An operational semantics for a weak memory model with buffered writes, message passing, and goroutines

Daniel Fava, Martin Steffen, Volker Stolz, and Stian Valle

Technical report Nr. 466

IFI
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

April 21, 2017

An operational semantics for a weak memory model with buffered writes, message passing, and goroutines

Daniel Fava, Martin Steffen, Volker Stolz, and Stian Valle

21 April 2017

Abstract

In this paper, we formalize an operational semantics of a weak memory model similar to the one in Go, concentrating on buffered writes, i.e., ignoring the possibility of buffered reads.

keywords: operational semantics, goroutines, weak memory model, write buffering, partial store ordering.

1 Introduction

Concurrency is central to modern hardware platforms and programming languages. The usefulness of independent simultaneity of computations is limited, as many interesting problems require *communication* and *synchronization*. Very generally, communication relates to the transfer of data between computational units, while synchronization relates to curbing interleavings on computations and limiting observable data values. One fundamental mode of communication is *shared variable concurrency,* where the value written by a thread[1] to a shared variable may be observed by another thread that reads from that same shared variable. The exact values and the circumstances under which these values can and cannot be observed are laid down in a *memory model*. The memory model is specific for a given platform, language, or a combination of the two. In one of the simplest memory models, shared memory is interpreted as 1) a shared global repository mapping variables or locations to values where 2) the read and write actions of the individual threads appear to be effected atomically in some global total order (*i.e.*, as one interleaving of global read and write events). This corresponds to Lamport's notion of *sequential consistency* [14]; it is also sometimes known as the von Neumann model and corresponds to Single Instruction stream, Single Data stream (SISD) in

[1]We use the words processes and threads interchangeably when referring to units of concurrency. In the formalization later, we introduce *goroutines*, a neologism use in Go and a play on words on the notion of *coroutine*. Conceptually, like processes or threads, goroutines are an asynchronously executing unit of concurrency.

Flynn's classification [8]. Though sequential consistency is conceptually simple and has become a baseline for other memory models, it is much too restrictive: it does not reflect current hardware and enforcing it in software would preclude many established compiler code optimizations. Consequently, numerous relaxations to the basic interleaving model of sequential consistency have been proposed, investigated, and implemented. For a tutorial on various memory models, see Adve and Gharachorloo [1].

The design of a proper memory model is a balancing act. Rigor, conciseness, and intelligibility are undoubtedly desirable attributes. It is less clear, however, how to design a model that captures a variety of different concrete implementations without committing too narrowly to one particular solution or technology. On one hand, the model should be lax enough to allow common current optimization and potential future ones. On the other, the more lax the model, the greater the chances it allows for unintuitive behavior, which in turn increases the burden on developers. Unfortunately, a comprehensive and generally accepted "universal weak memory model" is not in sight. Neither there exists an uncontroversial comprehensive specification of the C^{++}11 [4][5] or Java memory models [15] [20]. Though the right balance between relaxation and intelligibility is up for debate, models should, in principle, preclude definitely unwanted behavior. One class of unwanted behavior is called the *out-of-thin-air* behavior [6]. Intuitively, these are results that can be justified via some sort of circular reasoning. According to Pichon-Pharabod and Sewell [19], however, there is not even an exact, generally accepted definition of out-of-thin-air behavior, and even doubts have been cast upon a general style of defining weak memory models. For example, Batty et al. [3] 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. 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. According to Batty et al., the problem is that there exist different programs in the C/C^{++}11-semantics with the *same* candidate executions, yet their resulting execution is deemed acceptable for some programs and unacceptable for others.

