to do is to impose program order and thus preclude:

1. prophetic reads, which are reads that observe values from the future, and

[5] Note that steps 4 and 5 of the second process are similar: they involve a read followed by a write where the write depends on the value being read. However, in this case, we concretize the value before the write because a delay would not be interesting here.

2. outdated reads, which are reads that observe shadowed values.

These two forms of reads correspond to the two *negative observability conditions* in the Go memory model. Precisely, the model states that " a read r of a variable v is allowed to observe a write w to v if both of the following hold" [40]

$$r \text{ does not happen before } w. \tag{5.3}$$

$$\text{There is no other write } w' \text{ to } v \text{ that happens after } w \text{ but before } r. \tag{5.4}$$

In the delayed writes semantics of Chapter 2, the situation described by equation (5.3) could not happen. The delayed writes semantics prevents a load from observing the result of a future write[6] simply because the identity of the write is not available when the read takes place. In that semantics, the we only had to protect reads from observing outdated writes. This protection was accomplished by the introduction of *shadow sets*. The semantics also kept track of *happens-before sets*, but these took a lesser role: The happens-before sets existed as a way to keep shadow sets updated.

With the introduction of delayed reads, we must prevent prophetic reads in addition to shadowed reads. In the case of prophetic reads, the key to regulating observability is to switch perspective: instead of thinking about what the read can observe, concentrate on how writes lend their values to observers. A write event must track the reads that are in its past, and it must preclude the written value from being observed by these past reads.

In contrast with the delayed write semantics of Chapter 2, the introduction of delayed reads elevates the concept of happens-before set to the same level as shadow-sets: both sets now regulate observability. We then have the following symmetry:

> A read cannot observe a shadowed write, and
> a write cannot publish to a happened-before read.

As we will see next, the reduction rules for the semantics follow naturally from this discussion.

5.2.2 Reduction rules

The reduction rules for delayed reads and writes are given in Figure 5.4. Memory events related to reads are in green and writes in red. Just like in the reduction rules from previous chapters, the ν-binder, known from the π-calculus, indicates dynamic scoping [71].

Instead of performed eagerly, reads and writes generate events which record the state σ of the thread at the time the event was generated, and the "intention" of the event. In the case of a read, the intent of observing a value associated with a variable.

[6] For example a future write within the same thread.

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (n, !z), E_s + E_{hb}(!z)) \qquad \mathit{fresh}(n)}{p\langle \sigma, z := v; t\rangle \;\rightarrow\; \nu n\; (p\langle \sigma', t\rangle \parallel n\llparenthesis \sigma, z := v \rrparenthesis)}\;\textsc{R-Write}$$

$$\frac{\sigma = (E_{hb}, \_) \qquad \sigma' = (E_{hb} + (n, ?z), \_) \qquad \mathit{fresh}(n)}{p\langle \sigma, \texttt{let}\; r = \texttt{load}\; z\; \texttt{in}\; t\rangle \;\rightarrow\; \nu n\; (p\langle \sigma', \texttt{let}\; r = \; n\; \texttt{in}\; t\rangle \parallel n\llbracket \sigma, \texttt{load}\; z \rrbracket)}\;\textsc{R-Read}$$

$$\frac{m \notin E^r_s \qquad (n, \_) \notin E^w_{hb}}{\begin{array}{lcl} n\llbracket (E^r_{hb}, E^r_s), \texttt{load}\; z \rrbracket & \parallel & m\llparenthesis (E^w_{hb}, E^w_s), z := v \rrparenthesis \;\rightarrow \\ n\llbracket (E^r_{hb}, E^r_s), z, v \rrbracket & \parallel & m\llparenthesis (E^w_{hb}, E^w_s), z := v \rrparenthesis \end{array}}\;\textsc{R-Deref}$$

$$\frac{}{P \parallel n\llparenthesis \_, \_ := v \rrparenthesis \;\rightarrow\; P[v/n] \parallel n\llparenthesis \_, \_ := v \rrparenthesis}\;\textsc{R-Subst}_W$$

$$\frac{}{P \parallel n\llbracket \_, \_, v \rrbracket \;\rightarrow\; P[v/n] \parallel n\llbracket \_, \_, v \rrbracket}\;\textsc{R-Subst}_R$$

Figure 5.4: Operational semantics with delayed reads (*i.e.,* load buffering) as well as delayed writes: memory access rules.

In the case of a write, the intent of associating a new value with a variable. The observability rule R-DEREF is responsible for fulfilling this intent by associating a read to a write. R-DEREF captures the two observability restrictions detailed in the Go memory model: the one preventing shadowed reads, as captured by equation (5.4), and the other preventing prophetic reads, equation (5.3).

Notice that, when doing a read, we now record the read event on the happens-before set of the reading thread. Compared to the reduction rules from Chapter 2, we now distinguish between read (?) an write (!) events in happens-before sets. The syntax $E_{hb}(!z)$ means the labels of all writes to z in E_{hb}—note from rule R-WRITE that only prior writes get shadowed by a new write. Note also that read and write labels are values, so we can have a chain of indirections: the R-DEREF rule can dereference a load even when the corresponding write-event stores a symbolic value.

Rules R-SUBST$_W$ and R-SUBST$_R$ allow for the substitution of a value v that is associated with a handle n. These rules have no premise. Substitution is "safe" because, by the time it takes place, prior reductions[7] must have already verified the necessary constraints related to shadowing and happens-before.

[7] By *prior reductions* we mean prior R-READ, R-WRITE, or R-DEREF reductions.

Next, we illustrate the reductions with an example. The example shows how the semantics prevents out-of-thin-air behavior at the cost of non-termination.

Example 6 (No out-of-thin-air)**.** *Here we show how the semantics precludes a prototypical out-of-thin-air result associated with load buffering. Given the code below, it is important to forbid the possibility of producing $r_1 = r_2 = 42$. In fact, any value for r_1 and r_2, other than 0, is considered out-of-thin-air.*

$$
\begin{array}{ll}
\quad T1 & \qquad T2 \\
r_1 := x & \qquad r_2 := y \\
y\ := r_1 & \qquad x\ := r_2
\end{array}
$$

Note that we translate the example to the syntax of Chapter 2, which uses, for example, the let-construct. Execution starts in equation (5.5). The write events n_0 and n_1 represent the initial value of variables x and y; recall that $\sigma_\bot$ is the empty state $(E_{hb}, E_s) = (\emptyset, \emptyset)$. Initial thread states σ_1 and σ_2 are both $(\{(n_0, !x), (n_1, !y)\}, \emptyset)$ meaning that the initialization of the shared variables x and y happens-before the threads' initialization, and that no shadowing has occurred since no variables have been written to.

Without loss of generality, we start with the first thread, which loads from the shared variable and executes its let-construct, and similarly for the second thread. The question is, what values can r_1 and r_2 assume at the end of execution, where the values are associated with the read events n_2 and n_3 respectively.

$$
\begin{aligned}
&\langle \sigma_1, \texttt{let}\ r_1 = \underline{\texttt{load}\ x}\ \texttt{in}\ y := r_1 \rangle \parallel \langle \sigma_2, \texttt{let}\ r_2 = \texttt{load}\ y\ \texttt{in}\ x := r_2 \rangle \parallel \\
&\quad n_0 \llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1 \llparenthesis \sigma_\bot, y := 0 \rrparenthesis && \xrightarrow{p_1(x?\_)_{n_2}} \\
&\langle \sigma_1', \underline{\texttt{let}\ r_1 = n_2\ \texttt{in}\ y := r_1} \rangle \parallel \langle \sigma_2, \texttt{let}\ r_2 = \texttt{load}\ y\ \texttt{in}\ x := r_2 \rangle \parallel \\
&\quad n_0 \llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1 \llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2 \llbracket \sigma_1, ?x \rrbracket && \xrightarrow{let} \\
&\langle \sigma_1', y := n_2 \rangle \parallel \langle \sigma_2, \texttt{let}\ r_2 = \underline{\texttt{load}\ y}\ \texttt{in}\ x := r_2 \rangle \parallel \\
&\quad n_0 \llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1 \llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2 \llbracket \sigma_1, ?x \rrbracket && \xrightarrow{p_2(y?\_)_{n_3}} \\
&\langle \sigma_1', y := n_2 \rangle \parallel \langle \sigma_2', \underline{\texttt{let}\ r_2 = n_3\ \texttt{in}\ x := r_2} \rangle \parallel \\
&\quad n_0 \llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1 \llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2 \llbracket \sigma_1, ?x \rrbracket \parallel n_3 \llbracket \sigma_2, ?y \rrbracket && \xrightarrow{let} \\
&\langle \sigma_1', y := n_2 \rangle \parallel \langle \sigma_2', x := n_3 \rangle \parallel \\
&\quad n_0 \llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1 \llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2 \llbracket \sigma_1, ?x \rrbracket \parallel n_3 \llbracket \sigma_2, ?y \rrbracket
\end{aligned}
\tag{5.5}
$$

At this point, we can reduce the assignments. These reductions can also be done in *any order. Let us start with the second thread.*

$$\begin{aligned}
&\langle \sigma_1', y := n_2\rangle \parallel \langle \sigma_2', \underline{x := n_3}\rangle \parallel \\
&\quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_1, ?x\rrbracket \parallel n_3\llbracket \sigma_2, ?y\rrbracket \qquad \xrightarrow{(x!n_3)_{n_4}} \\
&\langle \sigma_1', \underline{y := n_2}\rangle \parallel \langle \sigma_2'', \rangle \parallel \\
&\quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_1, ?x\rrbracket \parallel n_3\llbracket \sigma_2, ?y\rrbracket \\
&\quad n_4\llparenthesis \sigma_2', x := n_3\rrparenthesis \qquad \xrightarrow{(y!n_2)_{n_5}} \\
&\langle \sigma_1'', \rangle \parallel \langle \sigma_2'', \rangle \\
&\quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_1, ?x\rrbracket \parallel n_3\llbracket \sigma_2, ?y\rrbracket \parallel \\
&\quad n_4\llparenthesis \sigma_2', x := n_3\rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2\rrparenthesis
\end{aligned} \tag{5.6}$$

Threads have now been reduced to the empty term. R-DEREF is the only rule that can be applied. Given that the thread states are:

$$\begin{aligned}
\sigma_1' &= (\{(n_0, !x), (n_1, !y), (n_2, ?x), \}, \emptyset) \\
\sigma_2' &= (\{(n_0, !x), (n_1, !y), (n_3, ?y), \}, \emptyset) \\
\sigma_1'' &= (\{(n_0, !x), (n_1, !y), (n_2, ?x), (n_5, !y), \}, \{(n_1, !y), \}) \\
\sigma_2'' &= (\{(n_0, !x), (n_1, !y), (n_3, ?y), (n_4, !x), \}, \{(n_0, !x), \})
\end{aligned}$$

then both writes to x (the one in n_0 and the one in n_4) can service the read of x associated with n_2. Similarly, both the writes to y, which are n_1 and n_5, can service the read of y associated with n_3. Let us look at all these possibilities:

Observing the initial values of x and y. The case in which the reads of x and y observe the initial value of the shared variables is captured by n_2 and n_3 being serviced by n_0 and n_1 respectively. This case does not involve load buffering, and there is no possibility for circular reasoning. Although not shown here, the configuration reduces to a term where all reads and writes have been "concretized," meaning, there are no terms of the form:

$$n\llbracket \sigma, ?n'\rrbracket \quad \text{or} \quad n\llparenthesis \sigma, \_ := n'\rrparenthesis$$

Load buffering on one thread. The scenario covers the case in which load buffering plays a role in one of the threads' load but not the other. In this scenario, there are two symmetrical sub-cases:

1. n_2 is serviced by n_4 while n_3 is serviced by the initial value of y, or
2. n_3 is serviced by n_5 while n_2 is serviced by the initial value of x.

