Anything goes unless forbidden

Notes on synchronization and the operational semantics of a relaxed memory model

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 *memory model* dictates which values may be observed when reading from memory, thereby regulating how concurrent processes communicate through shared memory.
In this note, we discuss a weak memory model for a calculus inspired by the Go programming language, focusing on buffered channel communication as the sole synchronization primitive. In contrast to an axiomatic semantics, we present an operational interpretation and discuss its rationale and its design.

1 Introduction

A *memory model* dictates which values may be observed when reading from memory, thereby affecting how concurrent processes communicate through shared memory. In this paper we discuss ideas behind an operational formalization of a memory model that mixes message passing synchronization with shared-variable communication. The formal model is inspired by the informal specification [14] of the Go programming language's [13, 9] memory model. In the following, we sketch the principles underlying our design choices. Up to recently, when it comes to memory models, axiomatic semantics have arguably received more focus than operational ones. We chose the *operational* approach and believe it leads to more intuitive interpretation of the model's guarantees. The semantics takes a perhaps unusual view on synchronization, stressing the following dichotomy between synchronization and communication:

> communication makes information observable, synchronization makes it *unobservable.*

Synchronization as constraint on observations Memory models regulate the interaction of multiple threads using shared memory. Often, this is formulated in an *observational* manner. It is not the realization of the memory that matters, it is what/which values can be encountered by threads interacting with memory. Interactions, in their most basic form, are elementary read and writes or loads and stores, while "observing" a value by a thread means reading it from memory. One general reason which complicates the situation is the asynchronous nature of interaction. For example, threads performing a read operation may not instantly observe values, as in the case of speculative execution. Similarly, writes done by a process do not immediately and not necessarily

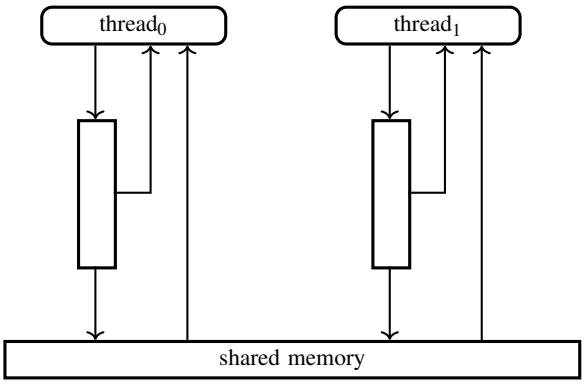


Fig. 1: Write buffers between processors and memory.

atomically change the status of a shared variable. When it comes to writes, this delay can be understood by the presence of write buffers, shown in Figure 1.

The picture is meant to only give a schematic impression. In hardware memory models, the non-instantaneous effect of writes is caused by complex interaction in the memory hierarchy. One complicating factor is the fact that memory, unlike in Figure 1, might not consist of a uniform global "data store." Instead, multiple copies of a shared variable may exist. Thus, it may not be appropriate to interpret a read-access to a variable as observing *the* variable's value, as there might be more than one copy. Different threads may read from different copies and observe different values.

Synchronization is what allows us to program on such slippery grounds. In very broad terms, synchronization between processes can be understood as *restricting* possible interleavings. Processes running independently in parallel, without synchronization, perform their respective steps unconstrained and undisturbed by each other. Synchronization means disallowing some of those interleavings, for example, blocking the progress of one process up until the other process has arrived at a certain point or produced some result. This interpretation casts synchronization as constraining the control flow of concurrently executing processes.

Communication, the exchange of data between processes, also needs consideration. For weak memory models and shared variable communication, synchronization is often "explained" as guaranteeing that data is available, thereby assisting communication. Synchronization statement like fences may understood as "flushing the write buffer." For example, with the execution of a write operation in Figure 1, a value is placed in the write buffers. Synchronization is then needed to make sure that the value becomes observable by other threads. See, for instance, a statement from an ARM programmer's guide talking about that a data memory barrier (DMB).[3]

> "... forces all earlier-in-program-order memory accesses to *become globally visible* before any subsequent accesses."

