Operational Semantics of a Weak Memory Model with Channel Synchronization

Proof of Sequential Consistency for Race-Free Programs

Daniel S. Fava, Martin Steffen, and Volker Stolz

Technical report Nr. 477

IFI
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

January 26, 2018

Operational Semantics of a Weak Memory Model with Channel Synchronization

Proof of Sequential Consistency for Race-Free Programs

Daniel Schnetzer Fava,[1] Martin Steffen[1] and Volker Stolz[1,2]

[1] Dept. of Informatics, University of Oslo
[2] Western Norway University of Applied Sciences

Abstract. A multitude of weak memory models exists supporting various types of relaxations and different synchronization primitives. On one hand, they must be lax enough to allow for hardware and compiler optimizations; on the other, the more lax the model, the harder it is to reason about programs. 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.

In this work we present a weak memory model for a calculus inspired by Go. Thus, different from previous approaches, we focus on a memory model with buffered channel communication as the sole synchronization primitive. While memory models are often defined axiomatically where the notion of a program is abstracted away (often as a graph with memory events as nodes), we formalize our model via an operational semantics. This allows us to prove the SC-DRF guarantee using a standard simulation technique that highlights invariants and gives insight into the how the memory works. Finally, we provide a concrete implementation in $\mathbb{K}$, a rewrite-based executable semantic framework, which allows us to derive an interpreter for the proposed language.

Table of Contents

1 Introduction 3
2 Background 4
3 Abstract syntax 7
4 Operational semantics 8
  4.1 Local states, events, and configurations 8
  4.2 Reduction steps 9
5 Relating the strong and the weak semantics 14
  5.1 Preliminaries 14
  5.2 The weak semantics simulates the strong 15
  5.3 The strong semantics conditionally simulates the weak one 15
6 Implementation 20
7 Related work 22
8 Conclusion 24
A Strong semantics 27
B The weak semantics simulates the strong 27
C Proofs via a weak semantics augmented with read and write events 29
  C.1 Augmenting the weak semantics 29
  C.2 Additional concepts and lemmas 30
D Rules for the select statement 36

1 Introduction

A *memory model* dictates which values may be observed when reading from memory, thereby affecting how concurrent processes communicate through shared memory. *Sequential consistency* [26] was one of the first models to be clearly specified. It stipulates that operations must appear to execute one at a time and in program order. Sequential consistency still constitutes a baseline for well-behaved memory, being arguably the simplest or strongest. For a long time already, however, hardware does not guarantee sequential consistency. Likewise, it is agreed upon that this form of consistency is much too strong to serve as the underlying memory semantics of a programming language: it would rob the compiler writer the chance to exploit the underlying HW for efficient parallel execution and would prevent many established compiler optimizations. Less agreement exists, however, on what exactly should a proper memory model offer. Consequently, a bewildering array of weak or relaxed memory models have been proposed, investigated, and implemented. Different taxonomies and catalogues of so-called *litmus tests*, which highlight specific aspects of memory models, have also been researched (cf. e.g. [1]). Not only are weak memory models harder to understand and to program for, they are also hard to formalize. For example, there does not exist an uncontroversial comprehensive specification of the $C^{++}11$ [8, 9] or Java memory models [6, 28, 35]. Certainly, that is not for lack of trying. Despite the lack of consensus, *one generally accepted principle of relaxed memory models has emerged*, namely: no matter how much relaxation one wishes to allow, if the program is *data-race free* or *properly synchronized*, then it at least behaves sequentially consistently [2, 28]. This is known as the *SC-DRF* guarantee.

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, however, is that either too much "good" behavior (i.e., behavior that is deemed desirable) is excluded by the model or some "bad" behavior fails to be filtered out [7]. A common class of undesired behavior that often plagues weak memory specifications is the so called *out-of-thin-air* behavior. These are results that can be justified by the model via circular reasoning but that do not appear in the actual executions of a program [10]. In contrast to an axiomatic approach, we present in this paper an *operational* semantics for a weak memory. Similar to Boudol and Petri, we favor an operational semantics because it allows us to prove the SC-DRF guarantee using a standard simulation [11]. Compared to other memory model semantics in which the notion of a program is abstracted away (often in the form of a graph with nodes as memory events), we think our formalism leads to an easier to understand proof of the SC-DRF guarantee. 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.

The calculus is inspired by the Go programming language, whose memory model is described, albeit succinctly and precisely, in prose [19]. Like C^{++} and Java, Go's memory model builds upon on a so-called "happens-before" relation, which regulates which memory actions are guaranteed to occur before others. The contributions of this paper are:

– There are few studies on channel communication as synchronization primitive for weak memory. We give an operational theory for a weak memory with bounded channel communication by leveraging thread-local happens-before information.
– We prove that the proposed memory upholds the *sequential consistency guarantee for data-race free* programs using a standard conditional simulation proof.
– We implement the operational semantics in the $\mathbb{K}$ executable semantics framework [23, 36]. The implementation forms a basis for experimentation and analysis.

The $\mathbb{K}$ executable semantics is publicly available in the accompanying git-repository [15].

This paper is organized as follows. Section 2 presents background information directly related to the formalization of our memory model. Sections 3 and 4 provide the syntax and the semantics of the calculus with relaxed memory and channel communication. Section 5 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. Sections 7 and 8 conclude with related and future work.

This report contains additional material compared to the 15 pages conference version. Most of the additional material is collected in the appendix. Besides that, the main part of the report contains additional material and explanations left out from the proceedings version for lack of space. The order of presentation in the main part is unchanged otherwise.

2 Background

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

Go's memory model. The Go language [18, 14] recently gained traction in networking applications, web servers, distributed software and the like. It prominently features goroutines (i.e., asynchronous execution of function calls resembling lightweight threads) and buffered channel communication in the tradition of CSP [21] (resp. the π-calculus [31]) or Occam [22]. 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 [19]. 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 (see [19]), i.e., 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 it will become clear in the next sections, our semantics does not, however, relax read events; thus making it stronger than Go's. On the plus side, this prevents a class of undesirable behavior called *out-of-thin-air* [10]. On the negative, the absence of relaxed reads comes at the expense of some forms of compiler optimizations.

Happens-before relation and observability. Like many other memory models, most notably Java's [28, 35] and C^{++}11's [8, 9], our memory model centers around the definition of a *happens-before* relation. The concept dates back to 1978 [25] and was introduced in a pure *message-passing* setting, i.e., without shared variables.[3] The relation is a technical vehicle for defining the semantics of memory models. It is important to note, however, that just because an instruction (resp. a corresponding 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[4], and especially, it 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 or the 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 schedule may be coincidental and does not necessarily imply that they are in happens-before relationship. 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 (also abbreviated as $e_1 \rightarrow_{\mathsf{hb}} e_2$ in this section) as opposed to its natural language interpretation. Similarly for "happens-after." 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 etc. occurs after another.

Languages like Java and C^{++} go to great lengths not only to offer the crucial SC-DRF guarantee for well-synchronized programs, but beyond that, strive to clarify the resulting non-SC behavior when the program is *ill*-synchronized. This involves ruling out definitely unwelcome behavior. Doing this precisely, however, is far from trivial. One class of unwanted behavior that is particularly troublesome is the so called *out-of-thin-air* behavior [10]. Intuitively, this represents behavior that can be justified via some sort of circular reasoning. According to Pichon-Pharabod and Sewell, however, there is not even an exact, generally accepted definition of out-of-thin-air behavior [34]. Doubts have even been cast upon a general style of defining weak memory models. For example, Batty et al. 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 [7]. In this formalization, the distinction between "good" (*i.e.*, expected behavior) and "bad" (*i.e.*, outlawed behavior) is usually given as 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

[3] The relation was called happened-before in the original paper.
[4] Assuming that x and y are not aliases in the sense that they refer to the same or "overlapping" memory locations.

