Operational Semantics of a Weak Memory Model with Channel Synchronization

Daniel S. Fava[a], Martin Steffen[a], Volker Stolz[a,b]

[a]*Dept. of Informatics, University of Oslo*
[b]*Western Norway University of Applied Sciences*

Abstract

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 *SC-DRF 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 SC-DRF 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.

Keywords: operational semantics, weak memory model, data-race freedom guarantee, channel communication

---

Email addresses: `danielsf@ifi.uio.no` (Daniel S. Fava), `msteffen@ifi.uio.no` (Martin Steffen), `stolz@ifi.uio.no` (Volker Stolz)

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 [30]. 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 [7]. 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 [7]. *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 [11]. In light of these difficulties and despite many attempts, there are no well-accepted comprehensive specification of the C++11 [9, 10] and Java memory models [6, 33, 40].

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, 33]. This is known as the *SC-DRF* guarantee. SC-DRF 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 [12], we favor an operational semantics because it allows us to prove the

SC-DRF 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 SC-DRF 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, however, is described, albeit succinctly and precisely, in prose [20]. 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 [26, 41] and make the source code publicly available [16].

This paper contains additional material compared to the 15 pages of the conference version [17]. In particular:

- We fill in proofs and additional lemmas omitted from the short version. Specifically, we provide details of an auxiliary semantics augmented with read events, needed for an inductive proof of the SC-DRF guarantee.
- 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.
- We address limitations of the model and give directions for further research.

The remaining of the paper is organized as follows. Section 2 presents background information directly related to the formalization of our memory model. Sections 3 and 5 provide the syntax and the semantics of the calculus with relaxed memory and channel communication. Section 6 establishes the SC-DRF 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 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 8 illustrates the behavior of the proposed memory model on litmus tests. Section 9 addresses the model's limitations. Sections 10 and 11 conclude with related and future work.

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 [19, 15] 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 [22] (resp. the π-calculus [36]) or Occam [25]. 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 [20].

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 relaxed read events prevents a class of undesirable behavior called *out-of-thin-air* [11]. 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 SC-DRF 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 [11] 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 [39], 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. [7] 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 [7]. In contrast, Go's memory model is rather "laid back." Its specification [20] 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 [33, 40], C++11's [9, 10], and many other memory models, ours centers around the definition of a *happens-before* relation. The concept dates back to 1978 [29] 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

[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"

does not mean that the effect of the two writes become observable in the given order. For 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:

$$r \rightarrow_{\mathsf{hb}} w \qquad \text{or} \tag{1}$$

$$w \rightarrow_{\mathsf{hb}} w' \rightarrow_{\mathsf{hb}} r \qquad \text{for some other write } w' \text{ to the same variable.} \tag{2}$$

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 (1), and writes w that have been supplanted or *shadowed* by a more recent write w' as detailed in condition (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 locations.

[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 (1)
Failed sync. [20]

```
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)
Channel sync. [20]

```
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 (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 1: Synchronization via channel communication

memory model [20], we have the following constraints related to a channel c with capacity k:

A send on c happens-before the corresponding receive from c completes. (3)

The i^{th} receive from c happens-before the $(i+k)^{th}$ send on c. (4)

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

Example 2.1 (Synchronization via channel communication) *Listing 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`*, 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. 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 (1) or (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). 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 (3), channels implicitly communicate the happens-before relation from the sender to the receiver. Then, with condition (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 (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 (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 2.2 (Synchronization via channel capacity) *Listing 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 (3) and (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 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 (3) travels with data through channels.

Channel capacity: *backward channels* are used to account for condition (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 [13].

3. Abstract syntax

The abstract syntax of the calculus is given in Figure 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 [43] 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 [19]. 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 [42] 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" [37, 38] 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 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 let $r = e$ 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 3.

4. Strong operational semantics

Before introducing the main contribution of the paper, 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 5. Secondly, the strong semantics will later be used in the SC-DRF proof of Section 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 4.2 and 4.3.

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

Figure 2: Abstract syntax

$$
\begin{array}{rcll}
e;\, t & ::= & \texttt{let}\, r = e\, \texttt{in}\, t & \text{when } r \notin fn(t) \\
\texttt{stop} & ::= & \Sigma_0 &
\end{array}
$$

Figure 3: Syntactic sugar

4.1. Configurations

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

$$S ::= p\langle t \rangle \mid (\!| z{:=}v |\!) \mid \bullet \mid S \parallel S \mid c[q] \mid \nu n\, P\,. \tag{5}$$

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 [36].

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, a initial configuration takes the following form.

Definition 4.1 (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.

4.2. Local steps

The reduction steps are given modulo structural congruence $\equiv$ on configurations. The congruence rules are standard and given in Figure 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 [36].