There exist several approaches to *formalizing* a memory model. One widely followed approach, called *axiomatic* semantics,[2] describes the semantics of a set of parallel threads as a number of memory events with various relations between them — the shortly mentioned per-candicate execution approach falls into this camp. Depending on the memory model, different classes of relations are taken into account. The most straightforward relation is known as *program order*, which, as the name suggests, reflects the order in which instructions appear within the program. Besides this basic relation, various others may enter the picture depending on the choice of the memory model and the selection of instructions onto which

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

it has bearing. Typically, a model takes into account various synchronization operations like locks, different forms of fences, atomic memory accesses, and also channel sends and receives. Often the program order and an appropriate combination of other relations (like a "synchronize-with" relation) are combined into one *happens-before* relation [13]. This relation captures a form of causality between events. Lamport introduced the *happened-before* relation not in the context of shared variable concurrency but in a message the passing setting [13]. In that context, the relation captured the union of program-order and a form of causality between channel sends and receive, requiring a channel send event to happen-before its corresponding receive.

The happens-before relation is a technical vehicle to define the semantics of memory models. It is important to note that just because an instruction is in a *happens-before* relation with a second one, it does not necessarily mean that the first instruction *actually* "happened" before the second. 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 committed (*i.e.*, its effect becomes observable) before the second.[3] For instance, a compiler might choose to change the order of the two instructions. The processor may also rearrange memory instructions so that their effect may not be visible in program order. To avoid the confusion between the technical axiomatic 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 as opposed to its natural language interpretation. Similarly for "happens-after."

This paper presents an *operational* semantics of a weak memory model that bypasses the out-of-thin-air problem. It utilizes the happens-before relation when reasoning about possible read values. The ideas here are a rework of the Valle's master thesis [23]. Apart from changes of representation and notation, the paper covers channel creation/disposal and non-deterministic choices. This brings us a step closer in the direction of a realistic formalization of the memory model for the Go programming language. Our goal for future work is to further relax the model, yet, at the same time, keeping out-of-thin-air behavior at bay.

The remainder of the paper is organized as follows. Section 2 discusses aspects of weak memory models (especially of Go's memory model) that are relevant in the context of this paper. Section 3 introduces the syntax of the calculus, concentrating on goroutines, shared-memory interaction, and channel communication. Section 4 represents the operational semantics of the calculus. The concluding Section 5 briefly discusses future work.

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

2 Background

The Go language [9] [7], supported by Google, 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 [11] or Occam [12]. The happens-before relation is used in the Go memory

Listing (1) Erroneous synchronization

```
1  var a string
2  var done bool
3
4  func setup() {
5          a = "hello, world"
6          done = true
7  }
8
9  func main() {
10         go setup()
11         for !done { }
12         print(a)
13 }
```

Listing (2) Channel synchronization

```
1  var a string
2  var c = make(chan int, 10)
3
4  func setup() {
5          a = "hello, world"
6          c <- 0
7  }
8
9  func main() {
10         go setup()
11         <-c
12         print(a)
13 }
```

Figure 1: Example of shared variable and channel communication [10]

model to describe which reads can observe which writes to the same variable [10]. For example:

Rule 1 (Single Thread) Within a single goroutine, the happens-before relation boils down to program order. In other words, within a single goroutine, reads and writes behave as if they executed in the order specified by the program.

Consider the code snippet of Listing 1 where a `main` function calls a `setup` function. The keyword `go` is used to specify that the call to `setup` should execute asynchronously (one can think of `setup` running on its own thread). Rule 1 tells us that, within `setup`, the write to variable `a` happens-before the write to `done`. The Go memory model is, however, a relaxed memory model, and "the execution order observed by one goroutine may differ from the order perceived by another" [10]. Therefore, the `main` function may observe a different ordering of the assignments to `a` and `done` because read and write events in the asynchronous call to `setup` are not related to the remaining reads and writes in `main`. As a result, the print of `a` on line 12 of Listing 1 may print an uninitialized value as opposed to "hello world."

If the effects of a goroutine are to be observed by another, a synchronization mechanism must be used in order to establish a relative ordering between events belonging to the different goroutines. The Go memory model advocates channel communication as the main method of synchronization. Rule 2 describes how sends and receives on a channel establish an ordering between events across different goroutines [10].

Rule 2 (Channel) A send on a channel happens before the corresponding receive from that channel completes.