Listing (1.1) Erroneous synchronization

```
var a string
var done bool

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

func main() {
        go setup()
        for !done { } // try waiting
        print(a)
}
```

Listing (1.2) Channel synchronization

```
var a string
var c = make(chan int, 10)

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

func main() {
        go setup()
        <-c     // receive
        print(a)
}
```

Fig. 1: Synchronization via channel communication [19]

unacceptable for others [7]. In contrast, Go's memory model is rather "laid back." Its specification [19] does not even mention "out-of-thin-air" behavior, let alone attempting to define exactly what that should be.

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:

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

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

For the sake of discussion, let's concentrate on the following two constituents for the happens-before relation: 1) *program order* and 2) the order stemming from channel communication.[5] According to the Go memory model [19], we have the following constraints related to a channel c:

$$\text{A send on } c \text{ happens-before the corresponding receive from } c \text{ completes.} \tag{3}$$

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

To illustrate, consider the example on Listing 1.1. The main function spawns an asynchronous execution of `setup`, both afterwards running concurrently. In the thread or goroutine executing `setup`, the write to variable `a` happens-before the write to `done` (being in program order), and 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`; it is possible for main to observe the corresponding initial values instead. This potentially makes the writes to appear out-of-order from the main thread's perspective.

Replacing the use of `done` by channel synchronization remedies the problem and properly synchronizes the two functions (cf. Listing 1.2). Intuitively, the channel synchronization places the reading of the variables after their writing. Technically, as the

[5] 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.

receive happens-after the send, the receiving main thread "receives" also the knowledge that the two write events are ordered happen-before-wise. With condition (2), the initial values of the two variables are no longer observable by the reads in the main thread. This can be interpreted as channels effectively communicating the happens-before relation from the sender to the receiver. Condition (4), due to the boundedness of channels, transmits happens-before information in the backward direction for some receiver to some sender (not shown in the example, though). For *synchronous* channels, where $k = 0$, the two threads participating in the rendezvous 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. 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.

3 Abstract syntax

The abstract syntax of the calculus is given in Table 1. *Values* v include local variables (or registers) r and names (or references) n. References, such as references to channels, are dynamically created and are, therefore, part of the *run-time* syntax. Run-time syntax is highlighted with an underline as $\underline{n}$ in the grammar. Later we use c specifically for references to channels. We do not explicitly list values such as the unit value, booleans, integers, etc. We also omit compound local expressions like $r_1 + r_2$, which would be straightforward to add. Shared variables are denoted by x, z etc, $\texttt{load}\ z$ represents the reading the shared variable z into the thread, and $z := v$ denotes writing to z. The syntax for reading global variables makes the shared memory access explicit in this representation (unlike in the concrete Go surface syntax). Especially, 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.

A new channel is created by $\texttt{make}\ (\texttt{chan}\ T, v)$ where T represents the type of values carried by the channel and the non-negative integer v 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 for an operational semantics dealing with goroutines and

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

Table 1: Abstract syntax

closures in a purely functional setting, i.e., without need of any shared memory model [37]. Note that by being functional, the referenced semantics does not deal with shared memory and memory models.

The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches (called communication clauses by the Go specification [18]). These branches act as guarded threads. In Go, general expressions can serve as guards of a select statement's branch. In our calculus, however, we impose 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 and does not impose any actual reduction in expressivity. Both in Go and in our formalization, there is at most one branch guarded by `default` in each select-statement. It is allowed that a channel is mentioned in more than one guard. Also "mixed choices" [32, 33] are allowed, in that sending and receiving guards can be used in the same select-statement. We use `stop` as syntactic sugar for the empty select statement; it represents a permanently blocked thread. The `stop`-thread is also the only way to syntactically "terminate" a thread, i.e., it is the only element of t without syntactic sub-terms. The `let`-construct $\texttt{let}\ r = e\ \texttt{in}\ t$ combines sequential composition and the use of scopes for local variables r: after evaluating e, the rest t is evaluated where the resulting value of e is handed over using r. The let-construct is seen as a binder for variable r in t. It becomes *sequential composition* when r does not occur free in t. We use semicolon as syntactic sugar in such situations.

4 Operational semantics

In this section we define the operational semantics of the calculus. We fix the runtime configurations of a program before giving the operational rules in Section 4.2. Besides processes (or goroutines) running concurrently, the configuration will contain "asynchronous writes" to shared variables.

4.1 Local states, events, and configurations

Let X represent a set of shared variables such as x, $z\ldots$ and let N represent an infinite set of names or identifiers with typical elements n, $n'_2\ldots$ As mentioned earlier, for readability, we will use names like c, $c_1,\ldots$ for channels, and p, p'_1 …for goroutines (or processes). A run-time configuration is then given by the following syntax:

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

Configurations consist of the parallel composition of goroutines $p\langle\sigma,t\rangle$ and write events $n(\!|z{:=}v|\!)$; $\bullet$ represents the empty configuration. A write event is labeled by a unique identifier n and records the shared variable being written to and the written value. The ν-binder, known from the π-calculus, indicates dynamic scoping [31]. Goroutines or processes $p\langle\sigma,t\rangle$ contain, besides the code t to be executed, a local view detailing the observability of write events from the perspective of p. Local observability is formulated "negatively:" all write events are observable by default and σ keeps track of which events are *non-observable* from a thread's point of view. In order to properly update the list of non-observable events, σ also contains thread-local information about the "happens-before" relationship between write events.

Definition 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's furthermore denote by* $E_{hb}(z)$ *the set* $\{n \mid (n,z) \in E_{hb}\}$. *We write* $\sigma_\perp$ *for the local state* $(\emptyset,\emptyset)$ *containing neither happens-before nor shadow information.*

4.2 Reduction steps

The operational semantics is given in several stages. We start with local steps, i.e., steps not involving shared variables.

4.2.1 Local steps The reduction steps are given modulo structural congruence $\equiv$ on configurations. The congruence rules are standard and given in Table 2. Besides specifying parallel composition as a binary operator of an Abelian monoid and with 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 [31].

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

Table 2: Structural congruence

Reduction modulo congruence and other "structural" rules are given in Table 3. The basic steps of the relations $\rightsquigarrow$ and $\rightarrow$ will be defined in the following, starting with local steps $\rightsquigarrow$.

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

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

Table 3: Congruence and reduction

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

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

$\mathtt{if\ true\ then}\, t_1\, \mathtt{else}\, t_2 \rightsquigarrow t_1$ R-COND$_1$ $\qquad$ $\mathtt{if\ false\ then}\, t_1\, \mathtt{else}\, t_2 \rightsquigarrow t_2$ R-COND$_2$

Table 4: Operational semantics: Local steps

Local steps $\rightsquigarrow$ (cf. Table 4) reduce a thread t without touching shared variables. The corresponding reduction relation $\rightsquigarrow$ is straightforward and can be formulated without referring to a local state. Rule R-LOCAL (from Table 3) "lifts" the local reduction relation to the global level of configurations.

4.2.2 Global steps 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 $n(\!|z{:=}v|\!)$ in the global configuration, with n freshly generated (cf. 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 local perspective of the threads.

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, that is, write events whose identifiers are 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 are stored in E_{hb}. So, issuing a write command (rule R-WRITE) with a write event labelled n updates the local E_{hb} by adding (n,z). Additionally, it marks all previous writes to the variable z (i.e., all writes which are known to have happened-before according to E_{hb}) as shadowed, thus enlarging E_s.

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.

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

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

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

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

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

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

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

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

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

Table 5: Operational semantics: Global steps

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 a channel. 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 from resp. sending to that channel is allowed. Very little 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.* Our semantics largely ignores 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.