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

Figure 4: Structural congruence

Reduction modulo congruence and other "structural" rules are given in Figure 5. There are two basic reduction steps $\rightsquigarrow$ and $\rightarrow$. Local steps $\rightsquigarrow$ reduce a thread t without touching shared variables; see Figure 6. Global steps are given in the next section.

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

Figure 5: Congruence and reduction

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

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

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

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

Figure 6: Operational semantics: Local steps

4.3. Global steps

To differentiate the strong global steps introduced here from the weak global ones introduce in Section 5, we use a subscript s in the strong semantics rules. For example R-WRITE$_s$ versus R-WRITE.

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 7. Since

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

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

Figure 7: Strong operational semantics: read and write steps

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.

Definition 4.2 (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.*

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.

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

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

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

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

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

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

Figure 8: Strong operational semantics: channel communication

In our semantics (see Fig. 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 4.3 (Invariant for channel queues) *The following global invariant holds for a channel c created with capacity k:*

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

In the case of asynchronous 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.

4.3.3. Thread creation and select statement

The thread creation rule, presented in Figure 9, is unsurprising: the newly spawned thread executes the code t' passed to the corresponding go statement.

$$\frac{\mathit{fresh}(p_2)}{p_1\langle \mathrm{go}\, t';t\rangle \ \rightarrow\ \nu p_2\ (p_1\langle t\rangle \parallel p_2\langle t'\rangle)}\ \text{R-GO}_s$$

Figure 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 5.3.4.

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 translated to the syntax of the paper (see Listing 4).

Listing 4: Channel synchronization example using the syntax of Section 3

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

Figure 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

[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.

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: $\texttt{let}\ \_ = \leftarrow c$. 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 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$_s$).

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

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

Figure 10: Reduction of a simple program according to the strong operational semantics.

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 5.2 and 5.3. Besides processes (or goroutines) running concurrently, the configuration will contain "asynchronous writes" to shared variables.

5.1. Local states, events, and configurations

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

$$P ::= p\langle\sigma,t\rangle \mid m(\!|z{:=}v|\!) \mid c[q] \mid \bullet \mid P \parallel P \mid \nu n\, P \tag{6}$$

Similar to the strong semantics of Section 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 5.3.

Definition 5.1 (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 5.2 (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.

5.2. Local steps

Structural congruence $\equiv$ and the local transition steps $\rightsquigarrow$ defined in Section 4.2 are carried unchanged from the strong to the weak semantics (cf. Figures 4, 5, and 6). The only addition is rule R-LOCAL, which "lifts" the local reduction relation to the global level of configurations.

$$\frac{t_1 \rightsquigarrow t_2}{\langle\sigma, t_1\rangle \rightarrow \langle\sigma, t_2\rangle}\ \textsc{R-Local}$$

5.3. Global steps

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

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 24, 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 24, and will be addressed further in the Discussion section.

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

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

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

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

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

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

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

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

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

Figure 11: Operational semantics: Global steps

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.

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 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 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 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 5.3 (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 $(Val \times \Sigma) + (\{\bot\} \times \Sigma)$, 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 $\bot$ 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.

5.3.3. Select statement

Rules dealing with the select statement in the weak semantics are given on Figure 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.

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).

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 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 4.

The first thing to notice from the run depicted in Figure 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

$$\frac{g_i = c \leftarrow v \qquad \neg closed(c_f[q_f]) \qquad \sigma' = \sigma + \sigma''}{\begin{array}{l} c_b[q_b :: (\sigma'')] \parallel \quad p\langle \sigma, \sum_i \texttt{let } r_i = g_i \texttt{ in } t_i\rangle \quad \parallel c_f[q_f] \;\; \rightarrow \\ \qquad c_b[q_b] \parallel \qquad p\langle \sigma', t_i[()/r_i]\rangle \qquad \parallel c_f[(v,\sigma)) :: q_f] \end{array}} \text{ R-SEL-SEND}$$

$$\frac{g_i = \leftarrow c \qquad q_f = q'_f :: (v, \sigma'') \qquad v \neq \bot \qquad q'_b = (\sigma) :: q_b \qquad \sigma' = \sigma + \sigma''}{\begin{array}{l} c_b[q_b] \parallel \quad p\langle \sigma, \sum_i \texttt{let } r_i = g_i \texttt{ in } t_i\rangle \quad \parallel c_f[q_f] \;\; \rightarrow \\ c_b[q'_b] \parallel \quad p\langle \sigma', \texttt{let } r_i = v \texttt{ in } t_i\rangle \quad \parallel c_f[q'_f] \end{array}} \text{ R-SEL-REC}$$