[3] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CJAIAJFI.html`

This view of synchronization as a mechanism for making writes *observable* is perfectly legitimate, of course, and may reflect the way hardware realizations of memory systems actually operate. Our semantics takes a *different view,* though. In line with seeing synchronization as restricting the control flow, our semantics formalizes synchronization as follows

a) for control: it prevents a process from proceeding (temporarily);
b) for data: it makes written values *unobservable*.

The formalization highlights a dichotomy between *communication* and *synchronization* and makes a sharp distinction of responsibilities: communication makes data available, while synchronization makes data *unavailable.* In these terms, reading and writing to memory is about communication only, not synchronization. Note that such interpretation is also implicit in the informal description of the happens-before memory model of Go [14], which takes a very liberal standpoint: any written value is observable unless and until it is made unobservable by synchronization. Such standpoint is what leads also to a very relaxed memory model.

Basically, the *only* mechanism with synchronizing power is *channel communication.* There are no fences, synchronized or volatile variables, locks,[4] semaphores, etc. Channels are thereby a mechanism used both for communication of data as well as synchronization. On the one hand, a channel transmits "positive" information, namely the data being relayed from sender to receiver. On the other, channels transmit "*negative*" information related to synchronization. Receiving is blocking, as usual; it prevents the receiver to proceed until a value is available to be received. Blocking covers the control-related part of synchronization. The other part, which has to do with shared variables and their observable values, is also affected by synchronization via channel communication.

Sequential consistency as baseline One of the simplest memory models, called *sequentially consistent*, stipulates that operations must appear to execute one at a time and in program order [17]. SC was one of the first formalizations and, to this day, constitutes a baseline for well-behaved memory. For efficiency reasons, however, modern hardware does not guarantee sequential consistency. SC is also considered much too strong to serve as the underlying memory semantics of programming languages; the reason being that sequential consistency prevents many established compiler optimizations and robs from the compiler writer the chance to exploit the underlying hardware for efficient parallel execution. The research community has struggled to agree on what, exactly, a proper memory model should offer. Consequently, a bewildering array of *weak* or *relaxed memory models* have been proposed, investigated, and implemented. Different taxonomies and catalogs of so-called *litmus tests*, which highlight specific aspects of memory models, have also been researched [1].

In light of these difficulties and despite many attempts, there does not exist a well-accepted comprehensive specification of the C++11 [5, 6] or Java memory models [4, 18, 20]. Despite the existence of broadly disparate approaches towards memory and

[4] Except that locks are available via libraries.

its formalization, the following principle of relaxed memory has gained overwhelming acceptance: regardless of how much relaxation is permitted by the memory model, if a program is *data-race free* or *properly synchronized*, then the memory model must ensure that the program behaves sequentially consistently [2, 18]. This principle is known as the *SC-DRF* guarantee.

Operational semantics We sketch an *operational* semantics for a weak memory. Our calculus is inspired by the Go programming language: similar to Go, our model focuses on channel communication as the main synchronization primitive. Go's memory model, however, is described, albeit succinctly and precisely, in prose [14]. We provide a formal semantics instead. In this paper, we informally discuss ideas underlying the semantics and possible extensions. We refer the reader to [11] and the technical report [12] for more details.

2 Background

Go's memory model Concerning synchronization primitives, the model covers goroutine creation and destruction, channel communication, locks, and the `once`-statement. Our semantics will concentrate on thread creation and channel communication because lock-handling and the `once` statement are *not* language primitives but part of the `sync`-library. Thread destruction, i.e. termination, comes with *no* guarantees concerning visibility: it involves no synchronization and thus the semantics does not treat thread termination in any special way. In that sense, our semantics treats all of the *primitives* covered by Go's memory model specification. As will become clear in the next sections, our semantics does not, however, relax read events. Therefore, our memory model is stronger than Go's. On the plus side, this prevents a class of undesirable behavior called *out-of-thin-air* [7]. On the negative, the absence of relaxed reads comes at the expense of some forms of compiler optimizations.