Definition 2 (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, contains elements of type $(Val \times \Sigma) + (\{\perp\} \times \Sigma)$, where $\perp$ is a distinct, separate value representing the "end-of-transmission"; the second queue, q_2, contains elements of type Σ. We write (v,σ), $(\perp,\sigma)$ resp. (σ) for the respective queue values. The queues are also referred to as* forward *resp.* backward *queue. 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 not: it 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. An asynchronous channel is created with empty forward $c_f[]$ and backward queue $c_b[]$. Channel creation does not involve synchronization.

Channels can be closed, after which no new values can be sentotherwise 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 (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. Note that there is a difference between an empty open channel $c[]$ and an empty closed one $c[\perp]$. The value $\perp$ is relevant to the forward channel only. Rules R-SEND and R-REC govern asynchronous channel communication while R-SEND-REC 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, i.e., 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 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. Note that the R-SEND-REC 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-SEND-REC 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, this is captured by the absence of enabled transitions.

Thread creation leads to a form of a synchronization where the spawned goroutine inherits the local state of the parent (cf. rule R-GO). For completeness, rules dealing with the select statement are given in Appendix D.

Starting from an *initial weak configuration*, as far as the sizes of the queues of a channel in connection with the channel's capacity are concerned, the semantics assures the following invariant.

Definition 3 (Initial weak configuration). *An* initial *weak configuration is of the form*

$$\nu\vec{n}\ (\langle\sigma_0, t_0\rangle \parallel n_0(\!|z_0{:=}v_1|\!) \parallel \ldots \parallel n_k(\!|z_k{:=}v_k|\!))$$

where $z_0, \ldots z_k$ are all shared variables of the program, $\vec{n}$ represents $n_0, \ldots, n_k$, and $\sigma_0 = (E^0_{hb}, E^0_s)$ where $E^0_{hb} = \{(n_0, z_0), \ldots, (n_k, z_k)\}$ and $E^0_s = \emptyset$.

Lemma 4 (Invariant for channel queues). *The following global invariant holds for a channel c created with capacity k:*

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

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. □

Proof. By straightforward induction on the steps of the operational semantics. For open channels, the invariant holds initially upon creation of a channel (cf. rule R-MAKE), and rules R-SEND and R-REC preserve it. The invariant is stated only for non-closed channels; thus R-REC$_\perp$ (which applies only to channels previously closed) and R-CLOSE are both covered. The rest of the rules do not change the state of channels. When a channel is closed, $\perp$ is added to q_f and no element is removed for either queue; therefore, the sum of the queue sizes is increased by one. □

5 Relating the strong and the weak semantics

This section describes the relationship between the strong and the weak semantics. After some preliminary definitions, Section 5.2 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 5.3. Additional intermediate lemmas are relegated to the appendix, in particular Appendix C.

5.1 Preliminaries

Let's recall the definition of simulation [30] relating states of labelled 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 leads to a distinction between internally and externally visible steps. Let's 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 5 (Simulation). *Assume two labelled 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.*

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 4. 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 (see Appendix A).

The operational semantics is given as unlabelled 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 labelling 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" labelling 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, i.e. $\xrightarrow{p_2(z!)}$ and $\xrightarrow{p_2(z?)}$. Note that the identity of the write event is omitted as well as the value exchanged; they will not be needed in the proofs. We often use subscripts when distinguishing the strong from the weak semantics; e.g. $\xrightarrow{p(z!)}_w$ and $\xrightarrow{p(z!)}_s$. We write $\Rightarrow$ for $\xrightarrow{\tau}^*$ and $\overset{a}{\Rightarrow}$ for $\xrightarrow{\tau}^*\xrightarrow{a}\xrightarrow{\tau}^*$.

5.2 The weak semantics simulates the strong

Lemma 6 (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 $P_0 \gtrsim S_0$.*

Proof given on Appendix B.

5.3 The strong semantics conditionally simulates the weak one

That the weak semantics "contains" the sequentially consistent strong one as special case, i.e., the weak semantics simulates the strong one, should be intuitively clear and expected. 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,[6] 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 have access to the same shared variable "simultaneously," with at least one of the accesses being a write.

Definition 7 (Data race). *A (well-formed) configuration S_i contains a* manifest data race *if $S_i \xrightarrow{p_1(z!)}_s$ and $S_i \xrightarrow{p_2(z!)}_s$ for some $p_1 \neq p_2$ (a manifest write-write race on z), or if $S_i \xrightarrow{p_1(z?)}_s$ and $S_i \xrightarrow{p_2(z!)}_s$ (a manifest read-write race on z). We say a program S* has a data race *if a manifest data race is* reachable *from the initial configuration S_0.*

Definition 8 (Data race). *A (well-formed) configuration S_i contains a* manifest data race *if either hold:*

$$S_i \xrightarrow{p_1(z!)}_s \text{ and } S_i \xrightarrow{p_2(z!)}_s \text{ for some } p_1 \neq p_2 \qquad \text{(manifest write-write race on z)}$$

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

[6] 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.

We say a program S has a data race *if a manifest data race is* reachable *from the initial configuration S_0.*

We also say a program is data-race free (or *properly synchronized*) if it does not have a data race. Note that for the manifest read-write race, we don't need to require that the to transitions are done by different goroutines since that is already the only possible scenario.

The definition of races and proper synchronization is based on the *sequentially consistent* semantics. It is clearly possible to use the same definitions on the weak semantics as well. It's worthwhile to reflect on that: as established, the weak semantics does everything the sequentially consistent semantics does, and more. Therefore, every race in the strong semantics will be exhibited in the weak semantics as well (though the converse is not true). That there are really *new* races in the weak semantics for a given program comes from the fact that once a race is reachable, the weaker version of the semantics allows values to be read at points in the execution where those values would be unobservable to the corresponding sequentially consistent configuration. This, in turn, can lead to completely new behavior which may include additional races, of course. In other words, it's the *first* race condition in a run that sparks all subsequent additional behavior of the weak semantics. While the weak semantics does show more behavior in general, as long as there is no race, the weak behavior does not deviate from the strong one. Even if programs under a weak semantics have more races than under a strong semantics, their race-free behavior coincides. In particular, a program is race free in the strong semantics *iff* it is race free in the weak semantics. This is good news: it means that, in order to check for race freedom of a program in the weak semantics, it suffices to check for it in the sequentially consistent semantics, which is simpler. This is the fundamental common denominator of many relaxed memory models; namely, no matter what the model allows in general, you can rely at least on the following:

properly synchronized programs are sequentially consistent!

This is, of course, an informal discussion. Next we will prove that the weak semantics indeed has this property. The proof will be another simulation result: the strong semantics conditionally simulates the week one; the condition requires programs to be be data race free.

Definition 9 (Observable and concurrent writes). *Let W_P stand for the set of all write events $n(\!|z{:=}v|\!)$ in a weak configuration P and let $W_P(z)$ stand for the set of identifiers of writes events to the variable z:*

$$W_P(z) = \{n \mid n(\!|z{:=}v|\!) \in W_P\} \ . \tag{6}$$

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:*

$$W_P^{\mathsf{hb}}(z@p) = E_{hb}(z@p) \tag{7}$$

$$W_P^{|||}(z@p) = W_P(z) \setminus E_{hb}(z@p) \tag{8}$$

$$W_P^{\circ}(z@p) = W_P(z) \setminus E_s(z@p) \ . \tag{9}$$

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

5.3.1 General invariant properties

Let's 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 below are mostly relegated to Appendix C.2.2.