In the example of Figure 1, the channel Rule 2 can be used to establish a relation between the write to variable `a` by the asynchronous `setup` and the use of `a` in `main`. Since a read on an empty channel blocks, `setup`'s send of 0 on channel `c` will happen-before `main`'s read of channel `c` completes, in other words, `c<-0` $\sqsubset$ `<-c` where $\sqsubset$ represents the happens-before relation. We can now conclude that the print in Listing 2, which uses channel communication for synchronization, will indeed print "hello world" as follows:

$$\begin{array}{rr}
\texttt{a=''hello world''} \sqsubset \texttt{c <- 0} & \text{(by rule 1)} \\
\texttt{c<-0} \sqsubset \texttt{<-c} & \text{(by rule 2)} \\
\texttt{<-c} \sqsubset \texttt{print(a)} & \text{(by rule 1)} \\
\texttt{a=''hello world''} \sqsubset \texttt{print(a)} & \text{(by transitivity of } \sqsubset\text{)}
\end{array}$$

Finally, rule 3 helps us reason about bounded channels.

Rule 3 (Channel capacity) The k^{th} receive on a channel with capacity C happens before the $(k+C)^{th}$ send from that channel completes.

The operational semantics described on Section 4 mirrors rules 2 and 3 by accounting for bounded channel creation (R-MAKE), disposal (R-CLOSE), and sends and receives on channels (R-SEND and R-RECEIVE). The operational semantics also respects rule 1, meaning that, from a goroutine's point of view, its reads and writes happen in program order. This is done by keeping track of values of previous writes that should no longer be visible from the goroutine's read perspective. We call writes that are no longer observable as shadowed writes. A write of value v to variable z, represented as $n(\!|z{:=}v|\!)$ where n is a unique identifier, is shadowed by another write n' if n and n' write to the same variable and $n \sqsubset n'$.

In order to support the behavior like that of Figure 1, where the write to variable `a` on line 5 is perceived as happening after the write to `done` on line 6, one needs to let go of sequential consistency. As shown in the hierarchy of Figure 2, write delays are a step away from Sequential Consistency (SC) and in the direction of increasing relaxation. The *Total Store Order* (TSO) model is a well known model that "allows a read to return the value of its own processor's write even before the write is serialized with respect to other writes to the same location" [1]. *Partial Store Ordering* (PSO) is a further relaxation from TSO that allows writes to different memory location to appear out of order [24]. In addition to delaying the effect of writes, one can obtain greater relaxation by also delaying the effects of reads. The Go memory model falls in the delayed read/write category.

The proposed memory model of Section 4 is relaxed enough to allow for delayed writes as in a partial store order model. Writes are delayed by being placed on a global pool; a subsequent read can read any previous write from the pool as

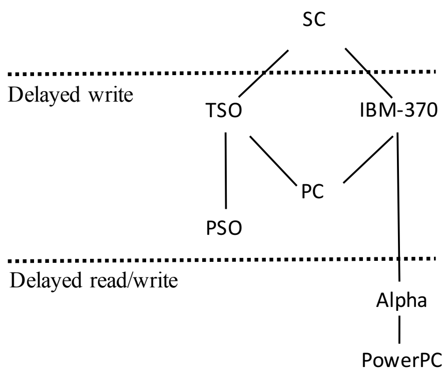

Figure 2: Hierarchy of memory models

long as the write is not considered shadowed from the reader's perspective. In future work we plan to relax reads with the goal of obtaining an operational definition of a delayed read/write memory model that is even closer to the Go memory model.

3 Abstract syntax

The abstract syntax of the calculus is given in Table 1. *Values* are written generally as v. Note that local variables (or registers) r counts among v. Additionally, names or references n are values, representing here channel names. These are dynamically created and therefore are *run-time* system (thus represented underlined as $\underline{n}$ in the grammar). Later, we often use c specifically for references to channels. We do not explicitly list values such as the unit value, booleans, integers etc. further. Expressions includes local variables r.