Happens-before relation and observability Like Java's [18, 20], C++11's [5, 6], and many other memory models, ours centers around the definition of a *happens-before* relation. The concept dates back to [16] and was introduced in a pure *message-passing* setting, i.e., without shared variables.[5] The relation is a technical vehicle for defining the semantics of memory models. It is important to note that just because an instruction or event is in a *happens-before* relation with a second one, it does not necessarily mean that the first instruction *actually* "happens" before the second in the operational semantics. Consider the sequence of assignments $x := 1; y := 2$ as an example. The first assignment "happens-before" the second as they are in program order, but it does not mean the first instruction is actually "done" before the second,[6] 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. Alternatively, a processor may rearrange memory instructions so that their effect may not

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

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

be visible in program order. Conversely, the fact that two events happen to occur one after the other in a particular schedule does not imply that they are in happens-before relationship, as the observed order may have been coincidental. To avoid confusion between the technical happens-before relation and our understanding of what happens when the programs runs, we speak of event e_1 "happens-before" e_2 in reference to the technical definition (also abbreviated as $e_1 \rightarrow_{\mathsf{hb}} e_2$ as opposed to its natural language interpretation. Also, when speaking about steps and events in the operational semantics, we avoid talking about something happening before something else, and rather say that a step or transition "occurs" in a particular order.

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

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

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

For the sake of discussion, let us concentrate on the following two constituents for the happens-before relation: 1) *program order* and 2) the order stemming from channel communication.[7] According to the Go memory model [14], we have the following constraints related to a channel c with capacity k:

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

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

Condition (4) accounts for the boundedness of channels by transmitting happens-before information in the backward direction for some receiver to some sender. Note that for *synchronous* channels, which have capacity zero, conditions (3) and (4) degenerate: the send and receiving threads participate in the rendezvous and symmetrically exchange their happens-before information.

In summary, the operational semantics captures the following principles:

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

Delayed negative information: in contrast, negative information overwriting previously observable *writes* is *not* immediately effective. 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 [8].

3 Axiomatic semantics and litmus tests

To position the work in a slightly broader context, we revisit notions from axiomatic semantics of memory models. In particular, we use well-known litmus tests to highlight similarities and differences between our semantics and alternative ones. In the

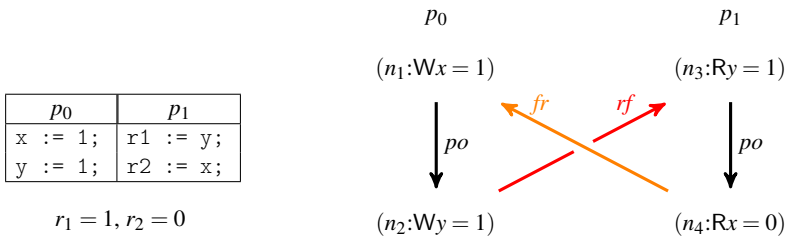
[7] There are additional conditions in connection with channel creation and thread creation, the latter basically a generalization of program order; we ignore it in the discussion here.

discussion, we sometimes refer to [3], which not only contains a general and elaborate axiomatic semantics, but also collects numerous litmus tests for illustrating their axiomatic framework. Other relevant related work is captured in the tutorial [19].