Lemma 10 (Invariants about write events). *The weak semantics has the following invariants.*

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

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

Lemma 11 (Consensus possible). *Weak configurations obey the following invariant*

$$\textstyle\bigcap_{p \in P} W_P^{\circ}(z@p) \neq \emptyset\ . \tag{10}$$

5.3.2 Race-free reductions

Next, we present invariants that hold specifically for race-free programs but not generally. They 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 12 (No concurrent writes when it counts). *Let P be a reachable configuration in the weak semantics, i.e., $P_0 \to_w^* P$ where P_0 is the initial configuration derived from program P.*

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

Lemma 13 (Race-free consensus when it counts). *Assume $P_0 \to_w^* P$ with P_0 race-free. If $P \xrightarrow{p(z?)}_w$ or $P \xrightarrow{p(z!)}_w$, then*

$$\textstyle\bigcap_{p_i} W_P^{\circ}(z@p_i) = \{n\}\ , \tag{11}$$

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

Lemma 14 (Race-free consensus). *Weak configurations for race-free programs obey the following invariant*

$$\textstyle\bigcap_{p_i \in P} W_P^{\circ}(z@p_i) = \{n\}\ . \tag{12}$$

Definition 15 (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 (12) from Lemma 14 on the preceding page 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 16 (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 6 and R is a set of write event identifiers. On the queues q_1 and q_2 in the last case, the function simply jettisons the σ-component in the queue elements.*

$$\lfloor \bullet \rfloor^R = \bullet \tag{13}$$

$$\lfloor p\langle \sigma, t\rangle \rfloor^R = \langle t \rangle \tag{14}$$

$$\lfloor n(\!|z{:=}v|\!) \rfloor^R = \begin{cases} \bullet & \text{if } n \in R \\ (\!|z{:=}v|\!) & \text{otherwise} \end{cases} \tag{15}$$

$$\lfloor P_1 \parallel P_2 \rfloor^R = \lfloor P_1 \rfloor^R \parallel \lfloor P_2 \rfloor^R \tag{16}$$

$$\lfloor \nu n\, P \rfloor^R = \begin{cases} \lfloor P \rfloor^R & \text{if } \forall p \in P.\ n \in W_p^{\circ}(\_@p) \\ \lfloor P \rfloor^{R \cup \{n\}} & \text{otherwise} \end{cases} \tag{17}$$

$$\lfloor c[q_1, q_2] \rfloor^R = c[\lfloor q_1 \rfloor^R, \lfloor q_2 \rfloor^R] \tag{18}$$

Table 6: Definition of the erasure function $\lfloor P \rfloor^R$

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

Lemma 17 (Erasure and congruence). *$P_1 \equiv P_2$ implies $\lfloor P_1 \rfloor \equiv \lfloor P_2 \rfloor$.*

Proof. Straightforward. □

Lemma 18 (Erasure preserves well-formedness). *Let P be a race-free reachable weak configuration. If $\vdash_w P : ok$ then $\vdash_s \lfloor P \rfloor : ok$.*

Proof. Assume $\vdash_w P : ok$. The one condition for well-formedness of $\lfloor P \rfloor$ that requires checking is that strong configurations contain exactly one write event $(\!|z{:=}v|\!)$ for each shared variable. To not be erased by $\lfloor P \rfloor$, a write-event in P must be observable by all processes (cf. Definition 16, especially the crucial case for ν-binders from equation (17)). By Lemma 14, there is exactly one such universally observable write event per variable. □

Theorem 19 (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

$$P \,\mathcal{R}\, S \quad \text{if} \quad S \equiv \lfloor P \rfloor \tag{19}$$

using the erasure from Definition 16. Note that by Lemma 17, $P_1 \;\mathcal{R}\; S$ and $P_1 \equiv P_2$ implies $P_2 \;\mathcal{R}\; S$.

Case: R-WRITE$_w$: $p\langle \sigma, z := v; t\rangle \rightarrow_w \nu n\, (p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!))$,
where $\sigma = (E_{hb}, E_s)$ and $\sigma' = (E'_{hb}, E'_s) = (E_{hb} + (n,z), E_s + E_{hb}(z))$. By the concurrent-writes Lemma 12(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 $n'(\!|z{:=}v'|\!)$ in P, we have $n' \in E_{hb}$. If $n' \in E_s$, then $n \in E'_s$ as well. If $n' \in E_{hb} \setminus E_s$, then $n' \in E'_s$ 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 n in P': clearly $n \in W^{\circ}_{P'}(z@p_i)$, i.e., the event is observable for all threads. By the race-free consensus Lemma 14, we have that this is the only event that is observable by all threads, i.e.

$$\bigcap_{p_i} W^{\circ}_{P'}(z@p_i) = \{n\} \; . \tag{20}$$

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 $n(\!|z{:=}v|\!)$ of P. In particular, equation (20) shows that the write event n is not "filtered out" (cf. the cases of equation (15) and (17) in Definition 16) and furthermore that all other write events for z in P' *are* filtered out.[7] 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] The latter is indirectly clear already as we have established that $\lfloor\rfloor$ preserves well-formedness under the assumption of race-freedom (Lemma 18).

6 Implementation

We have implemented the strong and the weak semantics in $\mathbb{K}$ [15], a rewrite-based executable semantics framework [23, 36]. To give the reader a flavor of the rewriting rules used, Figure 2 shows part of the implementation of R-RECEIVE of Table 5 on page 11.

The implemented rewriting rule is enabled when a goroutine is receiving on a channel with identifier Ref, the channel contains one element in its forward queue, and that element is not the special end-of-transmission marker. 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) which corresponds to the value at the head of the forward queue (line 13). The goroutine's happens-before and shadowed information (HMap and SSet resp.) are rewritten on lines 4 and 5 to take into account the happens-before and shadow information in the forward queue (HMapDp and SSetDp on lines 14 and 15 resp.). The received entry is removed from the forward queue (lines 13-15), and the receiver's local state is added to the channel's backward queue (lines 16 and 17).

```
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          <select> _ </select>
8          <id> _ </id>
9        </goroutine>
10       <chan>
11         <ref> Ref </ref>
12         <type> _ </type>
13         <forward> ListItem( ListItem(V)
14                    ListItem(HMapDP)
15                    ListItem(SSetDP) ) => .List </forward>
16         <backward> BQ:List => ListItem( ListItem(HMap)
17                                ListItem(SSet) ) BQ </backward>
18       </chan>
19       requires notBool( V ==K $eot )
```

Fig. 2: Snippet from the implementation of the channel receive rule in $\mathbb{K}$

A byproduct of the implementation is the ability to execute programs and observe their output. To illustrate, Listing 1.3 and Figure 4 show the source code and corresponding output for the synchronization example given earlier on Listing 1.2 on page 6. The program is written in the syntax proposed in Section 3 and executed by an interpreter derived from the reduction rules of Section 4. In the example, an initial goroutine creates a channel, starts a second goroutine that performs some setup, reads from a channel, and then utilizes resources initialized by setup (since we do not have a print-statement, we declare the shared variable to be an integer instead of string). The goroutine performing the setup writes 42 to a shared variable and then sends a message onto the channel. The message indicates that the setup is complete and, according to

Listing 1.3: Channel synchronization

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

the semantics of channel communication, the message receiver can no longer read the initial shared variable's value 0 and will instead read an updated value.

Figure 3 shows the runtime configuration at the start of execution. It features a

```
1  <mmgo>
2    <T>
3      <goroutine>
4        <id> 1 </id>
5        <k> $PGM:Pgm </k>
6        <select> .SelBranchCellBag </select>
7        <sigma>
8          <HB> .Map </HB>
9          <S> .Set </S>
10       </sigma>
11     </goroutine>
12   </T>
13   <W> .Map </W>
14   <C> .ChanCellBag </C>
15 </mmgo>
```

Fig. 3: The initial runtime configuration

single goroutine with id 1 (line 4) and empty happens-before and shadowed information (lines 8 and 9 resp.). The tokens $PGM:Pgm stand for a syntactically valid program. It is filled by $\mathbb{K}$ when execution starts. The tags <W> and <C> respectively correspond to the fact that no write events have been recorded and no channels have been created yet. If the program to be executed declares shared variables, the implementation initializes them to 0 inside <W>.