In either of the sub-cases, we have $(r_1, r_2) = (0,0)$ at the end of execution, and the configuration reduces to concrete terms. Take for example sub-case 1, starting with the end configuration from equation (5.6)—note that we drop the threads for convenience:

$$\begin{aligned}
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel \underline{n_2\llbracket\sigma_1, ?x\rrbracket} \parallel n_3\llbracket\sigma_2, ?y\rrbracket \parallel \\
& \quad \underline{n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis} \parallel n_5\llparenthesis\sigma_1', y := n_2\rrparenthesis \qquad \xrightarrow{deref} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel \underline{n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis} \parallel n_2\llbracket\sigma_1, x, n_4\rrbracket \parallel \underline{n_3\llbracket\sigma_2, ?y\rrbracket} \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis \parallel n_5\llparenthesis\sigma_1', y := n_2\rrparenthesis \qquad \xrightarrow{deref} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel \underline{n_2\llbracket\sigma_1, x, n_4\rrbracket} \parallel n_3\llbracket\sigma_1, y, 0\rrbracket \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis \parallel n_5\llparenthesis\sigma_1', y := \underline{n_2}\rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, x, n_4\rrbracket \parallel \underline{n_3\llbracket\sigma_1, y, 0\rrbracket} \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := \underline{n_3}\rrparenthesis \parallel n_5\llparenthesis\sigma_1', y := n_4\rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, x, \underline{n_4}\rrbracket \parallel n_3\llbracket\sigma_1, y, 0\rrbracket \parallel \\
& \quad \underline{n_4\llparenthesis\sigma_2', x := 0\rrparenthesis} \parallel n_5\llparenthesis\sigma_1', y := n_4\rrparenthesis \qquad \xrightarrow{subst_w} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, x, 0\rrbracket \parallel n_3\llbracket\sigma_1, y, 0\rrbracket \parallel \\
& \quad \underline{n_4\llparenthesis\sigma_2', x := 0\rrparenthesis} \parallel n_5\llparenthesis\sigma_1', y := \underline{n_4}\rrparenthesis \qquad \xrightarrow{subst_w} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, x, 0\rrbracket \parallel n_3\llbracket\sigma_1, y, 0\rrbracket \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := 0\rrparenthesis \parallel n_5\llparenthesis\sigma_1', y := 0\rrparenthesis
\end{aligned}$$

Load buffering on both threads. Here we have n_2 and n_3 being serviced by n_4 and n_5 respectively. In other words, load buffering plays are role in both threads. In this case, the configuration reduces to the following:

$$\begin{aligned}
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel \underline{n_2\llbracket\sigma_1, ?x\rrbracket} \parallel n_3\llbracket\sigma_2, ?y\rrbracket \parallel \\
& \quad \underline{n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis} \parallel n_5\llparenthesis\sigma_1', y := n_2\rrparenthesis \qquad \xrightarrow{deref} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, ?n_3\rrbracket \parallel \underline{n_3\llbracket\sigma_2, ?y\rrbracket} \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis \parallel \underline{n_5\llparenthesis\sigma_1', y := n_2\rrparenthesis} \qquad \xrightarrow{deref} \\
& n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_1, ?n_3\rrbracket \parallel n_3\llbracket\sigma_2, ?n_2\rrbracket \parallel \\
& \quad n_4\llparenthesis\sigma_2', x := n_3\rrparenthesis \parallel n_5\llparenthesis\sigma_1', y := n_2\rrparenthesis
\end{aligned} \tag{5.7}$$

Note the circular dependency between n_2 and n_3.

At this point, we have different options. For example, by applying SUBST$_R$ on n_2 and n_3, the configuration can be reduced to contain a term $n_2\llbracket\sigma_1, ?n_2\rrbracket$, where the value of n_2 depends on itself! Although it does not produce an actual out-of-thin-air behavior, unfortunately the execution terminates in a state where symbolic values cannot be concretized—this situation can be interpreted as a deadlock.

$$\begin{aligned}
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2\llbracket \sigma_1, ?\underline{n_3} \rrbracket \parallel \underline{n_3\llbracket \sigma_2, ?n_2 \rrbracket} \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := n_3 \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2\llbracket \sigma_1, ?n_2 \rrbracket \parallel \underline{n_3\llbracket \sigma_2, ?n_2 \rrbracket} \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := \underline{n_3} \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2\llbracket \sigma_1, ?n_2 \rrbracket \parallel n_3\llbracket \sigma_2, ?n_2 \rrbracket \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := n_2 \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis
\end{aligned}$$

Unfortunately, not all reductions of this program are terminating! For example, going back to the end-state of equation (5.7), the following is a non-terminating reduction:

$$\begin{aligned}
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2\llbracket \sigma_1, ?n_3 \rrbracket \parallel \underline{n_3\llbracket \sigma_2, ?n_2 \rrbracket} \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := \underline{n_3} \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel \underline{n_2\llbracket \sigma_1, ?n_3 \rrbracket} \parallel n_3\llbracket \sigma_2, ?n_2 \rrbracket \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := \underline{n_2} \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis \qquad \xrightarrow{subst_r} \\
& n_0\llparenthesis \sigma_\bot, x := 0 \rrparenthesis \parallel n_1\llparenthesis \sigma_\bot, y := 0 \rrparenthesis \parallel n_2\llbracket \sigma_1, ?n_3 \rrbracket \parallel \underline{n_3\llbracket \sigma_2, ?n_2 \rrbracket} \parallel \\
& \quad n_4\llparenthesis \sigma_2', x := \underline{n_3} \rrparenthesis \parallel n_5\llparenthesis \sigma_1', y := n_2 \rrparenthesis \qquad \xrightarrow{subst_r} \ldots
\end{aligned}$$

The example above shows that programs that terminate under sequential consistency can have non-terminating runs in the delayed read/write semantics! These non-terminating runs are related to circular dependencies. The same dependencies caused axiomatic semantics to produce out-of-thin-air behavior. Can we steer reductions towards end-configurations that contain only concrete values? Hardware does not live- or dead-lock and does not produce out-of-thin-air results under cyclic dependencies. Our intuition is that an operational semantics that mimics hardware should also be able to avoid these problems. Certainly that is a simplistic view, as the issue lays beyond the relaxed nature of hardware and also rests in the interplay with compiler optimizations. Although we have gained insights, our understanding is still incomplete. For example, our treatment does not clearly differentiate between compile-time versus-runtime transformations. We wonder if it is possible to define a minimal set of assumptions or restrictions that guarantee the well-behavior of a memory model.

Compared to the delayed write semantics, we show how the delayed/read write semantics supports certain types of optimizations at the expense of introducing non-termination. One important observation that came late to us is the following: If we were to frame the semantics in terms of instruction swaps, as opposed to read-delays, these optimizations could have been supported without also introducing non-termination. In Chapter 3, we discuss the notion of commutativity of trace events, and touch on the notion of equivalence of traces. Here, we are not talking about commuting trace events but commuting instructions in the program text. Instructions a and b that commute in the program text may not commute in a trace. For example, even if a and b are adjacent in the program text, the events associated with the execution of a and b may not be adjacent in a trace. The events associated with a and b may not be made to commute in a trace because (1) the execution of other threads may split apart the events generated by a and b, and (2) we may not be able to bring the events associated with a and b back together (*i.e.,* we may not be able to put the events adjacent to each other) while still maintaining trace equivalence. Had we, however, defined commutation as transformation on the program text, we could have taken a program P and derived an equivalent program P' such that the behavior of P' under a delayed write semantics, $[\![P']\!]_{DW}$, matches the behavior of P under a delayed read/write semantics, $[\![P]\!]_{DRW}$.

$$[\![P]\!]_{DRW} \approx [\![P']\!]_{DW}$$

What is significant about this observation is the following: the delayed write semantics side-steps cyclic dependencies and out-of-thin-air behavior. Therefore, we ask whether (A) the delayed write semantics following a static program transformation can be made as relaxed as (B) the delayed read/write semantics, while keeping degenerate situations at bay. Such an approach emphasizes the distinction between static versus runtime transformations.

5.3 Delaying reads in a setting with conditionals

With the basic observability rule ironed out, we can then add other language constructs to the delayed read/write semantics of Section 5.2. Here we highlight challenges related to incorporating branching. We proceed with an example due to Batty et al. [9] involving load buffering and conditional execution.

In the code snippet in Listing 5.4, two threads start with a load followed by an if-statement whose condition depends on the value of the load. With the semantics

```
  T1               T2
r1 := x          r2 := y
if r1 == 42      if r2 == 42
  y := r1          x := 42
                 else
                   x := 42
```

Listing. 5.4: LB+ctrldata+ctrl-double [9].

we have designed so far, the if-statement would require the load to be “concretized”

```
$T1$                  $T2$
$r_1 := x$            $r_2 := y$
if $r_1 == 42$        $x\ := 42$
  $y := r_1$
```

Listing. 5.5: LB+ctrldata+po [9].

so the condition could be evaluated. The example would then be reduced to two sets of interleavings.

1. One set of interleavings in which the load of x in the first thread occurs early—meaning, before the second thread assigns 42 to x in either arm of its if-statement. In this case, there exist no writes to x that can service the load other then x's initialization to zero. The load of x would then dereference the zero value via R-DEREF, and 0 would be substituted into the first thread's term via R-SUBST. The condition in the first thread's if-statement would evaluate to false, and execution would terminate with $(r_1, r_2) = (0, 0)$.

2. The other set of interleavings encompass the cases in which the load of x occurs after the assignment of 42 to x. Since the load of x occurs after the assignment, there exist two values that the load can observe: 0 or 42. When 42 services the load, then execution terminates with $(r_1, r_2) = (42, 0)$.