Axiomatic semantics operates on *event graphs*, i.e., graphs with events as nodes and various relations between the events as edges. Events correspond to occurrences of executed instructions, with loads and stores as the most basic form of memory interaction. Relations capture various forms of dependencies between instructions; for example, data dependencies and intra-process control-flow dependencies such as program order. The graph, in particular the relationship between the events, depends on the program code itself and also on the memory model. One program typically gives rise to more than one such graph, each of which represents one possible "run" of a program. Still, such graphs only represent *candidate* executions, as unrealizable graphs are filtered out by axioms. The axioms spell out conditions on the various relations, typically requiring *acyclicity* of particular combinations of the involved edges. Aspects of a memory model are often captured or illustrated by so-called litmus tests, which are tailor-made code snippets used to highlight expected or disallowed behavior in a given setting. Thus, a litmus test can also be seen as a partial pre- and post-specification for a small piece of concurrent code. The precondition, often left implicit, assumes that all shared variables are in some definite, initial state.

As illustration, Figure 2 contains the well-known litmus test for "message passing" on the left and a corresponding candidate execution on the right. The crucial question for the given code is whether the observation $r_1 = 1$ and $r_2 = 0$ is possible. The intention of the code is the following: process p_0 wants to send data via x to p_1 and uses a write to y to signal that the value is "ready" to be read. In a memory model where the litmus test of Figure 2 is expected to work without additional synchronizing instructions, the observation $r_1 = 1$ and $r_2 = 0$ must be *forbidden*. The assumption, in this case, is that the order of reads by p_1 reflects the order in which the writes are effected (when done in program order) by p_0. Under this assumption, one can generalize the pattern in that p_1 repeatedly reads the flag variable y in a busy wait loop until it is assured that the value communicated via x is ready to be read.

In moderately weak memory models, ones with per-location write buffering in particular, message passing is not realizable without additional synchronizing instructions. The candidate execution of Figure 2b shows exactly that; it gives a justification for the observation $r_1 = 1$ and $r_2 = 0$, which violates the MP pattern. The edge $n_2 \rightarrow_{\mathsf{rf}} n_3$ of the "read-from" relation $\rightarrow_{\mathsf{rf}}$ simply expresses the fact that n_3 reads the value written by n_2. More complex is the "from-read" relation: the edge $n_4 \rightarrow_{\mathsf{fr}} n_1$ stipulates that n_4 "reads-from" some write event left unmentioned for which n_1 comes "after." More precisely, it abbreviates $n_0 \rightarrow_{\mathsf{rf}} n_4$ for some write event n_0 with $n_0 \rightarrow_{\mathsf{co}} n_1$ and where $\rightarrow_{\mathsf{co}}$ represents the so-called *coherence order*, which is a total order of writes over the same memory location. In the example, n_0 is the write event setting x to its initial value 0 (which, by convention, is often left out in representations of candidate executions). Using the mentioned coherence order, the from-read relation captures the intuition that a read observes a value which is *not* yet overwritten by a given write. In contrast, in our setting as well as in others, the information on which writes are observable by a read is *local* per observing thread. Conceptually, there is no notion of a total order of writes



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

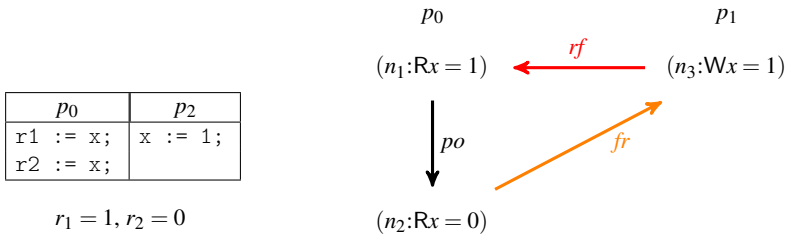
$r_1 = 1, r_2 = 0$

(a) Litmus test (b) Candidate execution

Fig. 2: mp

on a location. Past writes events are seen as unordered even if issued by one thread or when seen from the perspective of one observing thread.[8]

One, perhaps unfamiliar, aspect of our semantics is that writes, once performed, never invalidate earlier writes (at least those done by a thread different from the observer). In the absence of synchronization, writes remain observable indefinitely. A litmus test typifying that kind of behavior is known as coRR,[9] shown in Figure 3.