We omit introducing compound local expressions like $r_1 + r_2$, which would be straightforward to add. Shared variables are denoted by x, z etc, and $\texttt{load}\, z$ represents the reading the shared variable z into the thread. 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. Especially, expressions like $x \leftarrow \texttt{load}\, z$ or $x \leftarrow z$ are disallowed. That way the languages obeys a form of at-most-once restriction [2], 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. The expression $\texttt{pend}\, v$ represents the state immediately after sending a value over a channel. Note that `pend` is part of the *run-time* syntax as opposed to the user-level syntax, i.e., it is used to formulate the operational semantics of the language. Starting a new asynchronous activity, called goroutine in Go, is done using the `go`-keyword. Guards g (also called communication cases in the Go language specification [9]) are specific expressions used in combination with the select-statements. In Go, the `go`-statement is applied to function calls only. We omit here formalizing function calls, asynchronous or otherwise, as orthogonal to the issues at hand. See [22] for an operational semantics dealing with goroutines and closures.

For the branches of select statements, only communication statements, i.e., channel sending and receiving, or the `default`-keyword are allowed as guards (in Go, general expressions are allowed). The restriction imposed in the calculus is in line with the A-normal form representation and does not impose any actual restriction in expressivity. The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches (called communication clauses in the specification [9]), which are guarded threads. For each select-statement, there is at most one branch guarded by `default`. It is allowed that a channel is mentioned in more than one guard. Also "mixed choices" [17] [18] 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 goroutine that is permanently blocked. The `stop`-thread is also the only way to syntactically "terminate" a thread, i.e., it's 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.

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

Table 1: Abstract syntax

Remark 1 (Select statements and side effects) In line with the design of the abstract syntax, the guards are side-effect free. Go's concrete syntax does allow side effects in the guards, no matter how dubious such practice may seem. The side effects may include sending and receiving of channels itself. Evaluation of guards is phased: in the first phase, each guard is evaluated *exactly once* and they are evaluated in *source order*. In the second phase, one of the enabled branches is taken, if any; otherwise the default branch is taken, if present. This has the effect of disallowing side-effects and making the evaluation order explicit in the A-normal form style of syntax [21]. □

4 Operational semantics with write buffering

In this section we define the operational semantics of the calculus. The semantics delays the write to main memory, thereby realizing a form of write buffering that roughly corresponds to the well-known notion of *partial store order* (PSO) (see Section 2 for a discussion). PSO is a generalization of *total store ordering* or TSO. While both assume that writes to the memory are buffered, PSO only imposes order on a *per memory location* basis [24]. Therefore, writes to different location are not necessarily executed in FIFO order.

Before giving the operational rules in Section 4.2, we fix the run-time configurations of a program. Besides goroutines running concurrently, the configuration will contain "asynchronous writes" to shared variables in particular.

4.1 Local states, events, and configurations

Let N represent an infinite set of names or identifiers with typical elements n, n'_2. As mentioned earlier, for readability, we will use names like c, c_1 for channels. X represents the set of variables. A run-time configuration or program is given by the following syntax:

$$P ::= \langle \sigma, t \rangle \;\mid\; n(\!|z{:=}v|\!) \;\mid\; \bullet \;\mid\; P \parallel P \;\mid\; \nu n\, P\,. \tag{1}$$

Programs consist of the parallel composition of goroutines $\langle \sigma, t \rangle$ and write events; $\bullet$ represents the empty configuration. The ν-binder, as known from the π-calculus [16] indicates scoping. Later, we'll introduce *channels* as part of the run-time configurations and, besides write events, also read-events.

Definition 1 (Write events) *Write events* $n(\!|z{:=}v|\!)$ are 3-tuples from $N \times X \times \mathit{Val}$.

A write event simply records the shared variable being written to and the written value, together with a unique identifier n. In the current semantics, unlike writes, read accesses to the main memory cannot be delayed. Consequently, there are no read events.