$$\frac{g_i = c \leftarrow v \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{l} p_1\langle \sigma_1, \sum_i r_i = g_i \texttt{ in } t_i\rangle \parallel \quad p_2\langle \sigma_2, \texttt{let } r = \leftarrow c \texttt{ in } t_2\rangle \;\; \rightarrow \\ \qquad p_1\langle \sigma', t_i[()/r_i]\rangle \parallel \quad p_2\langle \sigma', \texttt{let } r = v \texttt{ in } t_2\rangle \end{array}} \text{ R-SEL-SYNC}_1$$

$$\frac{g_i = \leftarrow c \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{l} p_1\langle \sigma_1, c \leftarrow v; t_1\rangle \parallel \quad p_2\langle \sigma_2, \sum_i \texttt{let } r_i = g_i \texttt{ in } t_i\rangle \;\; \rightarrow \\ \qquad p_1\langle \sigma', t_1\rangle \parallel \quad p_2\langle \sigma', \texttt{let } r_i = v \texttt{ in } t_i\rangle \end{array}} \text{ R-SEL-SYNC}_2$$

$$\frac{g_i = c \leftarrow v \qquad g_j = \leftarrow c \qquad \sigma' = \sigma_1 + \sigma_2 \qquad c_b[] \qquad c_f[]}{\begin{array}{l} p_1\langle \sigma_1, \sum_i \texttt{let } r_i = g_i \texttt{ in } t_i\rangle \parallel \quad p_2\langle \sigma_2, \sum_j \texttt{let } r_j = g_j \texttt{ in } t_j\rangle \;\; \rightarrow \\ \qquad\qquad p_1\langle \sigma', t_i[()/r_i]\rangle \parallel \quad p_2\langle \sigma', \texttt{let } r_j = v \texttt{ in } t_j\rangle \end{array}} \text{ R-SEL-SYNC}_3$$

$$\frac{g_i = \leftarrow c \qquad c_f[(\bot, \sigma'')] \qquad \sigma' = \sigma + \sigma''}{p\langle \sigma, \sum_i \texttt{let } r_i = g_i \texttt{ in } t_i\rangle \;\; \rightarrow \;\; p\langle \sigma', \texttt{let } r_i = \bot \texttt{ in } t_i\rangle} \text{ R-SEL-REC}_\bot$$

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

Figure 12: Operational semantics: Select statement

through channels. Also take note of the additional structure σ, which is sued 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

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

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

Figure 13: Reduction of a simple program according to the weak operational semantics.

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 $\sigma_\perp, \sigma_\perp$ where $\sigma_\perp = (\emptyset, \emptyset)$. These local states represent the fact 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 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 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.

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 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 6.2. Additional intermediate lemmas are relegated to the appendix, in particular Appendix B.

Let us recall the definition of simulation [35] 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 6.1 (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 $s_1 \mathcal{R} t_1$ implies $t_1 \xrightarrow{\alpha} t_2$ for some state t_2. Diagrammatically:*

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

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 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 $\xrightarrow{\tau}^*\xrightarrow{a}\xrightarrow{\tau}^*$.

6.1. The weak semantics simulates the strong

Lemma 6.2 (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.

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

[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.

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 SC-DRF 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.

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 B.2.1.

Definition 6.3 (Observable and concurrent writes) *Let W_P stand for the set of all write events $m\llparenthesis z{:=}v \rrparenthesis$ in a weak configuration P and let $W_P(z)$ stand for the set of identifiers of writes events to the variable z:*

$$W_P(z) = \{m \mid m\llparenthesis z{:=}v \rrparenthesis \in W_P\}\ . \tag{7}$$

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 follows:*

$$\begin{aligned} W_P^{\mathsf{hb}}(z@p) &= E_{hb}(z@p) & (8)\\ W_P^{|||}(z@p) &= W_P(z) \setminus E_{hb}(z@p) & (9)\\ W_P^{\circ}(z@p) &= W_P(z) \setminus E_s(z@p)\ . & (10) \end{aligned}$$

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

Lemma 6.4 (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 $W^{\circ}_{P}(z@p)$ 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 6.5 (Consensus possible) *Weak configurations obey the following invariant*

$$\textstyle\bigcap_{p\in P} W^{\circ}_{P}(z@p) \neq \emptyset \,. \tag{11}$$

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 6.6 (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 6.7 (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*

$$\textstyle\bigcap_{p_i} W^{\circ}_{P}(z@p_i) = \{m\} \,, \tag{12}$$

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

Corollary 6.8 (Locally deterministic read) *Assume $P_0 \to_w^* P$ 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 6.9 (Race-free consensus) *Weak configurations for race-free programs obey the following invariant*

$$\bigcap_{p_i \in P} W_P^\circ(z@p_i) = \{m\} \tag{13}$$

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

Definition 6.10 (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 (13) from Lemma 6.9 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 6.11 (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 on Table 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.*