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

$r_1 = 1, r_2 = 0$

(a) Litmus test (b) Candidate execution

Fig. 3: coRR

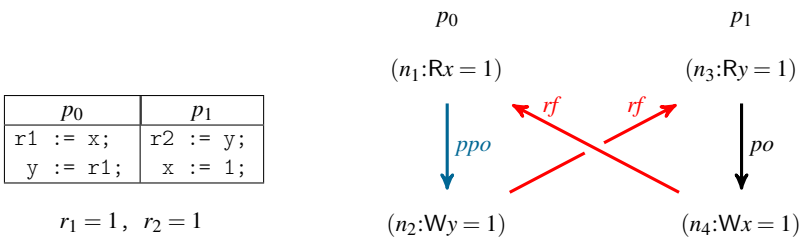
The fact that repeated reads by the same thread give different seemingly incoherent values may seem odd at first. It can, however, be interpreted as a form of oscillation: when reads and writes happen "at the same time," i.e., in a racy way without proper synchronization, the memory can be understood as oscillating between the racy memory updates. In the example, the value can, in theory, be perceived as oscillating between 0 and 1 indefinitely. This behavior is allowed by our proposed semantics. As a matter

[8] Except in the special single threads case, where the reads and writes are done by the same thread.

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

of fact, it is also officially allowed by Sparc RMO [15] and pre-Power4 machines [21]. Many other models, however, including the axiomatization by [3], disallow it.

Note that our treatment of write-accesses is rather liberal and therefore, some of the litmus tests for write buffering also characterize our semantics. For example, both our model and PSO-style memory models with per-location write buffers allow the observation $r_1 = 1$ and $r_2 = 0$ in the mp litmus test of Figure 2. On the other hand, from our perspective, the treatment of the writes is best not seen as "buffering;" after all, the value of a write in the operational semantics becomes *immediately* observable. It is the *negative* information of being *unobservable* that is not immediately effective for all observers. This negative information requires synchronization via channel communication in order for it to percolate though the concurrent system.



(a) Litmus test (b) Candidate execution

Fig. 4: Load buffers (lb)

Our basic weak semantics treats writes in a rather relaxed manner. Reads, in contrast, are treated in a conventional or "strong" way. *Load buffering* is a relaxation which complements write buffering. The effect of load buffers is often illustrated by the litmus test of Figure 4. The candidate execution graph on the right shows a run which justifies $r_1 = 1$ and $r_2 = 1$. This execution can be interpreted as follows: the load or read of event n_1 is buffered, thereby taking effect after the write event n_4. This buffering causes the instructions n_3 and n_4 to seem executed *out-of-order*, thus, the program order $n_3 \rightarrow_{\mathsf{po}} n_4$ is perceived as "ignored." For p_0, however, the read cannot be postponed until after the write instruction, as the value of the write *depends* (via r_1) on the value being read: the read and write events in p_0 are in program order, but unlike the situation in p_1, the load cannot be buffered; program order has to be preserved due to a data dependence. The preserved program order is marked in the graph by a $\rightarrow_{\mathsf{ppo}}$-edge. The circumstances in which program order is preserved or not depends on the programming language semantics and/or the given hardware memory model. For example, various forms of special fence instructions (e.g. light-weight fences, full fences, control fences) may be available on a given platform.

Load buffering is conceptually more challenging than write buffering. Thinking operationally, dispatching a write instruction in an asynchronous manner is like "fire-and-forget," but executing an "asynchronous" read means the corresponding process

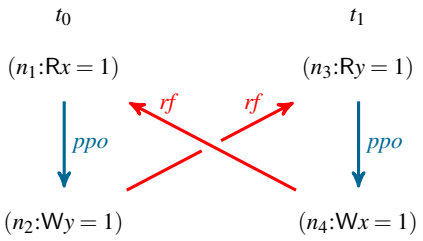
continues regardless of whether the value it wishes to read has been obtained. This non-blocking nature in particular problematic if it is assumed (as in our happens-before model) that reading a value, buffered or not, is done *without* any synchronization.[10] Subsequent code may *depend* on the value being read; the dependency may not only be a data-dependency (as the write to y in Figure 4a), but also a control flow dependency, where choosing a branch to be taken depends on a value that is not yet available.