In addition to the code t to be executed, goroutines $\langle \sigma, t \rangle$ contain local information about earlier memory interaction (cf. equation (1)). We call this the goroutine's *local state* σ. In first approximation, the local state contains information about events which occurred earlier. Local states are tuples of the form (E_{hb}, E_s); the first component E_{hb} contains the identities of all write events that have happened before, at the current stage of the computation of the goroutine. To help in bookkeeping, the set E_{hb} does not just remembers the identities of the write events but pairs (n, z), thus remembering the variable written to in the event n as well.

That makes E_{hb} a set of type $2^{N \times X}$. The second component E_s of the local state represents the set of identities of write events that, at the current point, are shadowed (i.e., writes to variables for which another write event had happened after the shadowed one but before the current point in time).

Definition 2 (Local state) *Local states* σ are are tuples 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 $\{n \mid (n,z) \in E_{hb}\}$. Let furthermore $E^w_{hb} = \{n \mid (n,z) \in E_{hb}\}$ represent the set of names of write events from E_{hb}. For a local state $\sigma = (E_{hb}, E_s)$, let σ^w stand for E^w_{hb}. We write $\sigma_\emptyset$ for the local state $(\emptyset, \emptyset)$, containing neither happens-before nor shadow information.

4.2 Reduction steps

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

4.2.1 Local steps

The local steps are given modulo structural congruence $\equiv$ on configurations. The congruence rules are standard and given in Table 2. Besides specifying parallel composition as binary operator of 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 [16] of Milner et al..

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

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 $\rightsquigarrow$.

Local steps (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.

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

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

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)

4.2.2 Shared variable interaction

Table 5 contains the transitions for the basic interactions with main memory, i.e., read and write steps.

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

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

Table 5: Operational semantics: Reads and writes

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 in rule R-WRITE simply records the corresponding event $n(\!|z{:=}v|\!)$ in the global configuration, where n is freshly generated in the write step. The write events are remembered without keeping track of the order of their issuance. In other words: as far as the global configuration is concerned, no write event ever invalidates an "earlier" write event resp. overwrites a previous value in a shared variable. This results in an out-of-order execution of memory instructions. Thus, the global configuration accumulates the "positive" information about all available write events which potentially can be observed by reading from shared memory.

Values which never have been written at all cannot be observed. Whereas the global configuration remembers all write events indefinitely, filtering out writes which are *no longer* observable is handled thread-locally. In other words, which writes are observable depends on the local perspective of the threads, and having different versions of write events available in the global configuration leads to a *multi-copy* interpretation of shared variables.

The local perspective regarding which events are observable and which are not is represented by the *local* state σ of a goroutine. In a conventional setting with a "strong memory model", a local state of a thread contains "*positive*" information mapping each variable to its current value, or perhaps more generally, to a set of possible values. Here, the local state contains information about which write events (or later also read events) have happened before. But the local information about the history of prior events is primarily used in the form of *negative* information: which observations *cannot* be made. A read can observe all write events *except* for those shadowed. So, issuing a write command in 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 accordingly.

Compared to writing, the treatment of reading from main memory is simpler (cf. rule R-READ) because reading is not delayed. It's simply checked whether there exists a write event to the intended variable, whose identity is not blacklisted in the E_s-set of the thread executing the `load`. So, loading a value from shared memory into local memory can take the value of *any* previous write event to the variable in question, *unless* the variable is shadowed for the goroutine issuing the read-command (see rule R-READ). Perhaps counter-intuitively, that has the following consequence: if a goroutine reads the same shared variable repeatedly, observing a certain value once does not imply that the same value is read next time (even if no new writes are issued to the shared memory). This is because all subsequent readings of the variable are independent and non-deterministically chosen from the set of write events which are not yet shadowed. This reflects that the memory model allows out-of-order reading (and writing) of shared variables.