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

Lemma 6.13 (Erasure preserves well-formedness) *Let P be a race-free reachable weak configuration. If $\vdash_w P : ok$ then $\vdash_s \lfloor P \rfloor : ok$.*

$$\lfloor \bullet \rfloor^R = \bullet \tag{14}$$

$$\lfloor p\langle \sigma, t\rangle \rfloor^R = \langle t\rangle \tag{15}$$

$$\lfloor m(\!|z{:=}v|\!)\rfloor^R = \begin{cases} \bullet & \text{if } m \in R \\ (\!|z{:=}v|\!) & \text{otherwise} \end{cases} \tag{16}$$

$$\lfloor P_1 \parallel P_2 \rfloor^R = \lfloor P_1 \rfloor^R \parallel \lfloor P_2 \rfloor^R \tag{17}$$

$$\lfloor \nu n\, P \rfloor^R = \begin{cases} \lfloor P \rfloor^R & \text{if } \forall p \in P.\ n \in W_P^{\circ}(\_@p) \\ \lfloor P \rfloor^{R \cup \{n\}} & \text{otherwise} \end{cases} \tag{18}$$

$$\lfloor c[q_1, q_2] \rfloor^R = c[\lfloor q_1 \rfloor^R, \lfloor q_2 \rfloor^R] \tag{19}$$

Table 1: Definition of the erasure function $\lfloor P \rfloor^R$

Theorem 6.14 (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

$$P \,\mathcal{R}\, S \quad \text{if} \quad S \equiv \lfloor P \rfloor \tag{20}$$

using the erasure from Definition 6.11. Note that by Lemma 6.12, $P_1 \,\mathcal{R}\, S$ and $P_1 \equiv P_2$ implies $P_2 \,\mathcal{R}\, S$.

Case: R-WRITE$_w$: $p\langle \sigma, z := v; t\rangle \to_w \nu m\, (p\langle \sigma', t\rangle \parallel m(\!|z{:=}v|\!))$,
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 6.6(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 6.9, we have that this is the only event that is observable by all threads, i.e.

$$\bigcap_{p_i} W^{\circ}_{P'}(z@p_i) = \{m\} \ . \tag{21}$$

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 (21) shows that the write event m is not "filtered out" (cf. the cases of equation (16) and (18) in Definition 6.11) 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. □

7. Implementation

We have implemented the strong and the weak semantics in $\mathbb{K}$, a rewrite-based executable semantics framework [26, 41]. 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 [16]. 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 (ie. 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 11. The code, given on Figure 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 queue (line 12). The receiving goroutine's local state is updated. In specific, its

[6]The latter is indirectly clear already as we have established that $\lfloor\rfloor$ preserves well-formedness under the assumption of race-freedom (Lemma 6.13).

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 14: Snippet from the implementation of the channel receive rule in 𝕂