One important aspect in connection with load buffering is illustrated in Figure 5. It closely resembles the previous case from Figure 4. In particular, the four memory events in the candidate execution graph in Figure 5b are *identical* to those in Figure 4b. The crucial difference is an additional data dependency in p_1: the write statement has a data dependency on the preceding read event. This dependency is reflected in the graph by a *ppo*-edge, as opposed to a *po*-edge as in Figure 4b.

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

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

(a) Litmus test



(b) Candidate execution

Fig. 5: lb+ppos

The outcome $r_1 = 1 = r_2$ of the litmus test could be justified by the following argument, as illustrated by the candidate execution graph: n_1 reads the value 1 written by n_4, subsequently used in the write n_2, which in turn is read by n_3, and used in the write event n_4. This justification involves a circular argument, and it is able to produce a value, the number 1, that does not even appear in the program text. Such behavior is termed "out-of-thin-air" and is generally, though not universally, considered illegal. In other words, the *candidate* graph of Figure 5b, which justifies the post-condition of the litmus test, is a candidate that is ruled out by many memory models. Note, however, that in the informal happens-before Go memory model, out-of-thin-air behavior of this kind is allowed, as there are no statements or mechanisms specified which forbid the behavior.

Note that the candidate execution from Figure 5b is, for *that* model, not significant. In particular the shown $\rightarrow_{\mathsf{ppo}}$-edges presuppose that the order of the reads and the subsequent write cannot be changed, i.e., due to the data dependency of the litmus test, there is no conceptual load-buffering in that particular situation. The Go memory model, on the other hand, does not mention data dependencies nor "preserved program order."

[10] Remember, that basically the *only* way of achieving synchronization is channel communication.

Instead, it operates with the plain notion of program order $\rightarrow_{\mathsf{po}}$, stipulating that $\rightarrow_{\mathsf{po}} \subseteq \rightarrow_{\mathsf{hb}}$. Therefore, in the situation of Figure 5, the out-of-thin-air observation is perfectly acceptable.[11] Certainly, the happens-before relation is assumed to be acyclic,[12] but still, the situation of Figure 5b does not constitute a happens-before cycle as $\rightarrow_{\mathsf{hb}}$ is not defined making use of the read-from relation. As a final remark, it should be noted that in [3] and similar sources, the happens-before relation is a *derived* relation. The relation is defined as an extension of preserved program order $\rightarrow_{\mathsf{ppo}}$, taking into account read-from-edges. In our discussion, however, the happens-before relation consists of $\rightarrow_{\mathsf{po}}$ and the dependencies implied by channel synchronization. The condition (3) can be interpreted as analogue to a $\rightarrow_{\mathsf{rf}}$-edge, though based on channel communication as opposed to communication via an unsynchronized shared variable.

4 Memory as collection of past write events

In this section we sketch the operational formalization of our memory model by highlighting a few of its central rules. We slightly simplify the rules and favor communicating intuition over notational rigor.

The model is formulated as structural operational semantic rules over program configurations. Our configurations contain "sets" of processes or threads, channels, and a "memory component." Memory consists of the collection of past write events. The write events remembered in the configuration are written as $n(\!|z{:=}v|\!)$ with n as unique identifier. To regulate observability in accordance to the happens-before relation, each thread keeps track of whether write events are, from the perspective of the given thread, *unobservable* (we use "unobservable" and "shadowed" interchangeably). Additionally, threads keep track of write events that are locally known to be in happens-before relation to their current point of execution. Happens-before and shadowed sets form a tuple (E_{hb}, E_s) which is referred to as "local state" and denoted as σ in the rules of the operational semantics.