As execution progresses, meaning, as write events are recorded and additional goroutines and channels are created, the configuration is expanded. Figure 4 shows the end configuration[8] outputted from an execution of the program in Listing 1.3. The end configuration has two goroutines inside the top-level tag <T>. The "main" goroutine (i.e. the goroutine existing from the start of the program's execution) terminates in the state 42 (line 5), which corresponds to the value it read from the channel. The "setup" goroutine (i.e. the one spun by "main") terminates in the state unit (line 14), which is the value resultant from executing its last instruction, namely c<-0. The tag <W> capture all write events. In this case there are two: one coming from the initial configuration establishing 0 as the initial value for the global variable x, and another corresponding to the write of 42 into x by the "setup" goroutine. The tag <C> captures all created channels.

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

```
 1 <mmgo>
 2   <T>
 3     <goroutine>
 4       <id> 1 </id>
 5       <k> 42 </k>
 6       <select> .SelBranchCellBag </select>
 7       <sigma>
 8         <HB> x |-> ( SetItem ( 3 ) SetItem ( 7 ) ) </HB>
 9         <S> SetItem ( 3 ) </S>
10       </sigma>
11     </goroutine>
12     <goroutine>
13       <id> 5 </id>
14       <k> $unit </k>
15       <select> .SelBranchCellBag </select>
16       <sigma>
17         <HB> x |-> ( SetItem ( 3 ) SetItem ( 7 ) ) </HB>
18         <S> SetItem ( 3 ) </S>
19       </sigma>
20     </goroutine>
21   </T>
22   <W> x |-> ( 3 |-> 0 7 |-> 42 ) </W>
23   <C>
24     <chan>
25       <ref> 4 </ref>
26       <type> int </type>
27       <forward> .List </forward>
28       <backward> ListItem ( ListItem ( x |-> SetItem ( 3 ) ) ListItem ( .Set ) )
29                  ListItem ( ListItem ( .Map ) ListItem ( .Set ) ) </backward>
30     </chan>
31   </C>
32 </mmgo>
```

Fig. 4: Sample output from running Listing 1.3 on the weak semantics

In this case, there is only one channel. Channels have an unique identifier (line 25), a type (line 26), a forward queue (line 27), which in this case is empty since there are no values waiting to be read from the channel, and a backward queue (lines 28 and 29). The backward queue holds the happens-before and shadowed sets of the "setup" goroutine at the time that it sent 42 on the channel (line 28). Finally, note that the other element of the backward queue (line 29) is the empty local state $\sigma_\perp$ described by Definition 1 on page 9 of the semantics, and that the invariant for channel queues (Lemma 4 on page 13) holds.

7 Related work

There are numerous proposals for and investigations of weak and relaxed memory models [1, 29, 3]. One widely followed approach, called *axiomatic*,[9] specifies allowed behavior by defining various ordering relations on memory accesses like read and writes and other more specific events like synchronization operations, fences, or channel sends and receives. The description of Go's memory model [19] gives an informal impression of that style of specification. Less frequent are *operational* formalizations.

[9] The notion is not directly related to Hoare-style axiomatic semantics based on pre- and post-conditions.

Boudol and Petri investigate a relaxed memory model for a calculus with locks relying on concepts of rewriting theory, like parallel reductions and semantics via evaluation contexts [11]. Unlike the presentation here, writes are buffered in a hierarchy of fifo-buffer 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, i.e. the terms representing configurations cannot be interpreted up-to congruence $\equiv$ as in our case. Flanagan and Freund give an operational semantics of a weak memory model ("adversarial" memory) used as the basis for a race checker. [17]. The model is not as weak as the official JMM but weaker than memory guarantees of standard JVM implementations. Zhang and Feng present an operationalization of (a variation of) a happens-before memory model using an abstract machine representation [38]. Different from us, they make use of event *buffers*, and similar to our semantics, their semantics keep "older" write events (called "history-based memory" and "replay buffers") 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., where threads can promise future writes, and a reader acquires information on the writer's view on memory. Fences then synchronize global time-stamps on memory with thread-local information. Bi-simulation proofs mechanized in Coq show correctness of compilation to various architectures, and the model is shown to capture the desired effects with regard to reorderings on those architectures, as well as compiler optimizations. Demange et al. formalize a weak semantics for Java using buffers (BMM) [13]. 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 BMM is defined axiomatically and operationally and the equivalence of the two formalizations is established. Pichon-Pharabod and Sewell 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 [34]. The semantics is studied in a calculus featuring locks as well as relaxed atomics and non-atomic memory accesses. It makes use of event-structures to represent thread-local information about memory. Guerraoui et al. introduces RML, 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 HW [20].

Alrahman et al. formalize and represent a small calculus with a relaxed memory model (total-store order TSO) and with a fence and a wait operation for synchronization [4]. They provide an implementation in Maude (a rewriting-based executable framework that precedes $\mathbb{K}$) and explore ways to battle the state-space explosion (for instance by making use of partial order reduction and finding appropriate abstractions). Lange et al. define a small calculus dubbed MiGo ("mini-Go") featuring channels and thread creation [27]. The formalization does not cover the weak memory model and the semantics for the process calculus is given is a standard way. The paper instead focuses on a communication analysis for channel usage making use of an intricate behavioral effect type system.

8 Conclusion

We present an *operational* specification for a weak memory model with channel communication as the prime means of synchronization. We prove the central guarantee of such models, namely that race-free programs behave sequentially consistently. The semantics is accompanied by an implementation in the $\mathbb{K}$ framework along with several examples and test cases [15]. We believe that the implementation of the strong and the weak semantics will be useful to verify program properties, e.g., via model checking. To do so, state-space reduction techniques (similar to the ones investigated by Alrahman et al. when representing a TSO memory model in the Maude rewriting framework [4]) will be needed.

Currently the semantics remembers past write events as part of the run-time configuration, but does not remember read events. Treating read events similar to the representation of writes would relax (albeit complicate) the model. We have started work on that direction.

Bibliography

[1] Adve, S. V. and Gharachorloo, K. (1995). Shared memory consistency models: A tutorial. Research Report 95/7, Digital WRL.

[2] Adve, S. V. and Hill, M. D. (1990). Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a).

[3] Alglave, J., Maranget, L., and Tautschnig, M. (2014). Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2).

[4] Alrahman, Y. A., Andric, M., Beggiato, A., and Lluch-Lafuente, A. (2014). Can we efficiently check concurrent programs under relaxed memory models in Maude? In Escobar, S., editor, *Rewriting Logic and Its Applications – 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 21–41. Springer Verlag.

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

[6] Aspinall, D. and Ševčík, J. (2007). Java memory model examples: Good, bad and ugly. *Proc. of VAMP*, 7.

[7] Batty, M., Mamarian, K., Nienhuis, K., Pinchion-Pharabod, J., and Sewell, P. (2015). The problem of programming language concurrency semantics. In Vitek, J., editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Verlag.

[8] Becker (2011). Programming languages — C++. ISO/IEC 14882:2001.

[9] Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[10] Boehm, H.-J. and Demsky, B. (2014). Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA. ACM.

[11] Boudol, G. and Petri, G. (2009). Relaxed memory models: An operational approach. In *Proceedings of POPL '09*, pages 392–403. ACM.