A byproduct of the implementation is the ability to execute programs and observe their output. At the start of execution, the runtime configuration has the format shown on Figure 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 on Section 2, where a simple setup function is called asynchronously from main. The example, rewritten in the proposed syntax, is shown on Listing 4. Coordination is achieved through a shared channel. A message indicates that the

```
<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 15: The initial runtime configuration

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.

Figure 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."

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 interleav-

[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 16: Sample output from running Listing 4 on the weak semantics

ing) gives rise to *candidate executions*. Candidate executions are graphs that help define and illustrate behavior accepted or rejected by the semantics; see [8] 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.

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 17 is the well-known litmus test for *message pass-*

ing (mp) and, on the right, a corresponding candidate execution. The code snippet

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

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

(a) Litmus test

$$\begin{tikzcd}
p_0 & & p_1 \\
n_1{:}\mathsf{W}x = 1 \arrow[d, "ppo"] & & n_3{:}\mathsf{R}y = 1 \arrow[d, "ppo"] \\
n_2{:}\mathsf{W}y = 1 \arrow[rru, "rf"] & & n_4{:}\mathsf{R}x = 0 \arrow[llu, "fr"']
\end{tikzcd}$$

(b) Candidate execution

Figure 17: mp (message passing)

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 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 $n_0 \rightarrow_{\mathsf{co}} n_1$ 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 form 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 4 In weaker settings, the $\rightarrow_{\mathsf{ppo}}$-edges may be replaced by $\rightarrow_{\mathsf{po}}$-edges. For example, both our model and PSOstyle memory models with per-location write buffers allow the observation $r_1 = 1$ and $r_2 = 0$ in the mp litmus test of Figure 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 18.

p_0	p_2
`r1 := x;`	`x := 1;`
`r2 := x;`	

$r_1 = 1, r_2 = 0$

(a) Litmus test

$$\begin{tikzcd}
p_0 & & p_1 \\
n_1{:}\mathsf{R}x = 1 \arrow[d, "po"] & & n_3{:}\mathsf{W}x = 1 \arrow[ll, "rf"'] \\
n_2{:}\mathsf{R}x = 0 \arrow[rru, "fr"'] & &
\end{tikzcd}$$

(b) Candidate execution

Figure 18: coRR

The fact that repeated reads by the same thread give different seemingly incoherent 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 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 [23] and pre-Power4 machines [44]. 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 19. The candidate execution graph

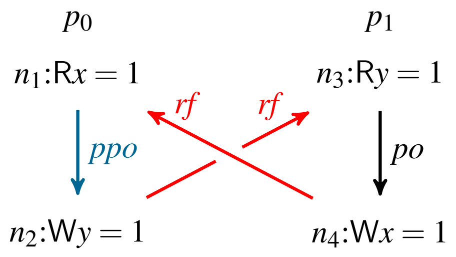
[8] In general, coherence tests coXY involve an access of kind X and an access of kind Y with X and Y standing for either R (read) or W (write).

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.

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

$r_1 = 1, r_2 = 1$

(a) Litmus test



(b) Candidate execution

Figure 19: Load buffering (lb)

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 19a), but also a control flow dependency. Control flow dependency 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 20. It closely resembles the previous case from Figure 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 19b.

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

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

(a) Litmus test

$$\begin{tikzcd}
p_0 & p_1 \\
n_1{:}\mathsf{R}x = 1 \arrow[d, "ppo"] & n_3{:}\mathsf{R}y = 1 \arrow[d, "ppo"] \\
n_2{:}\mathsf{W}y = 1 \arrow[ur, "rf"] & n_4{:}\mathsf{W}x = 1 \arrow[ul, "rf"']
\end{tikzcd}$$

(b) Candidate execution

Figure 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 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 [20], 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 20, with $\rightarrow_{\mathsf{po}}$ instead of $\rightarrow_{\mathsf{ppo}}$-edges, the out-of-thin-air observation is perfectly acceptable.[9]

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

Finally, we go back to the message passing pattern from Figure 17 to illustrate the role of channel communication. The assured ordering of the reads, resp. writes, represented by $\rightarrow_{\mathsf{ppo}}$-edges in Figure 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 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 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.

$$\begin{tikzcd}
p_0 & p_1 \\
n_1{:}\mathsf{W}x = 1 \arrow[d, "ff"] & n_3{:}\mathsf{R}y = 1 \arrow[d, "ff"] \\
n_2{:}\mathsf{W}y = 1 \arrow[ur, "rf"] & n_4{:}\mathsf{R}x = 0 \arrow[ul, "fr"']
\end{tikzcd}$$

(a) mp+ffences

$$\begin{tikzcd}
p_0 & p_1 \\
n_1{:}\mathsf{W}x = 1 \arrow[d, "hb"] & n_3'{:} \leftarrow c \arrow[d, "hb"] \\
n_2'{:}c \leftarrow 0 \arrow[ur, "hb"] & n_4{:}\mathsf{R}x = 0 \arrow[ul, "fr"']
\end{tikzcd}$$

(b) mp+chan

Figure 21: mp with synchronization

In Figure 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, 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.

with the specification.

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