The delayed read/write semantics can thus produce $(0, 0)$ or $(42, 0)$ and it cannot produce $(42, 42)$. Note that $(0, 0)$ and $(42, 0)$ are also possible in a sequentially consistent semantics, and that $(42, 42)$ is also precluded under sequential consistency.

However, since in the second thread the if- and else-branches assign 42 to x, we can pull the assignment out and eliminate the branch. Such a simple compiler optimization leads to the program of Listing 5.5. With the control dependency eliminated, we are now allowed to execute the assignment of $x := 42$ *before* the load of y. In other words, the two remaining instructions of the second thread can now be swapped. This swap leads to a new possible end-result: $(r_1, r_2) = (42, 42)$.

```
$T1$                  $T2$
$r_: = x$             $r_2 := y$
if $r_1 == 42$        if $r_2 == 42$
  $y := r_1$            $x := 42$
```

Listing. 5.6: LB+ctrldata+ctrl-single [9].

We would like for the memory model to account for this compiler optimization. The memory model should to admit $(42, 42)$ as a result to not only the modified program of Listing 5.5 but also to the original program of Listing 5.4. In what follows, we explore how the operational semantics can be further relaxed when dealing with conditionals. This relaxation will cause the semantics to admit $(42,42)$ as an end-result.

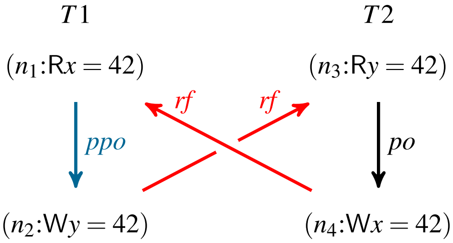

$T1$
$T2$
$(n_1{:}\mathrm{R}x = 42)$
$(n_3{:}\mathrm{R}y = 42)$
ppo
rf
rf
po
$(n_2{:}\mathrm{W}y = 42)$
$(n_4{:}\mathrm{W}x = 42)$

Figure 5.5: Candidate execution for load buffering with control dependency.

5.3.1 Branching on symbolic conditions

By incorporating branching on conditionals that contain symbolic values, we are able to admit $(r_1, r_2) = (42,42)$ as a possible end-result of Listing 5.4. Such a semantics is based on symbolic execution, where threads accumulate a path variable Ψ. When exploring the then-branch, a thread's path variable is "anded" with the symbolic branch condition, see R-SCOND$_t$. When exploring the else-branch, the variable is "anded" with the negation of the condition, see R-SCOND$_f$.

$$p\langle\sigma, \texttt{if}\ sb\ \texttt{then}\ t_1\ \texttt{else}\ t_2, \psi\rangle \ \rightarrow\ p\langle\sigma, t_1, \psi \wedge sb\rangle \quad \text{R-SCOND}_t$$

$$p\langle\sigma, \texttt{if}\ sb\ \texttt{then}\ t_1\ \texttt{else}\ t_2, \psi\rangle \ \rightarrow\ p\langle\sigma, t_2, \psi \wedge \neg sb\rangle \quad \text{R-SCOND}_f$$

Going back to Listing 5.4, the symbolic branching allows x to be assigned 42 in the second thread before the load of y is concretized. With the load delayed and the write executed, the first thread can then set y to 42. At this point, the second thread's load can finally commit and observe the value of 42, thus leading to $(r_1, r_2) = (42,42)$.

In equations (5.8), (5.9), and (5.10) we show how the then-branch leads to the desired end-result. Without loss of generality, we start reducing the second thread:[8]

[8] Note that with σ_1, σ_2, σ_1', etc, we are keeping track of the evolution of thread states. We do not, however, explicitly state the contents of the different thread states—although this content is taken into account in the reductions and in the accompanying the discussion.

$$\begin{aligned}
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2, \mathtt{let}\ r_2 = \underline{\mathtt{load}\,y}\ \mathtt{in}\ \mathtt{if}\ r_2 == 42\ \mathtt{then}\ x := 42\ \mathtt{else}\ x := 42, \top\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis && \xrightarrow{p_2(y?\_)_{n_2}}\\
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2', \underline{\mathtt{let}\ r_2 = n_2\ \mathtt{in}\ \mathtt{if}\ r_2 == 42\ \mathtt{then}\ x := 42\ \mathtt{else}\ x := 42}, \top\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis \parallel n_2\lBrace\sigma_2, ?y\rBrace && \xrightarrow{let}\\
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2', \mathtt{if}\ n_2 == 42\ \mathtt{then}\ x := 42\ \mathtt{else}\ x := 42, \top\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis \parallel n_2\lBrace\sigma_2, ?y\rBrace
\end{aligned} \tag{5.8}$$

Exploring the then-branch. We can then explore the first branch:

$$\begin{aligned}
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2', \underline{\mathtt{if}\ n_2 == 42\ \mathtt{then}\ x := 42\ \mathtt{else}\ x := 42}, \top\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis \parallel n_2\lBrace\sigma_2, ?y\rBrace && \xrightarrow{scond_t}\\
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2', \underline{x := 42}, \top \wedge n_2 = 42\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis \parallel n_2\lBrace\sigma_2, ?y\rBrace && \xrightarrow{p_2(x!42)_{n_3}}\\
&p_1\langle\sigma_1, \mathtt{let}\ r_1 = \mathtt{load}\,x\ \mathtt{in}\ \mathtt{if}\ r_1 == 42\ \mathtt{then}\ y := r_1, \top\rangle \parallel\\
&p_2\langle\sigma_2'',, \top \wedge n_2 = 42\rangle \parallel\\
&\quad n_0\llparenthesis\sigma_\bot, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\bot, y := 0\rrparenthesis \parallel n_2\lBrace\sigma_2, ?y\rBrace \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis
\end{aligned} \tag{5.9}$$

At this point, we want to reduce the first thread. The load from x can observe the original zero value of the variable and, because of the delayed write to x associated with n_3, the read can also observe 42. In either case load-buffering does not play a role: since there will be no alterations of x in the program, we can load a value right away—as opposed to post-pone the effects of the load. We will explore the two possible values for this load. First, in equation (5.10), we load, dereference, and substitute—in one combined step—the value of 42 when reading x. Later, in equation (5.11), we do the same combined reduction step when observing 0.

$$
\begin{aligned}
& p_1\langle \sigma_1, \texttt{let } r_1 = \underline{\texttt{load}\, x} \texttt{ in if } r_1 == 42 \texttt{ then } y := r_1, \top\rangle \parallel \\
& \sigma_2''\langle, \top \wedge n_2 = 42, \parallel\rangle \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis && \xrightarrow{p_1(x?\_)_{n_4}} \xrightarrow{deref_r} \xrightarrow{subst} \\
& p_1\langle \sigma_1', \underline{\texttt{let } r_1 = 42 \texttt{ in if } r_1 == 42 \texttt{ then } y := r_1}, \top\rangle \parallel \\
& \sigma_2''\langle, \top \wedge n_2 = 42, \parallel\rangle \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket && \xrightarrow{let} \\
& p_1\langle \sigma_1', \underline{\texttt{if } 42 == 42 \texttt{ then } y := 42}, \top\rangle \parallel \sigma_2''\langle, \top \wedge n_2 = 42, \parallel\rangle \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket && \xrightarrow{if} \\
& p_1\langle \sigma_1', \underline{y := 42}, \top\rangle \parallel p_2\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket && \xrightarrow{p_1(y!42)_{n_5}} \\
& p_1\langle \sigma_1'', , \top\rangle \parallel p_2\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket \parallel n_5\llparenthesis \sigma_1', y := 42\rrparenthesis && \xrightarrow{deref} \\
& p_1\langle \sigma_1'', , \top\rangle \parallel p_2\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, y, 42\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket \parallel n_5\llparenthesis \sigma_1', y := 42\rrparenthesis
\end{aligned}
\tag{5.10}
$$

By branching on symbolic condition, we had created the obligation that the value of 42 is read into x by n_2. In second to last configuration of equation (5.10), the write event n_5 services n_2 with the needed value of 42. The execution is, therefore, satisfiable: meaning that the path variables associated with each thread evaluate to true. Note that this run of the program matches by the candidate execution graph of Figure 5.5.

Going back to the end-state of equation (5.9), we explore what happens if the load of x in the first thread is serviced by n_0:

$$\begin{aligned}
& p_1\langle \sigma_1, \texttt{let}\ r_1 = \underline{\texttt{load}\,x}\ \texttt{in}\ \texttt{if}\ r_1 == 42\ \texttt{then}\ y := r_1, \top\rangle \parallel \\
& p_1\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis && \xrightarrow{p_1(x?\_)_{n_4}} \xrightarrow{deref_r} \xrightarrow{subst} \\
& p_1\langle \sigma_1', \underline{\texttt{let}\ r_1 = 0\ \texttt{in}\ \texttt{if}\ r_1 == 42\ \texttt{then}\ y := r_1}, \top\rangle \parallel \\
& p_2\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket && \xrightarrow{let} \\
& p_1\langle \sigma_1', , \top\rangle \parallel p_2\langle \sigma_2'', , \top \wedge n_2 = 42\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis \parallel \\
& \quad n_4\llbracket \sigma_1, x, 42\rrbracket
\end{aligned} \tag{5.11}$$

In the end configuration of equation (5.11), we again have the obligation that the value of 42 is read into x by n_2—this obligation comes from having branched based on a symbolic condition. However, at the end of execution (*i.e.,* in the end state of equation (5.11)), there does not exist a write that can service n_2 with the value of 42 for x. This execution, therefore, is not satisfiable and is discarded by the semantics.

Exploring the else-branch. Exploring the else-branch does not lead to an execution in which $(r_1, r_2) = (42, 42)$. Although the then-branch and the else-branch are symmetric (meaning that both paths contain a write of 42 to x), when it comes to exploring the else-branch, the semantics adds $\neg(n_2 = 42)$ to the path variable associated with the thread, as opposed to $n_2 = 42$. This modified path variable precludes $(r_1, r_2) = (42, 42)$.