[12] Castagna, G. and Gordon, A. D., editors (2017). *44th Symposium on Principles of Programming Languages (POPL)*. ACM.

[13] Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., and Vitek, J. (2013). Plan B: A buffered memory model for Java. In *Proceedings of POPL '13*, pages 329–342. ACM.

[14] Donovan, A. A. A. and Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.

[15] Fava, D. (2017). Operational semantics of a weak memory model with channel synchronization. `https://github.com/dfava/mmgo`.

[16] Fava, D., Steffen, M., and Stolz, V. (2018). An operational semantics for a weak memory model with channel synchronization. In *Submitted for publication*.

[17] Flanagan, C. and Freund, S. N. (2010). Adversarial memory for detecting destructive races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[18] Go language specification (2016). The Go programming language specification. `https://golang.org/ref/spec`.

[19] Go memory model (2014). The Go memory model. `https://golang.org/ref/mem`. Version of May 31, 2014, covering Go version 1.9.1.

[20] Guerraoui, R., Henzinger, T. A., and Singh, V. (2009). Software transactional memory on relaxed memory models. In Bouajjani, A. and Maler, O., editors, *Proceedings of CAV '09*, volume 5643 of *Lecture Notes in Computer Science*, pages 321–336. Springer Verlag.

[21] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

[22] Jones, G. and Goldsmith, M. (1988). *Programming in occam2*. Prentice-Hall International, Hemel Hampstead.

[23] K framework (2017). The K framework. available at `http://www.kframework.org/`.

[24] Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017). A promising semantics for relaxed-memory concurrency. In [12], pages 175–189.

[25] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[26] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

[27] Lange, J., Ng, N., Toninho, B., and Yoshida, N. (2017). Fencing off Go: Liveness and safety for channel-based programming. In [12].

[28] Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory memory. In *Proceedings of POPL '05*. ACM.

[29] Maranget, L., Sarkar, S., and Sewell, P. (2012). A tutorial introduction to the ARM and POWER relaxed memory models (version 120).

[30] Milner, R. (1971). An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann.

[31] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77.

[32] Palamidessi, C. (1997). Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM.

[33] Peters, K. and Nestmann, U. (2012). Is it a "good" encoding of mixed choice? In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '12)*, volume 7213 of *Lecture Notes in Computer Science*, pages 210–224. Springer Verlag.

[34] Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency-semantics for relaxed atomics that permits optimisation and avoids out-of-thin-air executions. In *Proceedings of POPL '16*. ACM.

[35] Pugh, W. (1999). Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*.

[36] Roşu, G. and Şerbănuţă, T. F. (2010). An overview of the K semantic framework. *Journal of Logic and Algebraic Methods in Programming*, 79(6):397–434.

[37] Steffen, M. (2016). A small-step semantics of a concurrent calculus with goroutines and deferred functions. In Ábrahám, E., Huisman, M., and Johnsen, E. B., editors, *Theory and Practice of Formal Methods. Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday (Festschrift)*, volume 9660 of *Lecture Notes in Computer Science*, pages 393–406. Springer Verlag.

[38] Zhang, Y. and Feng, X. (2016). An operational happens-before memory model. *Frontiers in Computer Science*, 10(1):54–81.

A Strong semantics

The strong semantics can be seen as a simpler version of the weak one. It represents a standard interleaving semantics, i.e., write and reads immediately interact with a shared global state. Therefore, there is no need for local thread information σ.

$$S ::= p\langle t\rangle \mid (\!|z{:=}v|\!) \mid \bullet \mid S \parallel S \mid n[q] \mid \nu n\, P\,. \tag{21}$$

Structural congruence $\equiv$ and the local transition steps $\rightsquigarrow$ remain unchanged (cf. Tables 2, 3, and 4). Apart from leaving out the events and other information, the only rules that conceptually change are the ones for read and write. These are included on Table 7.

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

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

Table 7: Strong operational semantics: read and write steps

Definition 20 (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.*

Cf. also the "weak" version from Definition 3 on page 13.

Definition 21 (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.*

B The weak semantics simulates the strong

Proof (of Lemma 6 on page 15 (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 notationally 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:

$$\sigma_P^{\overline{N}}(z) = \{v \mid n(\!|z{:=}v|\!) \in P \text{ and } n \notin N\} \ . \tag{22}$$

We then define the relation $\mathcal{R}$ between well-formed strong and weak configuration over the same set of shared variables as follows: $S \mathrel{\mathcal{R}} P$ if $\lfloor P \rfloor = S$ (as far as goroutines and channels is concerned) and furthermore, for each goroutine $p\langle(\_, E_s), t\rangle$ in P, and all shared variables z,

$$\sigma_S(z) \in \sigma_P^{\overline{E_s}}(z) \ . \tag{23}$$

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, $S \mathrel{\mathcal{R}} P$ implies that P contains a goroutine $p\langle\sigma, z := v; t\rangle$. Doing the corresponding weak step $P \xrightarrow{\tau}_w P'$ yields

$$P' = \nu n\ (p\langle\sigma', t\rangle \parallel n(\!|z{:=}v|\!))$$

where $\sigma' = (E'_{hb}, E'_s)$. Since n is a fresh name, it is not mentioned in any shadow set of any thread, in particular $n \notin E'_s$. Consequently, S' and P' satisfy the condition from equation (23) 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, $S' \mathrel{\mathcal{R}} P'$ as required.

Case: R-READ$_s$: $p\langle \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!) \xrightarrow{(z?v)}_s p\langle \texttt{let}\ r =\ v\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!)$
$S \mathrel{\mathcal{R}} P$ implies that P contains $p\langle(\_, E_s), z := v; t\rangle$ and write events $n(\!|z{:=}v|\!)$ (there may be more than one for z and v, but with different identifiers); specifically condition (23) guarantees that there exists one $n(\!|z{:=}v|\!)$ such that $n \notin E_s$, which enables R-READ$_w$ for P such that $P \xrightarrow{(z?v)}_w P'$ with $S' \mathrel{\mathcal{R}} P'$, as required.

The remaining cases are analogous or simpler, establishing that $\mathcal{R}$ is a simulation relation. It's immediate that the corresponding initial configurations are related, i.e., $S_0 \mathrel{\mathcal{R}} P_0$. Thus $P_0 \gtrsim S_0$, which concludes the proof. $\square$

C Proofs via a weak semantics augmented with read and write events

This section contains supplementary material and proofs for the lemmas of Section 5. In particular, the material here allows us to carry out the harder direction of the simulation proof of Section 5.3, namely that the strong semantics simulates the weak one for race-free programs.

We start in Section C.1 augmenting the weak semantics with additional information which has no relevance aside from assisting the proofs. Besides this additional information, it will be obvious that the steps of the weak semantics and of the variant semantics provided here are identical. Section C.2 covers properties of the augmented semantics.

C.1 Augmenting the weak semantics

This section presents an "alternative" representation of the weak semantics of Section 4. 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 and remembered as part of the configurations. In particular, the weak semantics from Section 4 makes use of write events as part of configurations. Read steps, however, were not treated in 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 (5) on page 9 are therefore adapted to contain events of the following form:

$$n(\!|\sigma, z := v|\!)_p \quad \text{and} \quad n[\![\sigma, ?z]\!]_p \ , \tag{24}$$

where $n(\!|\sigma, z := v|\!)_p$ are write events augmented with the local state σ and identity p of the issuing thread and $n[\![\sigma, ?z]\!]_p$ are read events augmented in the same manner.