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 5.4 and discussed in Section 8. This is in line with 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 [28], 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.

10. Related work

There are numerous proposals for and investigations of weak and relaxed memory models [1, 34, 3]. One widely followed approach, called *axiomatic*, specifies allowed behavior by defining various ordering relations on memory accesses

[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.

and synchronizing events. Go's memory model [20] gives an informal impression of that style of specification. Less frequent are *operational* formalizations.

Boudol and Petri [12] 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 [45] 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. [27]. 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 [39] 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. [21] 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 [18], 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 [32], 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. [14] 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. [24] 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, [24] 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. [31] 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.

11. Conclusion

This paper 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 [16]. 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 SC-DRF 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. We are working on further relaxing 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.

Acknowledgments. We thank Olaf Owe for his comments on draft versions of the paper and anonymous reviewers for their valuable input.

Bibliography

[1] S. V. Adve, K. Gharachorloo, Shared Memory Consistency Models: A Tutorial, Research Report 95/7, Digital WRL, 1995.

[2] S. V. Adve, M. D. Hill, Weak Ordering — A New Definition, SIGARCH Computer Architecture News 18 (3a) (1990) 2–14.

[3] J. Alglave, L. Maranget, M. Tautschnig, Herding Cats: Modelling, Simulation, Testing, and Data-Mining for Weak Memory, ACM Transactions on Programming Languages and Systems 36 (2).

[4] Y. A. Alrahman, M. Andric, A. Beggiato, A. Lluch-Lafuente, Can We Efficiently Check Concurrent Programs Under Relaxed Memory Models in Maude?, in: S. Escobar (Ed.), Rewriting Logic and Its Applications – 10th International Workshop, WRLA 2014. Revised Selected Papers, vol. 8663 of *Lecture Notes in Computer Science*, Springer Verlag, ISBN 978-3-319-12903-7, 21–41, 2014.

[5] G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

[6] D. Aspinall, J. Ševčík, Java memory model examples: Good, bad and ugly, Proc. of VAMP 7.

[7] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, P. Sewell, The Problem of Programming Language Concurrency Semantics, in: J. Vitek (Ed.), Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, vol. 9032 of *Lecture Notes in Computer Science*, Springer Verlag, 283–307, 2015.

[8] M. Batty, S. Owens, S. Sarkar, T. Weber, Mathematizing C++ Concurrency, in: Proceedings of POPL '11, ACM, 55–66, 2011.

[9] Becker, Programming Languages — C++, ISO/IEC 14882:2001, 2011.

[10] H.-J. Boehm, S. V. Adve, Foundations of the C++ Concurrency Memory Model, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 68–78, 2008.

[11] H.-J. Boehm, B. Demsky, Outlawing Ghosts: Avoiding Out-of-thin-air Results, in: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC '14, ACM, New York, NY, USA, ISBN 978-1-4503-2917-0, 7:1–7:6, doi:10.1145/2618128.2618134, 2014.

[12] G. Boudol, G. Petri, Relaxed Memory Models: An Operational Approach, in: Proceedings of POPL '09, ACM, 392–403, 2009.

[13] W. W. Collier, Reasoning about Parallel Architectures, Prentice Hall, international edn., 1992.

[14] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, J. Vitek, Plan B: A Buffered Memory Model for Java, in: Proceedings of POPL '13, ACM, 329–342, 2013.

[15] A. A. A. Donovan, B. W. Kernighan, The Go Programming Language, Addison-Wesley, 2015.

[16] D. Fava, Operational Semantics of a Weak Memory Model with Channel Synchronization, URL `https://github.com/dfava/mmgo`, 2017.

[17] D. Fava, M. Steffen, V. Stolz, Operational Semantics of a Weak Memory Model with Channel Synchronization, in: K. Havelund, J. Peleska, B. Roscoe, E. de Vink (Eds.), FM, vol. 10951 of *Lecture Notes in Computer Science*, Springer Verlag, 1–19, 2018.

[18] C. Flanagan, S. N. Freund, Adversarial Memory for Detecting Destructive Races, in: B. Zorn, A. Aiken (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 244–254, 2010.

[19] Go language specification, The Go Programming Language Specification, `https://golang.org/ref/spec`, 2016.

[20] Go memory model, The Go Memory Model, `https://golang.org/ref/mem`, version of May 31, 2014, covering Go version 1.9.1, 2014.

[21] R. Guerraoui, T. A. Henzinger, V. Singh, Software Transactional Memory on Relaxed Memory Models, in: A. Bouajjani, O. Maler (Eds.), Proceedings of CAV '09, vol. 5643 of *Lecture Notes in Computer Science*, Springer Verlag, 321–336, doi:10.1007/978-3-642-02658-4_26, 2009.

[22] C. A. R. Hoare, Communicating Sequential Processes, Communications of the ACM 21 (8) (1978) 666–677.

[23] S. I. Inc, D. L. Weaver, The SPARC architecture manual, Prentice-Hall, 1994.

[24] R. Jagadeesan, C. Pitcher, J. Riely, Generative Operational Semantics for Relaxed Memory Models, in: A. D. Gordon (Ed.), Programming Languages and Systems, vol. 6012 of *Lecture Notes in Computer Science*, Springer Verlag, 307–326, 2010.