Let us explore the else-branch of the if-statement starting from the end-state of equation (5.8). For brevity, we combine the branch and the subsequent store.

$$\begin{aligned}
& p_2\langle \sigma_1, \texttt{let}\ r_1 = \texttt{load}\,x\ \texttt{in}\ \texttt{if}\ r_1 == 42\ \texttt{then}\ y := r_1, \top\rangle \parallel \\
& p_2\langle \sigma_2', \underline{\texttt{if}\ n_2 == 42\ \texttt{then}\ x := 42\ \texttt{else}\ x := 42}, \top\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket && \xrightarrow{scond_f} \xrightarrow{p_2(x!42)_{n_3}} \\
& p_1\langle \sigma_1, \texttt{let}\ r_1 = \texttt{load}\,x\ \texttt{in}\ \texttt{if}\ r_1 == 42\ \texttt{then}\ y := r_1, \top\rangle \parallel \\
& p_2\langle \sigma_2'', , \top \wedge \neg(n_2 = 42)\rangle \parallel \\
& \quad n_0\llparenthesis \sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis \sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket \sigma_2, ?y\rrbracket \parallel n_3\llparenthesis \sigma_2', x := 42\rrparenthesis
\end{aligned} \tag{5.12}$$

Again, there is no point in delaying the load of x in the first thread (as there exist no future writes to x that can impact the load). And again, there exist two possibilities: we can read the initial zero value of x or read the value of 42 associated with n_3. If zero is loaded, then execution ends in the configuration depicted in equation (5.13). Note that the obligation of y in n_2 being different than 42 is satisfied by the initial value of y. Still, this end-state is fairly uninteresting, since we already knew that $(r_1, r_2) = (0,0)$ is a possible outcome of the model (the outcome is possible even under sequential consistency).

$$\begin{aligned}
&p_1\langle\sigma_1',,\top\rangle \parallel p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, y, 0\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis \parallel n_4\llbracket\sigma_1, x, 0\rrbracket
\end{aligned} \tag{5.13}$$

If, on the other hand, 42 is loaded instead of zero, then:

$$\begin{aligned}
&p_1\langle\sigma_1, \texttt{let } r_1 = \texttt{load}\, x \texttt{ in if } r_1 == 42 \texttt{ then } y := r_1, \top\rangle \parallel \\
&p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&\quad n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, ?y\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis && \xrightarrow{p_1(x?\_)_{n_4}}\xrightarrow{deref_r}\xrightarrow{subst} \\
&p_1\langle\sigma_1', \texttt{let } r_1 = 42 \texttt{ in if } r_1 == 42 \texttt{ then } y := r_1, \top\rangle \parallel \\
&p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&\quad n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, ?y\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis \parallel \\
&\quad n_4\llbracket\sigma_1, x, 42\rrbracket && \xrightarrow{let}\xrightarrow{if} \\
&p_1\langle\sigma_1', y := 42, \top\rangle \parallel p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&\quad n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, ?y\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis \parallel \\
&\quad n_4\llbracket\sigma_1, x, 42\rrbracket && \xrightarrow{p_1(y!42)_{n_5}} \\
&p_1\langle\sigma_1'',,\top\rangle \parallel p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&\quad n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, ?y\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis \parallel \\
&\quad n_4\llbracket\sigma_1, x, 42\rrbracket \parallel n_5\llparenthesis\sigma_1', y := 42\rrparenthesis && \xrightarrow{deref} \\
&p_1\langle\sigma_1'',,\top\rangle \parallel p_2\langle\sigma_2'',,\top\wedge\neg(n_2=42)\rangle \parallel \\
&\quad n_0\llparenthesis\sigma_\perp, x := 0\rrparenthesis \parallel n_1\llparenthesis\sigma_\perp, y := 0\rrparenthesis \parallel n_2\llbracket\sigma_2, y, 0\rrbracket \parallel n_3\llparenthesis\sigma_2', x := 42\rrparenthesis \parallel \\
&\quad n_4\llbracket\sigma_1, x, 42\rrbracket \parallel n_5\llparenthesis\sigma_1', y := 42\rrparenthesis
\end{aligned} \tag{5.14}$$

The observability rules captured by R-DEREF allow for n_2 to be serviced by the zero value of y in n_1 as well as by the value of 42 in n_5. However, the branching on the symbolic conditional adds the constraint that n_2 must not observe the value of 42. Thus, the execution terminates with n_2 being serviced by n_1. In other words, we have $(r_1, r_2) = (0, 42)$. This scenario, like the one in equation (5.14), is also present in a sequentially consistent execution.

The reductions above show that Listing 5.4 has an execution that matches the desired relaxations when executing in a delayed read/write semantics augmented with path variables and symbolic if-statements. Particularly, the run leading to equation (5.10) behaves as if the if-statement did not exist: the run allows the load of y to take effect after the setting of x to 42 in the second thread. In other words, the delayed read/write semantics can produce a run for Listing 5.4 as if it where a run for Listing 5.5, where the second thread's if-statement has been compiled out of the program.

There is, however, a problem with the type of relaxation introduced by R-SCOND$_t$ and R-SCOND$_f$. The relaxation allows for the result $(42,42)$ also for the code in Listing 5.6. However, when it comes to Listing 5.6, $(42,42)$ is considered to be out-of-thin-air! The justification for $(42,42)$ involves the circular reasoning as follows: By assuming that the second thread observes 42 when loading y, the condition $r_2 == 42$ is satisfied and x is written the value of 42. This leads to the first thread writing 42 to y, which justifies the initial assumption.

Delayed reads, path variables, and branching on symbolic conditions allow the semantics to be more relaxed, but to the point of admitting out-of-thin-air behavior. There exists an analogue problem in axiomatic semantics: when it comes to Listing 5.6, we want to forbid the candidate-execution of Figure 5.5 because it involved circular reasoning. Yet, the same candidate execution captures desired runs of Listing 5.4.

In order to fix the issue, a semantics should only be allowed to pull a statement out of an if-statement if the statement is executed in all branches of the if. However, when the semantics was augmented to support branching with conditionals, we allowed the write from inside the then-branch to be "swapped" with the load that preceded the branch. This swapping took place for both Listings 5.4 and 5.6. In other words, the semantics allowed for the swap without considering the existence of an else-branch.

One approach may be to explore all arms of a branch. For example, in the case of Listings 5.4, we would obtain two memory events for the two writes to x. These events would live in the same configuration. The event on the left captures the write to x in the then-branch and the one on the right corresponds to the else-branch. We also augment the memory events with the path variable of the executing thread. This augmentation is captured by condition $n_2 = 42$ on the left and $\neg(n_2 = 42)$ on the right:

$$n_3(\!|\sigma_2', x := 42, n_2 = 42|\!) \quad \| \quad n_3'(\!|\sigma_2', x := 42, \neg(n_2 = 42)|\!)$$

Such symbolic write events would not be able to service read events. We could, however, include reduction rules to manipulate these symbolic events. The semantics can be made to perform resolution on the conditions associated with memory events. For example, we can perform resolution on the path variables associated with the two memory events n_3 and n_3' above to obtain $m(\!|\sigma_2', x := 42|\!)$. This derived event represents the fact that the write of 42 to x in the second thread can be taken out of the branch, as in Listing 5.4. Such a semantics would thus admit

$(r_1, r_2) = (42, 42)$ as a result of Listing 5.4.

$$\frac{n\llparenthesis \sigma, x := v, a_1 \vee a_2 \ldots \vee c \rrparenthesis \qquad m\llparenthesis \sigma, x := v, b_1 \vee b_2 \ldots \vee c \rrparenthesis}{m\llparenthesis \sigma, x := v, a_1 \vee a_2 \ldots b_1 \vee b_2 \ldots \rrparenthesis} \qquad \frac{n\llparenthesis \sigma, x := v, \ \rrparenthesis}{n\llparenthesis \sigma, x := v \rrparenthesis}$$

Given that the else-branch is missing, the semantics would also preclude the result $(42, 42)$ for Listing 5.6. For one, without a corresponding event with which to perform resolution, there would be no path to converting the "conditional" memory event n_3 into a concrete memory event. And, for as long as the event remains "conditional," its value could not be used in a computation.

5.4 Channels and other considerations

With the introduction of delayed reads, we must consider whether it is reasonable to send a symbolic value on a channel. For example, in the code

$$\begin{array}{l} r := x \\ c \leftarrow r \end{array}$$

the local variable r can now be m where m is the label of a read event $m\llbracket \sigma, ?x \rrbracket$. This read event is representing the fact that the concrete value of x has not yet been determined. Like branching on a symbolic value, sending a symbolic value over a channel opens the door to new behavior that is not present in a delayed-write semantics. Since the state σ in the read event contains happens-before and shadowing information, sending a handle (as opposed to a concrete value) seems like a plausible design choice. When it comes to Listing 5.7, the transmission of a read handle is also a necessity.

$$\begin{array}{ll} \quad T1 & \quad T2 \\ r_1 := x & r_2 := y \\ & c \leftarrow r_2 \\ y \ := r_1 & x \ := 42 \end{array}$$

Listing. 5.7: Load buffering and send on an "initial" buffered channel

Let c in Listing 5.7 be a channel of capacity greater than one. The send into c by thread $T2$ is non-blocking. Since there are no other interactions with the channel, $T2$ learns the empty state $\sigma_\perp$ when sending.[9] Given that for all σ we have that

[9] See the channel communication rules of Section 2.5.3.2 and the R-SEND rule in particular.

$\sigma + \sigma_\perp = \sigma$, the send by thread $T2$ does not change the thread's state. Although the send deposits a message into the channel, the message remains unredeemed for the life of the program. For these reasons, the channel operation behaves like a no-op. By removing c and the send operation, it would be reasonable for a compiler to optimize Listing 5.7 into Listing 5.2. Therefore, we would like the two listings to produce the same set of results. However, in order for Listing 5.7 to produce the same results as Listing 5.2, the value of r_2 sent on c cannot be concrete; it has to be symbolic.[10]

In fact, we need be able to send symbolic values even if the send cannot be compiled out of the code. For example, if we were to remove $T2$'s send from Listing 5.8, $T3$ would become non-terminating. However, the synchronization between $T2$ and $T3$ does not alter the fact that $T1$ and $T2$ are not synchronized. Therefore, we still want to have $(r_1, r_2) = (42, 42)$ as a possible out-come.

$T1$	$T2$	$T3$
$r_1 := x$	$r_2 := y$	
	$c \leftarrow r_2$	$\leftarrow c$
$y \;:= r_1$	$x \;:= 42$	

Listing. 5.8: Load buffering and send on an "initial" buffered channel, three threads

5.5 Conclusion

In this chapter we have provided sketches for a semantics with delayed reads and well as delayed writes. The reduction rules in Section 5.2 lay out the ground work. Rule R-DEREF faithfully follows the Go memory model observability rules, which specify when a read is allowed to observe a write.

In Section 5.3, we show that there exist relaxations involving delayed reads and branches that are not accounted for in the reduction rules of Section 5.2. We explore ways to further relax the operational semantics by branching on symbolic reads—this addition gives the semantics the flavor of speculative execution. The relaxation afforded by symbolic branching, however, goes too far: the augmented semantics admits out-of-thin-air behavior. We draw parallels between operational and axiomatic semantics and conclude Section 5.3 with ideas on how to potentially deal with branches. Section 5.4 introduces channels and argues for the need to send symbolic values (*i.e.,* read event labels) over channels.