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

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

Table 1: Operational steps: reads and writes

Table 1 shows the rules covering reads and writes. Executing a write instruction spawns a new corresponding write event and updates the local state σ of the execut-

[11] That is not to say that language implementations will exhibit that behavior, just that out-of-thin-air is consistent with the specification.

[12] Note that it is not stated explicitly in [14] but it is safe to assume that is what is intended.

ing thread. The update records information about the write event as part of the issuing thread's happens-before memory. Additionally, when it comes to the variable in question, all events that are known to have happened-before are then marked as shadowed. In a single-threaded setting, this book-keeping realizes visibility according to program order, which means, the only value observable for each variable is the one written last. This behavior is what one would expect from a sequential program.

Channels communication is responsible not only for message passing communication but also for synchronization between threads. The synchronization conditions are spelled out in equations (3) and (4). Equation (3) corresponds to the fact that receiving over a channel is a potentially blocking operation: the receiver has to wait until a value is available. This blocking is a manifestation of the control-flow aspect of synchronization, as it disables, under certain conditions, steps from being taken. More concretely, equation (3) connects the steps of the sender before the send with the steps in the receiver after the reception; this connection is made by the $\rightarrow_{\mathsf{hb}}$-relation. In the operational interpretation of our semantics a channel send and receive communicates not just the communicated value, but also the local state σ from the sender to the receiver (see table 2). Upon reception, the receiver updates its local state with the new information. This update is reflected by setting $\sigma' = \sigma + \sigma''$ as shown in rule R-REC. With the update, the receiver increases its knowledge about which events have happened-before, thereby increasing the events which become *un*observable from the receiver's perspective.

Equation (4) works in reverse direction and represents synchronization due to the *boundedness* of channels. Boundedness connects the happens-before knowledge of a sender with that of a receiver on the same channel. Note that this connection is not necessarily from the receiver of a value back to the sender of said value: passing a message from a process p_1 to a process p_2 via a channel c makes the send of p_1 to happen-before an earlier receive on c. This earlier receive was not necessarily done by p_2; it could have been done by a "third party" p_3. In the operational semantics, this "receiver-to-sender" synchronization is captured by the transmission of local state σ in the reverse direction. Each program level channel is thus represented by two channels, a "forward" channel, transmitting in a conventional manner the data being transmitted (plus the happens-before and shadow information) and a "backward" channel (transmitting only happens-before and shadow information). Thus, receiving a value in rule R-REC not just dequeues one entry from the forward queue, but enqueues also the receiver's current σ into the backward channel. Correspondingly, a send in rule R-SEND is enabled only, if the backward queue is not empty.

Communication via synchronous channels, which can be seen as channels with capacity 0, is treated separately in rule R-SEND-REC. This rule leads to an immediate exchange of the local states between sender and receiver. Synchronous communication, therefore, corresponds to a full bidirectional fence between the two partners engaged on a rendezvous. There are additional rules, left out here, that deal with creating and closing channels.

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

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

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

Table 2: Operational steps: basic channel communication

5 Load buffering

In order to further relax the memory model described in the previous section, we need to introduce load buffering, also referred to as *delayed reads*. The presence of load buffering, however, gives rise to a number of complications. For one, reads have to be executed *asynchronously:* a thread reading a value has to proceed even if the value is not yet available. Such "non-blocking" reads are problematic in that a thread's continuation after an asynchronous read typically depends on the value read. Process p_0 in the code of Figure 4a and both processes in Figure 5a contain examples of this type of dependency. In these examples, the dependency is a pure *data dependency*: a "subsequent" instruction writes the value read back to memory.