Remark 2 (Local representation of E_{hb}) With access to the global configuration containing all global write events, there would be no need to store in E_{hb} the variable z given that the name n uniquely identifies the global event. To arrive at a local formulation of the reduction rules without need to refer to the totality of the global configuration, the thread-local set E_{hb} records the variable names of the write events (cf. Definition 2). It is therefore a global invariant that for each element (n,z) in the local state of a thread, there exists exactly one global write event $n(\!|z{:=}v|\!)$ for some value v. Note in passing that the inverse does not hold: For each global write event $n(\!|z{:=}v|\!)$, there may be *more* than one local representative (n,z) in the E_{hb}-sets insofar that the knowledge that this event happened-before may be available to more than one thread. Propagation of this happens-before knowledge is done via synchronization statements, discussed later. Given an event $n(\!|z{:=}v|\!)$

present in the global configuration, there is at least *one* thread, though, which is aware of that fact, i.e., one thread which contains (n, z) in its E_{hb}-set. This thread is the one that issued the write event, and it reflects the fact that the so-called *program order* is part of the happens-before order. □

4.2.3 Channel communication

Channels in Go are the primary mechanism for communication and synchronization. They are typed and assure FIFO communication from a sender to a receiver sharing 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 also closures, including closures involving higher-order functions. Channels can be dynamically created and be closed again. Channels are *bounded,* i.e., each channel has a finite, fixed capacity. Channels of capacity 0 are called *synchronous.*

Our semantics 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. We incorporate channels into the global configurations, i.e., the configurations P from equation (1) are extended by channels:

Definition 3 (Channels) A channel is of the form $c[q_1, q_2]$, where c is a name and (q_1, q_2) a pair of queues. The first queue, q_1, contains elements of type $(Val \times \Sigma) + \{\perp\}$; the second, q_2, contains elements of type $\Sigma + \{\perp\}$, where $\perp$ is a distinct, separate value corresponding to the "end-of-transmission." The first and the second 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, but they are both considered named by c (not by c_f or c_b). We write $[]$ for an empty queue, $a :: q$ for a queue with a as the element most recently added into q, and $q :: a$ for the queue where a 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.

Channels can be closed, after which no new values can be sent, otherwise a panic ensues (panics are a form of exception in Go). Values which are "on transit" in a channel when it's being closed are *not* discarded and can be received as normal. After the last sent value has been received from a closed channel, it's 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, even though 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.

As mentioned, without the possibility of *synchronization*, the model based only on loads and stores is intolerably weak. The primary means of synchronization in Go is via channel communication. In the operational semantics, this is captured in that sending and receiving of values additionally exchanges happened-before and shadowing knowledge between the communicating partners. Considering the fact that the role of the "positive" happened-before information is auxiliary to adequately track the local shadowing information, *synchronization* in this (and similar) models performs the following dual tasks. First, it *restricts the enabling* of certain commands, meaning, unlike reads and writes, sending and receiving operations over a channel are not uniformly enabled. Second, it *restricts the observability* of certain write events. The first role, therefore, is to narrow the choices of commands which can be executed; this narrows the set of possible "interleavings" of steps. The second role is to narrow the set of possible *data values* being readable.

Creating a channel is covered by R-MAKE and does not involve synchronization. The forward channel is initially empty but the backward part is not: it is initialized by a queue of length $cap(c)$, which corresponds to the capacity of the channel. The element of the queue contain *no* happens-before or shadow information (represented by $\sigma_\emptyset$).

Remark 3 (Initial state of a channel) After creation of a channel of capacity k, the backward part of the channel contains k elements $\sigma_\emptyset$. The backward channels are used to realize the synchronization in connection with the *boundedness* of channels. In particular, to realize the stipulation of Go's memory model that the i^{th} receive on a channel with capacity k happens before the completion of the $i+k^{th}$ send to that channel (see rule 3 of the Background section). Assuming that channel sends and receives are indexed starting with 0, the first k sends to a new channel then complete after the "sends" $-k, -(k-1), \ldots -1$ occur. That's an empty requirement, as there are of course no sends with negative indices. For uniformity of the semantics, the back-channel is therefore filled initially with k "dummy values" $\sigma_\emptyset$. These dummies contains no happens-before information and, therefore, acts as unity with respect the $+$-operation on location states, i.e., $\sigma + \sigma_\emptyset = \sigma$. Cf. also rule R-PEND. □

Sending a value over a channel c does not change the local state (cf. rule R-SEND). Besides that, sending can be done only on a channel which is not yet closed. In the post-configuration, the goroutine enters a "pending" state, denoted by the run-time syntax `pend`. The pending state is used to realize the two-way handshake communication though which the happens-before information is exchanged.