[10] By sending a symbolic value, we are able to obtain a reduction like in Example 4 and the end-result of $(r_1, r_2) = (42, 42)$.

We point at open questions throughout the chapter. Here, we remind the reader of the importance of establishing the DRF-SC guarantee for the delayed read/write semantics. This may involve the proof in Chapter 2 as a stepping stone.

Finally, it is worth noticing that the operational semantics discussed at the end of Section 5.3 starts to have the same flavor as its axiomatic counterparts. The operational semantics is, in essence, collecting concrete and symbolic memory events. These events carry information that allow us to relate them to one another; for example, they are related through their happens-before and shadow sets and through path variables. Like events of an axiomatic semantics, the memory events of the operational semantics can thus be interpreted as vertices that can be connected to one another. It would be interesting to further explore the parallels between operational and axiomatic semantics, and to seek a correspondence between the two approaches.

Conclusion and extensions 6

Each society and each individual usually explore only a tiny fraction of the horizon of possibilities.
Yuval Noah Harari, *Sapiens*

In the introduction, we set three goals for this thesis. The first was to *propose and evaluate a memory model for message passing systems using an operational semantics as formalism.* In Chapter 2, we presented a memory model inspired by the Go programming language, where threads synchronize by exchanging messages over channels. In that chapter, we also addressed the goal of *investigating how relaxed the model can be while keeping out-of-thin-air behavior at bay and while supporting the DRF-SC guarantee—which says that Data-Race Free executions are Sequentially Consistent.* This second goal was addressed by showing the absence of out-of-thin-air (OOTA) behavior and by proving the DRF-SC guarantee for a memory model supporting delayed writes.

The formalisms introduced in Chapter 2 can, however, be generalized by incorporating delayed reads (*aka.* load buffering). The further relaxation associated with delayed reads, however, presents significant challenges with respect to non-termination and the introduction of OOTA behavior. These challenges require further investigation, as discussed in Chapter 5.

In Chapter 3, we explored data-race detection and showed, in Chapter 4, how our formalisms helped unveil and remedy a bug in the Go runtime. These chapters support our third goal of *relating the theoretical model to its "material" counterpart and source of inspiration: the Go programming language.*

Here, we discuss additional extensions of our work, such as the potential for improvements in data-race detection, the verification of Go programs via model checking, and the exploration of language constructs at the intersection of concurrency and distribution.

6.1 Data-race detection

Even though Go is a language with channels, the infrastructure underlying the Go data-race detector is based on lock semantics! Locks have been the prevalent mode of synchronization partially because they can easily be mapped down to hardware, but employing locks are not always the best approach. As part of this thesis, we introduced a primitive (the *release-acquire-exchange* of Chapter 4) that closely matches the semantics of channels into the underlying data-race detector used in Go. By doing so, we not only repaired the detector's functionality but also improved its performance and memory consumption—see Section 4.5. We would like to see further improvements.

One a practical side, the underlying race detector employed by Go is written in C/C++ and reside in a different repository than the language itself. This complicates the process of upgrading the detector and keeping it in sync with the language. Although powerful, the detector is overly complex: it was originally built for detecting races in C++11, which means that large portions of the code base do not apply to Go. This added complexity creates a high barrier of entrance when it comes to modifying and improving the detector.

Go is a language for systems design and is thus suited for implementing a data-race detector. An implementation in Go, for Go, is not just an exercise of the "eat your own dog food" principle [103]. More importantly, a tailor-made implementation would be simpler and thus easier to modify and extend than the existing detector's code base. This simplicity would likely translate to improvements and to new features for the end user. We are particularly interested in exploiting the concept of *stale vector-clock entries*, defined in Chapter 3, Section 3.5.1. We conjecture that the memory consumption associated with a vector-clock based data-race detector can be reduced by taking the concept of stale entries into account.

6.2 Model checking and predictive data-race detection

Dynamic data-race detectors can flag synchronization issues in individual runs of a program. *Predictive data-race detection* attempts to extrapolate from an individual run in order to flag races in alternate executions. When checking Go programs for data races, we could explore alternate executions by reordering "channel races," meaning, when more than one thread competes on sending or receiving from a channel. According to the Go memory model, messages that sit next to each other in a channel's buffer are not related by happens-before. Therefore, it is possible to derive alternate runs of a program by swapping the order of messages in a channel.

Finding alternate execution paths can also be helpful from the point of view of model checking. Therefore, message swapping is an interesting avenue to explore. However, message reordering requires careful analysis. As an example, let $|c| > 1$ and $|c_2| = 0$ in the snippet below:

$T1$	$T2$
sd c	
sd c_2	rv c_2
	sd c

The messages sent by $T1$ and $T2$ may sit side-by-side in c's buffer. According to the happens-before rules of the Go memory model, these messages are unrelated. We would be tempted to want to swap them, thinking that there is an execution of the program where $T2$ sends before $T1$. However, because of the message exchange over c_2, such execution does not exist: the send and the receive on c_2 by $T1$ and $T2$ causes $T1$'s actions to be in the past of $T2$'s.

6.3 Bridging the gap between concurrency and distribution

In the 1990s, we learned from experience that simply making remote-procedure-calls available to the application programmer is not sufficient when it comes to developing robust distributed systems [101]. Although similar in many ways, there exist challenges particular to distribution that do not manifest themselves in a purely concurrent setting. In a concurrent system, we assume that cooperating agents fall within a single overarching environment. In Go, for example, these agents (goroutines) are under the umbrella of the Go runtime. The runtime, with the help of the operating system, allows us to assume that no messages are lost during transmission. When it comes to distributed systems, however, there exists no such point of authority—at least not without making strong assumptions about the network connection. Given that a network delay may be indistinguishable from a node failure, we are forced to deal with reliability issues in a distributed system.

Despite being useful in the context of cloud computing, Go channels are not meant for the development of distributed systems but rather as a synchronization mechanism in shared memory systems. An active area of research involves programming paradigms that can address the particular needs of distributed systems. Conflict-free replicated data types (CRDTs) [82] and lattice-based data structures [22, 55] have emerged as promising approaches. What would it take to turn Go into a language for expressing distributed computations? How can we best evolve message passing systems in light of recent advances in data structures and type systems for distributed programming.

Appendix A

A.1 The weak semantics simulates the strong

Proof of Lemma 2 (simulation). To prove the $\gtrsim$-relationship between the respective initial configurations, we need to establish a simulation relation, say $\mathcal{R}$, between (well-formed) strong and weak configurations such that S_0 and P_0 are in that relation. To ease the definition of the relation $\mathcal{R}$ connecting the strong and the weak semantics, we introduce a few abbreviations.

Configurations for the weak semantics contain additional book-keeping information, such as identifiers for write events and the thread local views on the global configuration. Given a configuration in the weak semantics, a corresponding strong configuration is one where all the extra information is removed. More formally: The *erasure* of a goroutine $p\langle\sigma,t\rangle$, written $\lfloor p\langle\sigma,t\rangle\rfloor$ is defined as $\langle t\rangle$. The erasure of forward channel $c_f[q]$, written $\lfloor c_f[q]\rfloor$, replaces each element (v,σ) of the queue by v. For a backward channel $c_b[q]$, the σ-elements are replaced by unit values. The special end-of-transmission value $\perp$ remains unchanged. We use erasure correspondingly also on whole configurations.

Given a strong well-formed configuration S, we allow ourselves to interpret it as a mapping from shared variables to their values, writing $\sigma_S(z) = v$ if S contains a write event of the form $(\!|z{:=}v|\!)$. This interpretation is independent of the configurations' syntactical representation, meaning $S_1 \equiv S_2$ implies $\sigma_{S_1} = \sigma_{S_2}$. Furthermore, according to this interpretation, σ_S is a well-defined function when S is well-formed (which means there exists one write event per shared variable). For weak configurations P, there is no uniqueness of write events for a given shared variable. Analogously, we could define a "multi-valued" state $\sigma_P(z) = \{v_1,\ldots,v_2\}$ collecting all values written to z in *any* write event. We need, however, a mild refinement of that notion for the definition of simulation: We must record the status of the shared variables *from the perspective* of an individual thread. In the weak semantics, goroutines maintain in σ information about which write events are observable for that goroutine, namely all those which are not "shadowed." So, given a well-formed configuration P and a set N of names, we define $\sigma_P^{\overline{N}}$ as follows:

$$\sigma_P^{\overline{N}}(z) = \{v \mid m(\!|z{:=}v|\!) \in P \text{ and } m \notin N\} \ . \tag{A.1}$$

We then define the relation $\mathcal{R}$ between well-formed strong and weak configuration over the same set of shared variables as follows: $S \mathrel{\mathcal{R}} P$ if $\lfloor P\rfloor = S$ (as far as goroutines and channels is concerned) and furthermore, for each goroutine $p\langle(\_,E_s),t\rangle$ in P, and all shared variables z,

$$\sigma_S(z) \in \sigma_P^{\overline{E_s}}(z) \ . \tag{A.2}$$

Case: R-WRITE$_s$: $p\langle z := v;t\rangle \parallel (\!|z{:=}v'|\!) \xrightarrow{\tau}_s p\langle t\rangle \parallel (\!|z{:=}v|\!)$
By definition, $S\,\mathcal{R}\,P$ implies that P contains a goroutine $p\langle\sigma, z := v;t\rangle$. Doing the corresponding weak step $P \xrightarrow{\tau}_w P'$ yields

$$P' = \nu m\ (p\langle\sigma',t\rangle \parallel m(\!|z{:=}v|\!))$$

where $\sigma' = (E'_{hb}, E'_s)$. Since m is a fresh name, it is not mentioned in any shadow set of any thread, in particular $m \notin E'_s$. Consequently, S' and P' satisfy the condition from equation (A.2) for variable z. The condition holds for the remaining shared variables as well: it was assumed to hold for S and P prior to the steps, and write-steps do not affect variables other than z. Consequently, $S'\,\mathcal{R}\,P'$ as required.

Case: R-READ$_s$: $p\langle \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!) \xrightarrow{(z?v)}_s p\langle \texttt{let}\ r =\ v\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!)$
$S\,\mathcal{R}\,P$ implies that P contains $p\langle(\_,E_s), z := v;t\rangle$ and write events $m(\!|z{:=}v|\!)$ (there may be more than one for z and v, but with different identifiers); specifically condition (A.2) guarantees that there exists one $m(\!|z{:=}v|\!)$ such that $m \notin E_s$, which enables R-READ$_w$ for P such that $P \xrightarrow{(z?v)}_w P'$ with $S'\,\mathcal{R}\,P'$, as required.