[25] G. Jones, M. Goldsmith, Programming in occam2, Prentice-Hall International, Hemel Hampstead, 1988.

[26] K framework, The K Framework, available at `http://www.kframework.org/`, 2017.

[27] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, D. Dreyer, A promising semantics for relaxed-memory concurrency, in: G. Castagna, A. D. Gordon (Eds.), Proceedings of POPL '17, ACM, 175–189, URL `http://dl.acm.org/citation.cfm?id=3009850`, 2017.

[28] R. Kumar, M. O. Myreen, M. Norrish, S. Owens, 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, 179–192, 2014.

[29] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21 (7) (1978) 558–565.

[30] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, IEEE Transactions on Computers C-28 (9) (1979) 690–691.

[31] J. Lange, N. Ng, B. Toninho, N. Yoshida, Fencing off Go: Liveness and Safety for Channel-Based Programming, in: G. Castagna, A. D. Gordon (Eds.), Proceedings of POPL '17, ACM, 748–761, 2017.

[32] A. Lochbihler, Making the Java Memory Model Safe, ACM Transactions on Programming Languages and Systems 35 (4) (2013) 12:1–12:65.

[33] J. Manson, W. Pugh, S. V. Adve, The Java Memory Model, in: Proceedings of POPL '05, ACM, 378–391, 2005.

[34] L. Maranget, S. Sarkar, P. Sewell, A Tutorial Introduction to the ARM and POWER Relaxed Memory Models (Version 120), 2012.

[35] R. Milner, An Algebraic Definition of Simulation between Programs, in: Proceedings of the Second International Joint Conference on Artificial Intelligence, William Kaufmann, 481–489, 1971.

[36] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, Part I/II, Information and Computation 100 (1992) 1–77.

[37] C. Palamidessi, Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus, in: Proceedings of POPL '97, ACM, 256–265, 1997.

[38] K. Peters, U. Nestmann, Is it a "Good" Encoding of Mixed Choice?, in: Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '12), vol. 7213 of *Lecture Notes in Computer Science*, Springer Verlag, 210–224, 2012.

[39] J. Pichon-Pharabod, P. Sewell, A Concurrency-Semantics for Relaxed Atomics that Permits Optimisation and avoids Thin-Air Executions, in: Proceedings of POPL '16, ACM, 622–633, 2016.

[40] W. Pugh, Fixing the Java Memory Model, in: Proceedings of the ACM Java Grande Conference, 89–98, 1999.

[41] G. Roşu, T. F. Şerbănuţă, An Overview of the K Semantic Framework, Journal of Logic and Algebraic Methods in Programming 79 (6) (2010) 397–434, doi:10.1016/j.jlap.2010.03.012.

[42] A. Sabry, M. Felleisen, Reasoning about programs in continuation-passing style, in: W. Clinger (Ed.), Conference on Lisp and Functional Programming (San Francisco, California), ACM, 288–298, 1992.

[43] M. Steffen, A Small-Step Semantics of a Concurrent Calculus With Goroutines and Deferred Functions, in: E. Ábrahám, M. Huisman, E. B. Johnsen (Eds.), Theory and Practice of Formal Methods. Essays Dedicated to Frank de Boer on the Occasion of his 60th Birthday (Festschrift), vol.

9660 of *Lecture Notes in Computer Science*, Springer Verlag, 393–406, 2016.

[44] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Q. Le, B. Sinharoy, POWER4 system microarchitecture, IBM Journal of Research and Development 46 (1) (2002) 5–25.

[45] Y. Zhang, X. Feng, An Operational Happens-Before Memory Model, Frontiers in Computer Science 10 (1) (2016) 54–81.

A. The weak semantics simulates the strong

PROOF OF LEMMA 6.2 (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: $S\ \mathcal{R}\ P$ 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. □

B. Proofs via a weak semantics augmented with read and write events

This section contains supplementary material and proofs for the lemmas of Section 6. In particular, the material here allows us to carry out the harder direction of the simulation proof of Section 6.2, namely that the strong semantics simulates the weak one for race-free programs.

We start in Section B.1 augmenting the weak semantics with additional information which has no relevance aside from assisting the proofs. Section B.2 covers properties of the augmented semantics.

B.1. Augmenting the weak semantics

This section presents an “alternative” representation of the weak semantics of Section 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 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 (6) on page 18 are therefore adapted to contain events of the following form:

$$m(\!|\sigma, z := v|\!)_p \quad \text{and} \quad m[\![\sigma, ?z]\!]_p\ , \tag{B.1}$$

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 B.1 (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 11, only the read and write steps require adaptation. See Figure B.22 for the augmented rules, which behave exactly as the originals except that the steps now record additional information as part of the configuration.

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (m, z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(m)}{p\langle \sigma, z := v; t\rangle \;\rightarrow\; \nu m\ (p\langle \sigma', t\rangle \parallel m(\!|\sigma, z := v|\!)_p)}\ \text{R-WRITE}_\sigma$$

$$\frac{\sigma = (\_, E_s) \qquad m \notin E_s \qquad \mathit{fresh}(m')}{\begin{array}{l} p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel \ m(\!|\_, z := v|\!)_\_ \quad \rightarrow \\ \nu m'\ (p\langle \sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel \ m(\!|\_, z := v|\!)_\_ \ \parallel m'[\![\sigma, ?z]\!]_p) \end{array}}\ \text{R-READ}_\sigma$$