Notation 22 (Events) *We use e for events and r and w for read and write events in specific. When talking about two different events, we generally assume that their identities are different. It is an invariant of the semantics that the labelling of the events are indeed unique. Furthermore, let e be an event with identifier n and referring to variable z. Instead of writing $n \in E_s$ 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 $(n, z) \in E_{hb}$.*

Concerning the operational rules from Table 5 on page 11, only the read and the write steps require adaptation. See Table 8 for the augmented rules, which behave exactly as the originals except that the steps now record additional information as part of the configuration.

Substituting R-WRITE and R-READ by their augmented version R-WRITE$_\sigma$ and R-READ$_\sigma$ of Table 8 yields an operational semantics that is obviously equivalent to the original one from Section 4: 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 5.

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

$$\frac{\sigma = (\_, E_s) \qquad n \notin E_s \qquad fresh(m)}{\begin{array}{l} p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel n(\!|\_, z := v|\!)_{\_} \;\; \to \\ \quad \nu m\ (p\langle \sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel n(\!|\_, z := v|\!)_{\_} \parallel m[\![\sigma, ?z]\!]_p) \end{array}} \ \textsc{R-Read}_\sigma$$

Table 8: Operational semantics: Read/write rules with augmented read/write events

C.2 Additional concepts and lemmas

In the following, we use $\to_w$, $\to_w^*$ etc. when referring to the steps of the augmented weak semantics, which we will, from now on, refer to simply as the "weak semantics" (unless stated otherwise).

We define three binary relations between events given the augmented read and write events. First, the *happens-before* relation, which can now be gathered from the augmented event information. Events are considered *concurrent* if unordered by the happens-before relation. Combinations of read-write resp. write-write events are in *conflict* if they are concurrent and concern the same variable. These definitions generalize Definition 9 on page 16 from the main part of the paper.

Definition 23 (Binary relations on events). *Let e_1 and e_2 be two different events, with E^2_{hb} the happens-before set of e_2 and n_1 the identity of e_1.*

1. *e_1* happens-before *e_2, written $e_1 \to_{\mathsf{hb}} e_2$, if $n_1 \in E^2_{hb}$.*
2. *e_1 and e_2 are concurrent, written $e_1 \parallel\!| e_2$, if neither $e_1 \to_{\mathsf{hb}} e_2$ nor $e_2 \to_{\mathsf{hb}} e_1$.*
3. *e_1 and e_2 are* in conflict*, written $e_1 \# e_2$, iff $e_1 \parallel\!| e_2$, both event concern the same variable, and one of the events is a write.*

We denote read/write conflicts as $\#^{rw}$ and write/write as $\#^{ww}$. We also say that a configuration contains a conflict if it contains two different events which are in conflict. Note that we need the augmented notion of configurations in order to obtain this definition; the original notion of weak configuration does not contain enough information to "detect" conflicts (not to mention, that read events were not even recorded). Note that the definition of $\to_{\mathsf{hb}}$ is slightly asymmetric: only the happens-before information from e_2 is relevant when defining $e_1 \to_{\mathsf{hb}} e_2$ (as e_1 does not have information about events that "happen-after"). See also Lemma 24, stating that $\to_{\mathsf{hb}}$ is a partial order.

Lemma 24 (Simple properties of event relations).

- *$\#$ and $\parallel\!|$ are symmetric, irreflexive by definition, but not transitive.*
- *$\#^{ww}$ is not transitive.*

Furthermore, all reachable configurations we have the following invariants:

- *$\to_{\mathsf{hb}}$ 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 $e_1 \mid\mid\mid e_2$ implies $p_1 \neq p_2$.*

Proof. # and $\mid\mid\mid$ 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 9 on page 16. 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 [19].

Definition 25 (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 9 on page 16). Note, however, that the transition-based definition from Section 5.3 does *not* include condition (2) from Definition 25. Even if the two definitions differ concerning that condition, they are intuitively capturing the same concept: In the earlier Definition 9, 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 25 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 ocurred 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.

C.2.1 Connecting the two versions of the weak-semantics We now have two views on what a "race" or "conflict" is: one from the original semantics and one from the augmented one.

The **original semantics** provides a transition-based view captured in the notion of *manifest* data race. It says that a read-write race or a write-write race exists if two conflicting accesses to a common variable can happen "at the same time." In the interleaving formulation of the semantics, which cannot actually execute two steps simultaneously, this is captured by a situation where two conflicting steps *could both* be executed next (see Definition 8 on page 15 of a manifest data race in a configuration).

In the **augmented semantics** from Section C.1, there is the notion of conflicting read or write events as part of the augmented configuration.

The following lemma is crucial; it connects the original and the augmented views.

Lemma 26 (Races and conflicts). *Let P be a reachable, conflict-free 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.* □

Proof. Assume $P_0 \rightarrow_w^* P$ where P_0 is an initial configuration. There are two directions to establish. We show the proof for w/w-races resp. w/w-conflicts. The argument is completely analogous for the r/w-situation.

Case: $\Rightarrow$

This direction is immediate: we are given that P has a w/w-race. This means with Definition 8 on page 15 that $P \rightarrow_w^* P'$ with P' containing a manifest w/w-race, i.e., $P' \xrightarrow{p_1(z!)}_s$ and $P' \xrightarrow{p_2(z!)}_s$ for some $p_1 \neq p_2$. This directly means $P' \xrightarrow{p_2(z!)}_s \xrightarrow{p_2(z!)}_s P''$ for some P'' where P'' contains two conflicting write events. Note in passing that the order of steps from P to P'' does not matter. Additionally, for this direction of the proof, we don't make use of the assumption that P is conflict-free.

Case: $\Leftarrow$

Assume now that P is conflict-free and $P \rightarrow_w^* P'$ with P' containing a w/w-conflict, i.e. there are two write events in P' with $w_1 \#_z w_2$ for some z. That means the reduction from P to P' is of the form

$$P \rightarrow_w^* \xrightarrow{p_1(z!)}_w \rightarrow_w^n \xrightarrow{p_2(z!)}_w \rightarrow_w^* P' .$$

By straightforward induction on the length n of the steps separating the two write-steps, one can establish that there exists an alternative reduction $P \rightarrow_w^* P'' \xrightarrow{p_1(z!)}_w \xrightarrow{p_2(z!)}_w \rightarrow_w^* P'$, which in turn implies that P'' has a manifest w/w-race. □

C.2.2 General invariant properties

See also Section 5.3.1 in the main part.

Lemma 27 (Invariants). *For all reachable configurations, we have the following invariants.*

1. *For all events e resp. processes with local state (E_{hb}, E_s), $E_s \subset E_{hb}(z)$.*
2. *$w \,|||\, r$ implies $w \rightarrow_{\mathsf{o}} r$.*
3. *For each read event r, there exists a write event w with $w \rightarrow_{\mathsf{o}} r$ and not $w \,|||\, r$.*
4. *For each read event r, there exists a write event w with $w \rightarrow_{\mathsf{o}} r$ and $w \rightarrow_{\mathsf{hb}} r$.*

Proof. By straightforward induction. □

The next proof establishes **Lemma 8** in the short version of [16].

Proof (of the invariants Lemma 10 on page 17). A straightforward consequence of the corresponding property for read and write events of the augmented semantics from Lemma 27 on the facing page. $\square$

The next proof refers to the **Consensus possible Lemma 9** in the short version [16].

Proof (of Lemma 11 ("consensus possible")). The property holds for an initial configuration P_0 because:

- it contains one write event for each shared variable and
- the initial process's shadowed set is empty.