The remaining cases are analogous or simpler, establishing that $\mathcal{R}$ is a simulation relation. It is immediate that the corresponding initial configurations are related, *i.e.,* $S_0\,\mathcal{R}\,P_0$. Thus $P_0 \gtrsim S_0$, which concludes the proof. □

A.2 Proofs via a weak semantics augmented with read and write events

This section contains supplementary material and proofs for the lemmas of Section 2.6. In particular, the material here allows us to carry out the harder direction of the simulation proof of Section 2.6.2, namely that the strong semantics simulates the weak one for race-free programs.

We start in Section A.2.1 augmenting the weak semantics with additional information which has no relevance aside from assisting the proofs. Section A.2.2 covers properties of the augmented semantics.

A.2.1 Augmenting the weak semantics

This section presents an "alternative" representation of the weak semantics of Section 2.5. The steps of the reformulation here are in one-to-one correspondence to the previous ones, with the difference that now, more information is stored as part of the configurations. In particular, the weak semantics from Section 2.5 makes use of write events as part of configurations. Read steps, however, were not treated the same way. The variant semantics augments the weak one by: 1) recording read events in addition to write events, and 2) storing in the read and write events the local state σ of the issuing thread at the point in time the read/write step was taken.

The configurations introduced in equation (2.6) on page 27 are therefore adapted to contain events of the following form:

$$m(\!|\sigma, z := v|\!)_p \quad \text{and} \quad m[\![\sigma, ?z]\!]_p \,, \tag{A.3}$$

where $m(\!|\sigma, z := v|\!)_p$ are write events augmented with the local state σ and identity p of the issuing thread and $m[\![\sigma, ?z]\!]_p$ are read events augmented analogously.

Notation 14 (Events)**.** *We use e for events and r and w for read and write events specifically. For two different events, we generally assume that their identities are different. It is an invariant of the semantics that the labeling of the events are indeed unique. Furthermore, let e be an event with identifier m and referring to variable z. Instead of writing $m \in E_s$ for some shadowed set E_s, we allow ourselves to write $e \in E_s$. Similarly, we write more succinctly $e \in E_{hb}$ instead of $(m, z) \in E_{hb}$.*

From the rules of Figure 2.11, only the read and write steps require adaptation. See Figure A.1 for the augmented rules, which behave exactly as the originals except that the steps now record additional information as part of the configuration.

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (m, !z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(m)}{p\langle\sigma, z := v; t\rangle \;\to\; \nu m\ (p\langle\sigma', t\rangle \parallel m(\!|\sigma, z := v|\!)_p)}\ \textsc{R-Write}_\sigma$$

$$\frac{\sigma = (\_, E_s) \qquad m \notin E_s \qquad \mathit{fresh}(m')}{\begin{array}{l} p\langle\sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel \quad m(\!|\_, z := v|\!)_\_ \qquad \to \\ \nu m'\ (p\langle\sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel \quad m(\!|\_, z := v|\!)_\_ \quad \parallel m'[\![\sigma, ?z]\!]_p) \end{array}}\ \textsc{R-Read}_\sigma$$

Figure A.1: Operational semantics: Read/write rules with augmented read/write events

The augmentation of the rules yield an operational semantics that is obviously equivalent to the one from Section 2.5: It is easy to envision the simulation relation as a function from the augmented semantics to the weak semantics (the function simply removes the augmented information). This augmented semantics, however, allows us to prove the lemmas of Section 2.6.

A.2.2 Additional concepts and lemmas

In the following, we use $\to_w$, $\to_w^*$, etc., when referring to the steps of the augmented weak semantics, which we will, from now on, refer to simply as the "weak semantics" (unless stated otherwise).

We define three binary relations between events given the augmented read and write events. First, the *happens-before* relation, which can now be gathered from the augmented event information. Events are considered *concurrent* if unordered by the happens-before relation. Combinations of read-write resp. write-write events are in *conflict* if they are concurrent and concern the same variable. These definitions generalize Definition 7 from the main part of the paper.

Definition 13 (Binary relations on events)**.** *Let e_1 and e_2 be two different events, with E^2_{hb} the happens-before set of e_2 and m_1 the identity of e_1.*

1. e_1 happens-before e_2*, written* $e_1 \rightarrow_{\mathsf{hb}} e_2$*, if* $m_1 \in E^2_{hb}$*.*

2. e_1 *and* e_2 *are concurrent, written* $e_1 \parallel\!| e_2$*, if neither* $e_1 \rightarrow_{\mathsf{hb}} e_2$ *nor* $e_2 \rightarrow_{\mathsf{hb}} e_1$*.*

3. e_1 *and* e_2 *are* in conflict*, written* $e_1 \# e_2$*, iff* $e_1 \parallel\!| e_2$*, both event concern the same variable, and one of the events is a write.*

We denote read/write conflicts as $\#^{rw}$ and write/write as $\#^{ww}$. We also say that a configuration contains a conflict if it contains two different events which are in conflict. Note that we need the augmented notion of configurations to obtain this definition; the original notion of weak configuration contains not enough information to "detect" conflicts (not to mention, that read events were not even recorded). Note that the definition of $\rightarrow_{\mathsf{hb}}$ is slightly asymmetric: only the happens-before information from e_2 is relevant when defining $e_1 \rightarrow_{\mathsf{hb}} e_2$ (as e_1 does not have information about events that "happen-after"). See also Lemma 15, stating that $\rightarrow_{\mathsf{hb}}$ is a partial order.

Lemma 15 (Simple properties of event relations)**.**

- *$\#$ and $\parallel\!|$ are symmetric, irreflexive by definition, but not transitive.*
- *$\#^{ww}$ is not transitive.*

Furthermore, all reachable configurations we have the following invariants:

- *$\rightarrow_{\mathsf{hb}}$ is a strict partial order (i.e., acyclic, transitive, and irreflexive).*
- *Assume two events e_1 and e_3 with p_1 the issuing process of e_1 and p_2 the one of e_2. Then $e_1 \parallel\!| e_2$ implies $p_1 \neq p_2$.*

Proof. $\#$ and $\parallel\!|$ are symmetric by definition. The invariants are proven by straightforward induction on the steps of the operational semantics. □

Finally, we define the notion of write events being observable by read-events. This again is a generalization of the corresponding notion of write events begin observable by *processes* from Definition 7. A write event is observable by a read event unless it is either "shadowed," *i.e.,* it is mentioned in the shadow set of the read event, or the write event "happens-after" the read event, *i.e.,* the write-event mentions the read-event in its happens-before set. The two conditions for observability correspond directly to the formulation in the informal description of the happens-before memory model [40].

Definition 14 (Observable writes by a read event)**.** *Assume two events on the same variable z: one being a read event r with shadow set E_s^r and the other a write event with happens-before set E_{hb}^w. The write event w on z is* observable by *the read event r on z, written $w \rightarrow_{\mathsf{o}}^z r$, if*

1. $w \notin E_s^r$ *and*

2. $r \notin E_{hb}^w$.

We also write $w \rightarrow_{\mathsf{o}} r$ if the variable which "connects" the events needs no mention. With this, we can define

$$W_P^{\mathsf{o}}(z@r) = \{w \in P \mid w \rightarrow_{\mathsf{o}}^z r\}$$

as the set of write-events observable by the read event r in a given (augmented) configuration P. This is analogous to the set of write events $W_P^{\mathsf{o}}(z@p)$ observable by process p (see Definition 7). Note, however, that the transition-based definition from Section 2.6.2 does *not* include condition (2) from Definition 14. Even if the two definitions differ concerning that condition, they are intuitively capturing the same concept: In the earlier Definition 7, the observability referred to a read-event r just about to occur, namely being executed by a process. Thus, there was no need to mention write events for which $r \rightarrow_{\mathsf{hb}} w$ would hold, as they could not be part of the configuration at that point. Definition 14 of observability by read events in the augmented semantics takes into account "historic" read events and therefore, condition (2) is needed as old read events cannot observe writes that are *guaranteed* to have occurred in the future (according to the happens-before relation). Write-events that just *coincidentally* were issued in a later reduction step but otherwise unordered via the happens-before relation may well be observable by such a read event.

We now make the informal definition of race from the discussion in page 38 precise. There we said a race is a situation in which two different threads access the same shared variable, at least one of the accesses is a write, and the accesses are not ordered by the happens-before relation. In light of the augmentation done to the weak semantics, this definition can easily be made precise.

Definition 15 (Data race)**.** *Let P be a reachable configuration in the augmented semantics. P has a r/w-race iff $P \rightarrow_w^* P'$ with P' containing a r/w-conflict. Analogously for w/w-races resp. w/w-conflicts.*

A.2.2.1 General invariant properties

See also Section 2.6.2.1 in the main part.

Lemma 16 (Invariants)**.** *For all reachable configurations, we have the following invariants.*

1. *For all events e resp. processes with local state (E_{hb}, E_s), $E_s \subset E_{hb}(z)$.*
2. *$w \,|||\, r$ implies $w \rightarrow_{\mathsf{o}} r$.*
3. *For each read event r, there exists a write event w with $w \rightarrow_{\mathsf{o}} r$ and not $w \,|||\, r$.*
4. *For each read event r, there exists a write event w with $w \rightarrow_{\mathsf{o}} r$ and $w \rightarrow_{\mathsf{hb}} r$.*

Proof. Part 3 or alternatively part 4 is used in the proof of Lemma 19. By straightforward induction. □

Proof of the invariants Lemma 3. A straightforward consequence of the corresponding property for read and write events of the augmented semantics from Lemma 16. □

Proof of Lemma 4 ("consensus possible"). The property holds for an initial configuration P_0 because:

- it contains one write event for each shared variable and
- the initial process's shadowed set is empty.

Therefore, every process observe, for each variable, the same initial value. Assuming $W^{\circ}_{P_i}(z@p) \neq \emptyset$ where $P_0 \rightarrow^*_w P_i$ then, for each possible step that P_i can take we argue as follows:

Case: Congruence, local steps, R-READ, R-MAKE, R-CLOSE, and R-GO
None of the rules modify W_P. In addition, congruence, local steps, R-READ, R-MAKE and R-CLOSE do not alter thread-local states, which means that shadowed sets are unchanged. R-GO creates a new goroutine that inherits the thread-local state of the parent.

Case: R-WRITE
R-WRITE adds a fresh write event, which, by definition, is not in the shadowed set of any process and, therefore, is in $\bigcap_{p \in P_{i+1}} W^{\circ}_{P_{i+1}}(z@p)$.