Figure B.22: 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 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 6.

B.2. Additional concepts and lemmas

In the following, we use $\rightarrow_w$, $\rightarrow_w^*$, 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 6.3 from the main part of the paper.

Definition B.2 (Binary relations on events) *Let e_1 and e_2 be two different events, with E_{hb}^2 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 $m_1 \in E_{hb}^2$.*
2. *e_1 and e_2 are concurrent, written $e_1 \mathrel{|||} e_2$, 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 $e_1 \mathrel{|||} e_2$, 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 B.3, stating that $\rightarrow_{\mathsf{hb}}$ is a partial order.

Lemma B.3 (Simple properties of event relations)

- *$\#$ and $|||$ 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 $e_1 \mathrel{|||} e_2$ implies $p_1 \neq p_2$.*

PROOF. # and ||| 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 6.3. 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 [20].

Definition B.4 (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 6.3). Note, however, that the transition-based definition from Section 6.2 does *not* include condition (2) from Definition B.4. Even if the two definitions differ concerning that condition, they are intuitively capturing the same concept: In the earlier Definition 6.3, 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 B.4 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 29 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 B.5 (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.*

B.2.1. General invariant properties

See also Section 6.2.1 in the main part.

Lemma B.6 (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. *$w \mathbin{|||} r$ 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 $w \mathbin{|||} r$.*
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 B.9. By straightforward induction. □

PROOF OF THE INVARIANTS LEMMA 6.4. A straightforward consequence of the corresponding property for read and write events of the augmented semantics from Lemma B.6. □

PROOF OF LEMMA 6.5 ("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 $W^{\mathsf{o}}_{P_i}(z@p) \neq \emptyset$ 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 $P_i \to P_{i+1}$, we have that $\bigcap_{p \in P_{i+1}} W^{\circ}_{P_{i+1}}(z@p)$. $\square$

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 7), the lemma implies that for race-free programs, there is no ambiguity when observing write events.

Lemma B.7 (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 $w_2 \parallel\mkern-3mu| r$, then $w_2 \# r$ but w_1 is not in conflict with any of the other two events.

B.2.2. Race-free resp. conflict-free reductions

See also Section 6.2.2 in the main part of the paper.

Lemma B.8 (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*

$$|\{w \mid r \leftarrow^z_{\mathsf{o}} w\}| \leq 1 \tag{B.2}$$

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 B.7, this implies that P' contains at least two conflicting events. With Definition 7, the existence of conflicting events contradicts the assumption of race-freedom, which concludes the proof. □

Corollary B.9 *Let* P*,* P' *and* z *be given as in Lemma B.8. Then we have*

$$|\{w \mid r \leftarrow^z_{\mathsf{o}} w\}| = 1\ . \tag{B.3}$$

PROOF. A direct consequence of B.8 and of Lemma B.6(3) (or alternatively of Lemma B.6(4)). □

PROOF OF LEMMA 6.6 (NO CONCURRENT WRITE WHEN IT COUNTS). A direct consequence of the equivalence of races and conflicts from Definition 7. 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 7, this contradicts the assumption that P_0 has no r/w race. The case for w/w races is analogous. □

Lemma B.10 (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*

$$W_P^{\circ}(z@p) = \{m\} \ . \tag{B.4}$$

PROOF. For the write step: assume that there are two different observable writes w_1 and w_2. By Lemma 4, $W_P^{|||}(z@p) = \emptyset$. By Definition 6.3, that means all observable writes are in happens-before relation, i.e., $W_P^{\circ}(z@p) = W_P^{\mathsf{hb}}(z@p)$. 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 , that contradicts the assumption of race-freedom. The case for a read-step is analogous (alternatively it follows from Lemma B.8). □

As an easy consequence, we obtain the following consensus lemma:

PROOF OF LEMMA 6.7 ("RACE-FREE CONSENSUS WHEN IT COUNTS"). A direct consequence of unique observability from Lemma B.10 and the possible consensus property from Lemma 6.5. □

PROOF (OF COROLLARY 6.8). A direct consequence of the consensus Lemma 6.7. □

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 6.7).

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 *SC-DRF* 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 6.11 would not be a function, resp. would not yield well-formed strong configurations.

PROOF OF LEMMA 6.9 (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 6.6(1), there are no concurrent writes for p before the step, i.e., $W_P^{|||}(z@p) = \emptyset$. By Definition 6.3, that means all observable writes are in happens-before relation, i.e., $W_P^{\circ}(z@p) = W_P^{\mathsf{hb}}(z@p)$.[12] 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, 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'. $\square$

[12]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.