Therefore, every process observe, for each variable, the same initial value. Assuming $W^{\circ}_{P_i}(z@p) \neq \emptyset$ where $P_0 \to^*_w P_i$ 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 n that is not in any process's shadowed set at P_i. Since shadowed sets are monotonically increasing, $n \notin E''_s$. Since $n \notin E_s$ and $n \notin E''_s$, then $n \notin E_s \cup E''_s$. This means n 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-SEND-REC
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 n that is not in any process's shadowed set at P_i; therefore, $n \notin E_s$ and $n \notin E'_s$ in specific. By the definition of R-SEND-REC, the sender's and receiver's shadowed sets at P_{i+1} is $E_s \cup E'_s$. Since $n \notin E_s$ and $n \notin E'_s$, then $n \notin E_s \cup E'_s$. Finally, since at P_{i+1} the sender's and receiver's shadowed sets do not contain n, 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 crucial 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*. Since the notion of conflicting events in the augmented weak semantics is in close correspondence with the notion of races (as established in Lemma 26), the lemma allows us to conclude that for race-free programs, there is no ambiguity when observing write events.

Lemma 28 (Observability and conflicts). *The weak semantics has the following invariant: If $w_1 \to_{\mathsf{o}}^{x} r \leftarrow_{\mathsf{o}}^{x} w_2$ 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 $w_1 \to_{\mathsf{hb}} w_2$ and both are concurrent wrt. the read event. If, in particular $w_1 \to_{\mathsf{hb}} w_2$, $w_1 \to_{\mathsf{hb}} r$, and $w_2 \parallel\!| \, r$, then $w_2 \# r$ but w_1 is not in conflict with any of the other two events.

C.2.3 Race-free resp. conflict-free reductions

See also Section 5.3.2 on page 17 in the main part of the paper.

Lemma 29 (Uniqueness of observability). *Let P be a reachable, conflict-free configuration in the augmented semantics. If P is race-free and $P \to_w^* P'$, then for all events in P' and all variables z we have*

$$|\{w \mid w \leftarrow_{\mathsf{o}}^{z} r\}| \leq 1 \tag{25}$$

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 $w_1 \to_{\mathsf{o}} r$ and $w_2 \to_{\mathsf{o}} r$. By Lemma 28, this implies that P' contains at least two conflicting events. With Lemma 26, the existence of conflicting events contradicts the assumption of race-freedom, which concludes the proof. □

Corollary 30. *Let P, P' and z be given as in Lemma 29. Then we have*

$$|\{w \mid w \leftarrow_{\mathsf{o}}^{z} r\}| = 1 \; . \tag{26}$$

Proof. A direct consequence of 29 and of Lemma 27(3) (or alternatively Lemma 27(4)). □

The proof concerns **Lemma 10 ("no concurrent writes when it counts")** of the short version [16].

Proof (of Lemma 12 on page 17 (no concurrent write when it counts)). A direct consequence of the equivalence of races and conflicts from Lemma 26. Assume for a contradiction $P \xrightarrow{p(z?)}_w P'$ and $W_P^{|||}(z@p) \neq \emptyset$. Then P' contains two events r and w with $r \# w$. With Lemma 26, this contradicts the assumption that P_0 has no r/w race. The case for w/w races is analogous. □

Lemma 31 (Unique observability when it counts). *Assume $P_0 \rightarrow^*_w P$ with P_0 race-free. If $P \xrightarrow{p(z?)}_w$ or $P \xrightarrow{p(z!)}_w$, then*

$$W^{\circ}_P(z@p) = \{n\}\ . \tag{27}$$

Proof. For the write step: assume that there are two different observable writes w_1 and w_2. By Lemma C.2.3, $W^{|||}_P(z@p) = \emptyset$. By Definition 9, that means all observable writes are in happens-before relation, i.e., $W^{\circ}_P(z@p) = W^{\mathsf{hb}}_P(z@p)$. 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 Lemma 26, that contradicts the assumption of race-freedom.

The case for a read-step is analogous (alternatively it follows from Lemma 29). □

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

The next proof covers **Lemma 11 ("Race-free consensus when it counts")** from the short version [16].

Proof (of Lemma 13 on page 17 ("race-free consensus when it counts")). A direct consequence of unique observability from Lemma 31 and the possible consensus property from Lemma 11 on page 17. □

Corollary 32 (Locally deterministic read). *Assume $P_0 \rightarrow^*_w P$ with P_0 race-free. Then $P \xrightarrow{pn_1(?)}_w$ and $P \xrightarrow{pn_2(?)}_w$ implies $n_1 = n_2$.*

Proof. A direct consequence of the consensus Lemma 13. □

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 13).

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 that is used in the strong one. Without the race-free consensus lemma, the construction would not be well-defined: the erasure function $\lfloor \_ \rfloor$ from Definition 16 on page 18 would not be a function, resp. would not yield well-formed strong configurations.

The next proof corresponds to the **race-free consensus Lemma 12** from the short version [16]

Proof (Lemma 14 on page 17 (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 $P \xrightarrow{p(z!)}_w P'$. By Lemma 12(1), there are no concurrent writes for p before the step, i.e., $W_P^{|||}(z@p) = \emptyset$. By Definition 9, that means all observable writes are in happens-before relation, i.e., $W_P^{\circ}(z@p) = W_P^{\mathsf{hb}}(z@p)$.[10] Consequently, after the $\xrightarrow{p(z!)}_w$ 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 $P \xrightarrow{p(z!)}_w P'$. This is a *new* write event in P with a fresh identity, say, n', which consequently is not mentioned in the shadow set of any process. Therefore, $\bigcap_{p_i \in P'} W_P^{\circ}(z@p_i) = \{n'\}$, establishing the invariant for the post-configuration P'. □

D Rules for the select statement

Rules dealing with the select statement in the weak semantics are given on Table 9. 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⊥ is analogous to R-REC⊥. Finally, the default rule (R-SEL-DEF) applies when no other select rule applies.

Index

$\equiv$ (structural congruence), 9
$|||$ (happens-before), 30
# (conflict), 30
$\rightarrow_{\mathsf{hb}}$ (happens-before), 30
$\bullet$ (empty configuration), 9
$\rightarrow^z_{\mathsf{o}}$ (observable), 31
$\perp$ (end-of-transmission), 12
$n(\!|x{:=}v|\!)$, 9
$n(\!|z{:=}v|\!)$ (write event), 9
$c_b[q]$ (backward channel), 12
$c_f[q]$ (forward channel), 12
$c[q_1,q_2]$, 12
$\leftarrow c$ (receive), 7
$c \leftarrow v$ (send), 7
$E_{hb}(z)$, 9
σ (local state), 9
$\sigma_\perp$ (empty local state), 9, 12
π-calculus, 9
$\lesssim$ (simulation), 14
$|q|$ (number of elements in q), 12
$\sigma_P^{\overline{N}}$, 28
$z := v$, 7
$\lfloor \_ \rfloor$ (erasure), 27
$\rightsquigarrow$ (local transition step), 9, 10
r (local variable), 7

[10] One could establish that there is exactly one such event, but it's not needed for the proof. The important property here is that there are no concurrent observable writes.

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

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

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

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

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

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

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

Table 9: Operational semantics: Select statement

A-normal form, 8
abstract syntax, 7
at-most-once property, 7

bisimulation, 15

channel, 11, 12
– closed, 12
channel name, 7
choice
– mixed, 8
closed, 12
closed channel, 12

configuration
– well-formed, 27

data race, 15
– manifest, 15
dynamic scope, 9

end-of-transmission, 12
erasure, 27
events, 29

happens-before, 23

`let`, 8

`load` z, 7
local state, 9
local variable, 7

`make` $(\texttt{chan}\ T, v)$, 7
memory model, 3
mixed choice, 8

N (names), 9

panic, 7
properly manifest, 15
properly synchronized program, 16

SC-DRF, 35
sequential composition, 8
simulation, 14
`stop`, 8
structural congruence, 9
synchronization, 11
– properly synchronized program, 16
synchronous channel, 12

value, 7

well-formed configuration, 27

X (variables), 9