Case: R-SEND
Let E_s be the sender's shadowed set at P_i. According to the definition of R-SEND, the sender's shadowed set at P_{i+1} is $E_s \cup E''_s$ where E''_s is the shadowed set of some thread in a configuration P_j where $j < i$. By the induction hypothesis, there exists a write event m that is not in any process's shadowed set at P_i. Since shadowed sets are monotonically increasing, $m \notin E''_s$. Since $m \notin E_s$ and $m \notin E''_s$, then $m \notin E_s \cup E''_s$. This means m is not in the sender's shadowed set at P_{i+1}, which, coupled with the fact that no other threads' shadowed set are modified by the R-SEND rule, we have that $\bigcap_{p \in P_{i+1}} W^{\circ}_{P_{i+1}}(z@p)$.

Case: R-REC, R-REC$_\perp$
Analogous to R-SEND.

Case: R-REND
Let E_s and E'_s be the sender's and receiver's shadowed sets at P_i. By the induction hypothesis, there exists a write event m that is not in any process's shadowed set at P_i; therefore, $m \notin E_s$ and $m \notin E'_s$ in specific. By the definition of R-REND, the sender's and receiver's shadowed sets at P_{i+1} is $E_s \cup E'_s$. Since $m \notin E_s$ and $m \notin E'_s$, then $m \notin E_s \cup E'_s$. Finally, since at P_{i+1} the sender's and receiver's shadowed sets do not contain m, and since no other threads' shadowed set were modified in the transition $P_i \rightarrow P_{i+1}$, we have that $\bigcap_{p \in P_{i+1}} W^{\circ}_{P_{i+1}}(z@p)$.

□

The next lemma expresses a property concerning observability and conflicts. Each read event may well observe more than one write-event; this corresponds to the situation where a read step yields a non-deterministic result. The lemma establishes that this ambiguity in observability is a symptom of *conflicts.* As the notion of conflicting events in the augmented weak semantics is in close correspondence with the notion of races (as established in Definition 15), the lemma implies that for race-free programs, there is no ambiguity when observing write events.

Lemma 17 (Observability and conflicts)**.** *The weak semantics has the following invariant: If $w_1 \rightarrow^x_{\mathsf{o}} r \leftarrow^x_{\mathsf{o}} w_2$ for two different write events w_1 and w_2, then $w_1 \#_x w_2$ or $w_1 \#_x r$ or $w_2 \#_x r$.*

Proof. By straightforward induction on the steps of the (augmented) weak semantics. □

Note that the fact that two write events w_1 and w_2 are observable by a read event does not imply that $w_1 \# w_2$. It may well be the case that $w_1 \rightarrow_{\mathsf{hb}} w_2$ and both are concurrent wrt. the read event. If, in particular $w_1 \rightarrow_{\mathsf{hb}} w_2$, $w_1 \rightarrow_{\mathsf{hb}} r$, and $w_2 \parallel\!| \, r$, then $w_2 \# r$ but w_1 is not in conflict with any of the other two events.

A.2.2.2 Race-free resp. conflict-free reductions

See also Section 2.6.2.2 in the main part of the paper.

Lemma 18 (Uniqueness of observability)**.** *Let P be a reachable, conflict-free configuration in the augmented semantics. If P is race-free and $P \rightarrow_w^* P'$, then for all events in P' and all variables z we have*

$$|\{w \mid r \leftarrow_{\mathsf{o}}^{z} w\}| \leq 1 \tag{A.4}$$

Proof. Assume for a contradiction that there exists in P' two different write events w_1 and w_2 for some variable and some read event such that $w_1 \rightarrow_{\mathsf{o}} r$ and $w_2 \rightarrow_{\mathsf{o}} r$. By Lemma 17, this implies that P' contains at least two conflicting events. With Definition 15, the existence of conflicting events contradicts the assumption of race-freedom, which concludes the proof. □

Corollary 19. *Let P, P' and z be given as in Lemma 18. Then we have*

$$|\{w \mid r \leftarrow_{\mathsf{o}}^{z} w\}| = 1\ . \tag{A.5}$$

Proof. A direct consequence of 18 and of Lemma 16(3) (or alternatively of Lemma 16(4)). □

Proof of Lemma 5 (no concurrent write when it counts). A direct consequence of the equivalence of races and conflicts from Definition 15. Assume for a contradiction $P \xrightarrow{(z?)p}_w P'$ and $W_P^{|||}(z@p) \neq \emptyset$. Then P' contains two events r and w with $r\#w$. With Definition 15, this contradicts the assumption that P_0 has no r/w race. The case for w/w races is analogous. □

Lemma 20 (Unique observability when it counts)**.** *Assume $P_0 \rightarrow_w^* P$ with P_0 race-free. If $P \xrightarrow{(z?)p}_w$ or $P \xrightarrow{(z!)p}_w$, then*

$$W_P^{\mathsf{o}}(z@p) = \{m\}\ . \tag{A.6}$$

Proof. For the write step: assume that there are two different observable writes w_1 and w_2. By Lemma A.2.2.2, $W_P^{|||}(z@p) = \emptyset$. By Definition 7, that means all observable writes are in happens-before relation, *i.e.,* $W_P^{\mathsf{o}}(z@p) = W_P^{\mathsf{hb}}(z@p)$. In particular, both w_1 and w_2 are in happens-before relation to process p at that point. For the case $w_1 \rightarrow_{\mathsf{hb}} w_2$, w_1 is unobservable by p, contradicting the assumption (the case $w_2 \rightarrow_{\mathsf{hb}} w_1$ is symmetric). Remains the case where w_1 and w_2 are unordered by $\rightarrow_{\mathsf{hb}}$, in other words, $w_1 \parallel w_2$, which implies $w_1\#w_2$. With Definition 15, that contradicts the assumption of race-freedom. The case for a read-step is analogous (alternatively it follows from Lemma 18). □

As an easy consequence, we obtain the following consensus lemma:

Proof of Lemma 6 ("race-free consensus when it counts"). A direct consequence of unique observability from Lemma 20 and the possible consensus property from Lemma 4. □

Proof of Corollary 7. A direct consequence of the consensus Lemma 6. □

The next property is central for the guarantees of the weak semantics. It states that, under the assumption of race freedom, at each point in time each variable has *exactly one "real" value*. In other words, for each variable, there is exactly one write commonly observable across all processes. If one would focus on one particular process (or a proper subset as opposed to all processes as the lemma does), then the set of observable writes may be larger than one. If a process or a set of processes are in a situation where there is *more* than one observable write, it simply means that those process will not do any observations until this nondeterminism is resolved. Doing a read-step in this situation would contradict the assumption of race-freedom (see Lemma 6).

Note that the configurations in the weak semantics do not contain any explicit information which *marks* a particular write event as "the" value (also not in the augmented weak semantics). Having a consensus value is not a feature of the semantics *per se*; instead, it hinges on the assumption that the program being executed is race-free.

Indeed, the existence of exactly one unique consensus value is the core of the *DRF-SC* guarantee (*i.e.,* in the absence of data races, the weak semantics behaves like the strong, sequentially consistent one). More technically, when establishing the connection between the strong and the weak semantics, relating the weak and the strong configurations obviously will make the "consensus" value of the weak semantics the one used in the strong one. Without the race-free consensus lemma, the construction would not be well-defined: the erasure $\lfloor \_ \rfloor$ from Definition 9 would not be a function, resp. would not yield well-formed strong configurations.

Proof of Lemma 8 (race-free consensus). By straightforward induction on the steps of the operational semantics. The property clearly holds for any initial configuration. The crucial case is when writing to a variable. So, assume $P \xrightarrow{(z!)p}_w P'$. By Lemma 5(1), there are no concurrent writes for p before the step, *i.e.,* $W_P^{|||}(z@p) = \emptyset$. By Definition 7, that means all observable writes are in happens-before relation, *i.e.,* $W_P^{\circ}(z@p) = W_P^{\mathsf{hb}}(z@p)$.[1] Consequently, after the $\xrightarrow{(z!)p}_w$ step of the weak semantics, all those observable write events are shadowed for p in P', thereby becoming unobservable by p. As a result, the *only* write-event observable by p is the one just executed by step $P \xrightarrow{(z!)p}_w P'$. This is a *new* write event in P with a fresh identity, say,

[1] One could establish that there is exactly one such event, but it is not needed for the proof. The important property here is that there are no concurrent observable writes.

m', which consequently is not mentioned in the shadow set of any process. Therefore, $\bigcap_{p_i \in P'} W_P^{\circ}(z@p_i) = \{m'\}$, establishing the invariant for the post-configuration P'. □

Bibliography

[1] Adve, S. V. and Gharachorloo, K. (1995). Shared memory consistency models: A tutorial. Research Report 95/7, Digital WRL.

[2] Adve, S. V. and Hill, M. D. (1990). Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a):2–14.

[3] Alglave, J., Maranget, L., and Tautschnig, M. (2014). Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2).

[4] Alrahman, Y. A., Andric, M., Beggiato, A., and Lluch-Lafuente, A. (2014). Can we efficiently check concurrent programs under relaxed memory models in Maude? In Escobar, S., editor, *Rewriting Logic and Its Applications – 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 21–41. Springer Verlag.

[5] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

[6] Aspinall, D. and Ševčík, J. (2007). Java memory model examples: Good, bad and ugly. *Proc. of VAMP*, 7.

[7] Back, R. and von Wright, J. (1998). *Refinement Calculus – A Systematic Introduction*. Graduate Texts in Computer Science. Springer.

[8] Banerjee, U., Bliss, B., Ma, Z., and Petersen, P. (2006). A theory of data race detection. In *Proceedings of the 4th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), PADTAD 2006, Portland, Maine, USA, July 17, 2006*, pages 69–78.

[9] Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., and Sewell, P. (2015). The problem of programming language concurrency semantics. In Vitek, J., editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Verlag.

[10] Batty, M., Owens, S., Sarkar, S., and Weber, T. (2011). Mathematizing C++ concurrency. In *Proceedings of POPL '11*, pages 55–66. ACM.

[11] Becker (2011). Programming languages — C++. ISO/IEC 14882:2001.

[12] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83.

[13] Blackshear, S., Gorogiannis, N., O'Hearn, P. W., and Sergey, I. (2018). RacerD: compositional static race detection. *PACMPL*, 2(OOPSLA):144:1–144:28.

[14] Blanchette, J. C., Weber, T., Batty, M., Owens, S., and Sarkar, S. (2011). Nitpicking C++ concurrency. In Schneider-Kamp, P. and Hanus, M., editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 113–124. ACM.

[15] Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78. ACM.

[16] 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*, MSPC '14, pages 7:1–7:6, New York, NY, USA. ACM.

[17] Boudol, G. and Petri, G. (2009). Relaxed memory models: An operational approach. In *Proceedings of POPL '09*, pages 392–403. ACM.