For receiving a value from a channel (cf. rule R-RECEIVE), the communication channel must be non-empty. 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 sent information. Furthermore, the state of the receiver before the update is sent back via the backward channel.

Executing a receive on a *closed* channel results in receiving the end-of-transmission marker $\perp$ (cf. rule R-RECEIVE$_\perp$) and updating the local state σ in the same way as when receiving a properly sent value. The "value" $\perp$ is not removed from the queue, so that all clients attempting to receive from the closed channel obtain the communicated happens-before synchronization information. Furthermore, there is no need to communicate happens-before constraints from the receiver to a potential future sender on the 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, the attempt would raise a panic. Here, this is captured by the absence of enabled transitions.

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

$$\frac{\neg closed(c_f[q])}{\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q] \to \langle \sigma, \texttt{pend}\ c; t\rangle \parallel c_f[(v,\sigma) :: q]}\ \text{R-SEND}$$

$$\frac{\sigma' = \sigma + \sigma''}{c_b[q_2 :: \sigma''] \parallel \langle \sigma, \texttt{pend}\ c; t\rangle \parallel c_f[q_1] \to c_b[q_2] \parallel \langle \sigma', t\rangle \parallel c_f[q_1]}\ \text{R-PEND}$$

$$\frac{\sigma' = \sigma + \sigma'' \qquad v \neq \perp}{c_f[q_1 :: (v,\sigma'')] \parallel \langle \sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \parallel c_b[q_2] \to c_f[q_1] \parallel \langle \sigma', \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel c_b[\sigma :: q_2]}\ \text{R-RECEIVE}$$

$$\frac{\sigma' = \sigma + \sigma''}{c_f[(\perp,\sigma'')] \parallel \langle \sigma, \texttt{let}\ x = \leftarrow c\ \texttt{in}\ t\rangle \to c_f[(\perp,\sigma'')] \parallel \langle \sigma', \texttt{let}\ x = \perp\ \texttt{in}\ t\rangle}\ \text{R-RECEIVE}_\perp$$

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

Table 6: Operational semantics: message passing

Note that R-SEND is enabled independent from the capacity of the channel. That makes the forward queue an unbounded data structure (likewise for the backwards queue). Of course, executing a send does not imply the sending process can proceed unconditionally; the sender needs a value (containing the happens-before information) from the backward queue, and the sender blocks if this value is not yet available. As far as the sizes of the queues of a channel in connection with the channel's capacity are concerned, the semantics assures the follow invariant.

Lemma 4.1 (Invariant for channel queues) *Let c be a non-closed channel created with capacity k. Let furthermore p be the number of goroutines pending on c in a configuration, i.e., goroutines where rule* R-PEND *is enabled with* `pend` *c as next step. Then the following global invariant holds,*

$$|q_f| + |q_b| - p = k \ . \tag{2}$$

Proof: By straightforward induction on the steps of the operational semantics. The invariant holds initially upon creation of a channel (cf. rule R-MAKE), and rules R-SEND, R-PEND, and R-RECEIVE preserve it. The invariant is stated only for non-closed channels; thus R-RECEIVE$_\perp$ (which applies only to channels previously closed) and R-CLOSE are both covered. The rest of the rules don't change the state of any channel nor change the number of pending thread. □

In the invariant from equation (2), the size of the queues are actually not bounded by the capacity; they are bounded by the number of goroutines instead. Therefore, given that goroutines can be created dynamically, we cannot, in general, put a cap on the size of the queues.

The "coordination" between senders and receivers on a channel is captured by the invariants of Lemma 4.2. In particular, equation (4) captures the boundedness of channels.

Lemma 4.2 (Invariant for channel communication) *Assume a non-closed channel c with capacity k. Let s_1 denote the number of applications of* R-SEND *to the channel, s_2 the number for* R-PEND*, and r the number of applications of* R-RECEIVE*. The semantics maintains the following invariants.*

$$\begin{align} s_1 &\geq r \tag{3} \\ s_2 &\leq k + r \ . \tag{4} \end{align}$$

Proof: By induction on the steps of the operational semantics. □

Remark 4 (Receiving from a channel) In the formulation of rule R-RECEIVE, the value of the local state *sent back* is σ, i.e., the state *before* doing the receive-step. In particular, it is not σ' after receiving σ''. For a synchronous channel, i.e., a channel of capacity 0, this would not make a difference as the information is "bounced back" to the original sender (therefore taking σ'' into account would not add any "new" information to that particular goroutine). The treatment as in rule R-RECEIVE corresponds to the formulation in the Go memory model [10], which postulates that sending on a channel happens-before the corresponding receive on the channel *completes* (see rule 2 of the Background section). □

Remark 5 (Channel capacity) Channels in Go are of bounded capacity. The rules of semantics do not impose such a restriction explicitly, meaning, the corresponding rules R-SEND, R-RECEIVE, and R-PEND do not impose an upper bound on the queues in the forward resp. backward channels $c_f[q_1]$ and $c_b[q_2]$. Indeed, given an unbounded number of threads, there is *no* upper bound on the number of elements in a forward queue; analogously for the backward queue. One could change the rule R-SEND by adding the premise:

$$|q_1| \leq cap(c) \tag{5}$$

where q_1 is the forward queue of the involved channel and $cap(c)$ its capacity. Analogously, rule R-RECEIVE could be changed by adding as premise

$$|q_2| \leq cap(c). \tag{6}$$

Adding these premises, however, would cause no observable difference in behavior. The only impact would be that the number of elements actually stored in the channels, both the forward as well as the backward channel, would be bounded and the maximal number of elements in each queue would be $cap(c)+1$.

In the semantics from Table 6, the boundedness of communication is nonetheless *implicitly* assured by the fact that each completed communication of a value from a sender to receiver consists of a kind of two-way handshake, in that the sender cannot proceed until it has received some data from the backward channel. In the semantics here, the data in the backward channel consists of happens-before information used to update the local information of the sender thread.

The picture of two-way handshake communication is completely fitting only for *synchronous* channels. There, the sender cannot proceed until it has received the acknowledgment back from the receiver processes. For channels with a higher capacity, the happens-before information from the back channel *does not* need to originate from the thread that fielded the value in that send-receive pair of communication.

The conceptual boundedness of the communication is assured then by the fact that the channel is initially created with only a finite number of elements, namely $\sigma_\emptyset$, which regulate the possibility to proceed for the senders and receiver and thus couple their relative speed. The semantics maintains the invariants from Lemma 4.2. In particular, equation (3) corresponds to the *causality* axiom of the memory model, that the send happens-before the corresponding receive. The second invariant (4) reflects the boundedness of the channels. For a synchronous channel with $k = 0$, in particular, it means each completed send has to be preceded by a corresponding reception. □

5 Conclusion

In this paper, we presented an operational semantics in a concurrent setting with write buffering and message passing with bounded buffers. The semantics can be seen as an operational representation of a memory model inspired by the Go memory model, based in a happens-before relation, but concentrating on write-buffering. The semantics here therefore resembles a memory model with *total store order* (*TSO*) [24]. In future work, we plan to further relax the model towards obtaining a definition of a delayed read/write memory model that is yet closer to Go.

References

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

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

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

[10] Go memory model (2016). The Go memory model. `https://golang.org/ref/mem`.

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

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

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

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

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

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

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

[18] 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.

[19] 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.

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

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

[22] Steffen, M. (2015). 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.

[23] Valle, S. (2016). Shared variables in Go. a semantic analysis of the Go memory model. Master's thesis, Faculty of Mathematics and Natural Sciences, University of Oslo.

[24] Weaver, D. and Germond, T. (1994). *The SPARC Architecture Manual (version 9)*. Prentice Hall.

Index