[18] Brewer, E. A. (2015). Kubernetes and the path to cloud native. In Ghandeharizadeh, S., Barahmand, S., Balazinska, M., and Freedman, M. J., editors, *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, page 167. ACM.

[19] Channel types, Go language specification (2016). Channel types, the Go programming language specification. `https://golang.org/ref/spec#Channel_types`.

[20] Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. (2002). Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) Berlin, Germany*, pages 258–269. ACM.

[21] Collier, W. W. (1992). *Reasoning about Parallel Architectures*. Prentice Hall, international edition.

[22] Conway, N., Marczak, W. R., Alvaro, P., Hellerstein, J. M., and Maier, D. (2012). Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM.

[23] Cook, B. (2018). Formal reasoning about the security of amazon web services. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 38–47. Springer.

[24] Cypher, R. and Leu, E. (1995). Efficient race detection for message-passing programs with non-blocking sends and receives. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 534–541. IEEE.

[25] Damodaran-Kamal, S. K. and Francioni, J. M. (1993). Nondeterminancy: testing and debugging in message passing parallel programs. *ACM SIGPLAN Notices*, 28(12):118–128.

[26] Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., and Vitek, J. (2013). Plan B: A buffered memory model for Java. In *Proceedings of POPL '13*, pages 329–342. ACM.

[27] Dijkstra, E. W. (n.d.). Over de sequentialiteit van procesbeschrijvingen. Circulated privately.

[28] Donovan, A. A. A. and Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.

[29] Fava, D. (2017). Operational semantics of a weak memory model with channel synchronization. `https://github.com/dfava/mmgo`.

[30] Fava, D. (2020a). Finding and fixing a mismatch between the Go memory model and data-race detector. In *18th International Conference on Software Engineering and Formal Methods (SEFM)*.

[31] Fava, D. (2020b). Grace: a race detector based on happens-before sets. `https://github.com/dfava/grace`.

[32] Fava, D., Steffen, M., and Stolz, V. (2018a). Anything goes unless forbidden. Notes on synchronization and the operational semantics of a relaxed memory model. In *35th Annual Meeting of the GI Working Group "Programming Languages and Computing Concepts*, pages 96–110.

[33] Fava, D., Steffen, M., and Stolz, V. (2018b). Operational semantics of a weak memory model with channel synchronization. In Havelund, K., Peleska, J., Roscoe, B., and de Vink, E., editors, *International Symposium on Formal Methods*, volume 10951 of *Lecture Notes in Computer Science*, pages 1–19. Springer International Publishing.

[34] Fava, D., Steffen, M., and Stolz, V. (2019). Operational semantics of a weak memory model with channel synchronization. *Journal of Logical and Algebraic Methods in Programming*, 103:1 – 30. An extended version of the FM'18 publication with the same title.

[35] Fava, D. S. and Steffen, M. (2020). Ready, set, Go! Data-race detection and the Go language. *Science of Computer Programming*, 195:102473.

[36] Flanagan, C. and Freund, S. N. (2009). FastTrack: Efficient and precise dynamic race detection. In Hind, M. and Diwan, A., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM.

[37] Flanagan, C. and Freund, S. N. (2010). Adversarial memory for detecting destructive races. In Zorn, B. and Aiken, A., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 244–254. ACM.

[38] Go developer survey (2019). Go developer survey 2019 results. https://blog.golang.org/survey2019-results.

[39] Go language specification (2016). The Go programming language specification. https://golang.org/ref/spec.

[40] Go memory model (2014). The Go memory model. https://golang.org/ref/mem. Version of May 31, 2014, covering Go version 1.9.1.

[41] Go share memory by communicating (2010). The Go blog. https://blog.go-lang.org/codelab-share.

[42] golang.race.detector (2013). https://blog.golang.org/race-detector.

[43] google.sanitizer (2014). https://github.com/google/sanitizers.

[44] google.thread.sanitizer (2015). https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm.

[45] Guerraoui, R., Henzinger, T. A., and Singh, V. (2009). Software transactional memory on relaxed memory models. In Bouajjani, A. and Maler, O., editors, *Proceedings of CAV '09*, volume 5643 of *Lecture Notes in Computer Science*, pages 321–336. Springer Verlag.

[46] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

[47] Huang, J., Meredith, P. O., and Rosu, G. (2014). Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 337–348. ACM.

[48] Inc, S. I. and Weaver, D. L. (1994). *The SPARC architecture manual*. Prentice-Hall.

[49] Jagadeesan, R., Pitcher, C., and Riely, J. (2010). Generative operational semantics for relaxed memory models. In Gordon, A. D., editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 307–326. Springer Verlag.

[50] Jones, G. and Goldsmith, M. (1988). *Programming in occam2*. Prentice-Hall International, Hemel Hampstead.

[51] K framework (2017). The K framework. available at http://www.kframework.org/.

[52] Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017). A promising semantics for relaxed-memory concurrency. In Castagna, G. and Gordon, A. D., editors, *Proceedings of POPL '17*, pages 175–189. ACM.

[53] Katz, S. and Peled, D. (1992). Defining conditional independence using collapses. *Theoretical Computer Science*, 101.

[54] Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). Cakeml: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192.

[55] Kuper, L. and Newton, R. R. (2013). Lvars: lattice-based data structures for deterministic parallelism. In Grelck, C., Henglein, F., Acar, U. A., and Berthold, J., editors, *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*, pages 71–84. ACM.

[56] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[57] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

[58] Lange, J., Ng, N., Toninho, B., and Yoshida, N. (2017). Fencing off Go: Liveness and safety for channel-based programming. In Castagna, G. and Gordon, A. D., editors, *Proceedings of POPL '17*, pages 748–761. ACM.

[59] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

[60] Lidbury, C. and Donaldson, A. F. (2017). Dynamic race detection for C++11. In Castagna, G. and Gordon, A. D., editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457. ACM.

[61] llvm.thread.sanitizer (2011). `https://clang.llvm.org/docs/ThreadSanitizer.html`.

[62] Lochbihler, A. (2013). Making the Java memory model safe. *ACM Transactions on Programming Languages and Systems*, 35(4):12:1–12:65.

[63] Maarand, H. (2020). *Operational Semantics of Weak Sequential Composition*. PhD thesis, Tallinn University of Technology.

[64] Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory model. In *Proceedings of POPL '05*, pages 378–391. ACM.

[65] Maranget, L., Sarkar, S., and Sewell, P. (2012). A tutorial introduction to the ARM and POWER relaxed memory models (version 120).

[66] Marino, D., Musuvathi, M., and Narayanasamy, S. (2009). Literace: effective sampling for lightweight data-race detection. In *ACM Sigplan notices*, pages 134–143.

[67] Mattern, F. (1988). Virtual time and global states in distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Algorithms*, pages 215–226.

[68] Mazurkiewicz, A. (1987). Trace theory. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, (Advances in Petri Nets 1986) Part II*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag.

[69] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[70] Milner, R. (1971). An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann.

[71] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77.

[72] Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM.

[73] Nestmann, U. and Steffen, M. (1997). Typing confluence. In *Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, pages 77–101.

[74] Netzer, R. H. B. and Miller, B. P. (1990). On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 2: Software.*, pages 93–97.

[75] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73.

[76] O'Callahan, R. and Choi, J.-D. (2003). Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178.

[77] Palamidessi, C. (1997). Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM.

[78] Peters, K. and Nestmann, U. (2012). Is it a "good" encoding of mixed choice? In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '12)*, volume 7213 of *Lecture Notes in Computer Science*, pages 210–224. Springer Verlag.

[79] Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency-semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of POPL '16*, pages 622–633. ACM.

[80] Pozniansky, E. and Schuster, A. (2003). Efficient on-the-fly data race detection in multi-threaded C++ programs. In *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*.

[81] Pratikakis, P., Foster, J. S., and Hicks, M. W. (2006). LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 320–331. ACM.

[82] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. (2009). A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 395–403. IEEE.

[83] Pugh, W. (1999). Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*, pages 89–98.

[84] Pugh, W. (2000). The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455.

[85] Rau, B. R. and Fisher, J. A. (1993). Instruction-level parallel processing: history, overview, and perspective. In *Instruction-Level Parallelism*, pages 9–50. Springer.

[86] Rhodes, D., Flanagan, C., and Freund, S. N. (2017). Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 141–156.

[87] Roşu, G. and Şerbănuţă, T. F. (2010). An overview of the K semantic framework. *Journal of Logic and Algebraic Methods in Programming*, 79(6):397–434.

[88] Rupp, K., Horovitz, M., Labonte, F., Shacham, O., Olukotun, K., Hammond, L., and Batten, C. (42). Years of microprocessor trend data. *by karlrupp. net.[Online*.

[89] Sabry, A. and Felleisen, M. (1992). Reasoning about programs in continuation-passing style. In Clinger, W., editor, *Conference on Lisp and Functional Programming (San Francisco, California)*, pages 288–298. ACM.

[90] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

[91] Serebryany, K. and Iskhodzhanov, T. (2009). Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM.

[92] Serebryany, K., Potapenko, A., Iskhodzhanov, T., and Vyukov, D. (2011). Dynamic race detection with llvm compiler. In *International Conference on Runtime Verification*, pages 110–114. Springer.

[93] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., and Flanagan, C. (2012). Sound predictive race detection in polynomial time. In *Proceedings of POPL '12*, pages 387–400. ACM.

[94] Steffen, M. (2016). A small-step semantics of a concurrent calculus with goroutines and deferred functions. In Ábrahám, E., Bonsangue, M., and Johnsen, E. B., editors, *Theory and Practice of Formal Methods. Essays Dedicated to Frank de Boer on the Occasion of his 60th Birthday (Festschrift)*, volume 9660 of *Lecture Notes in Computer Science*, pages 393–406. Springer Verlag.

[95] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210.

[96] Tendler, J. M., Dodson, J. S., Jr., J. S. F., Le, H. Q., and Sinharoy, B. (2002). Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25.

[97] Terauchi, T. and Aiken, A. (2008). A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):27.

[98] Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33.

[99] Valle, S. (2016). Shared variables in Go. A semantic analysis of the Go memory model. Master's thesis, Faculty of Mathematics and Natural Sciences, University of Oslo.

[100] Voung, J. W., Jhala, R., and Lerner, S. (2007). RELAY: Static race detection on millios of lines of code. In *Proceedings oof the 6th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214.

[101] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). *A note on distributed computing*. Springer.

[102] Wikipedia contributors (2020). Larrabee (microarchitecture) — Wikipedia, the free encyclopedia. [Online; accessed 10-November-2020].

[103] Wikipedia contributors (2021). Eating your own dog food — Wikipedia, the free encyclopedia. [Online; accessed 18-January-2021].

[104] Zhang, Y. and Feng, X. (2016). An operational happens-before memory model. *Frontiers in Computer Science*, 10(1):54–81.