Control dependencies, which have not yet been discussed in connection with the examples from Section 3, are more complicated than data dependencies. A control dependency happens, for example, when the condition of an if-then-else involves the value read by a prior load-instruction. If the load is executed "out-of-order" and after the conditional expression, then both branches are in principle possible. This superposition leads to a form of *speculative* execution.

The asynchronous execution of a load instruction is similar to a simple *future*-like mechanism; one involving *implicit* futures. While futures allow the asynchronous execution of arbitrary sequential code that produces an eventual value, asynchronous reads execute only the `load`-instruction and nothing else. We say that the load is done, or the "future" is resolved, when a concrete value can be passed back to issuer of the asynchronous read.

Asynchronous loads and futures differ in terms of synchronization. Conventionally, futures involve a form of synchronization that is sometimes called "wait-by-necessity," meaning that the caller blocks when accessing results that are not yet available. With its strict separation between synchronization (achieved by sending-to and receive-from channels) and communication, reads and writes are *completely devoid* of synchronization. Informally, the lack of synchronization can be seen as a "don't-wait-not-even-

when-necessary" semantics. In other words, a read and a subsequent write —subsequent in terms of program order— can be swapped and executed out-of-order. Thus, as an example, the "preserved-program-order" edge in Figure 4b would be represented in our model as $\rightarrow_{\mathsf{po}}$-edge instead.

To delay the read and thereby account for out-of-order execution, configurations are extended with read events (in addition to the write events already discussed in the previous section). The semantics is also extended to deal with references to future values. For example, before adding delayed reads, store instructions took the form $z := v$ where v denotes a user-level data value such as an integer. Once delays are added, a new form of the store instruction is needed: $z := n$ with n a "future reference" to an asynchronous read. Note that, as shown in rule R-READ, future references are written $n[\![\sigma, \texttt{load}\, z]\!]$.

When it comes to ensuring that "outdated" or shadowed writes are not observable by a read instruction (see equation (2)), the semantics with asynchronous reads differs from the semantics without them. In the presence of asynchronous reads, the future reference $n[\![\sigma, \texttt{load}\, z]\!]$ is responsible for making sure that shadowed writes are not observed. In other words, the future reference must ensure that the value resulting from an asynchronous read does not come from a write that is shadowed to the reader. In the semantics without asynchronous reads, a thread's shadow set was used to exclude visibility of shadowed writes. This exclusion is captured by the corresponding premise of the R-READ rule of Table 1.

When it comes to causality (see equation (1)), the semantics lacking asynchronous reads trivially guarantees that a read instruction cannot observe writes that happen-after. Once asynchronous reads are introduced, however, additional measures must be taken so causality is not violated. Note that the σ-information of the reader is of no help here: in the operational execution, the write event is generated *after* the asynchronous issuing of $n[\![\sigma, \texttt{load}\, z]\!]$. Consequently, the subsequent write is unknown to $n[\![\sigma, \texttt{load}\, z]\!]$ and, therefore, the write is not mentioned in σ. In general, the fact that a read event does *not "know"* about a write event is useless to determine whether the read can observe said write event (as both is consistent with being not mentioned in the local σ of the read event). Instead, it is from the perspective of the write-event that one can determine whether a read can observe the write: if the write event "knows" that a read event has "happened-before," then the write should not be made observable to the read. Thus, write events now also carry happens-before and shadow information and are of the form $n(\!|\sigma, z := v|\!)$. Note also that the conditions corresponding to equations (1) and (2) are captured by the premises of rule R-OBS from Table 3.

The implicit future references not only occur on the right-hand side of the let-construct (see rule R-READ), but can also occur as stand-in for the value to be loaded from memory. In particular, assignments may take now the form $z := n$ where n is a reference to a value that has not yet been resolved. Consequently, write events may take the form $n_1(\!|\sigma, z := n_2|\!)$ and delayed read constructs can take the form of a reference to a reference: $n_1[\![n_2]\!]$. As references are resolved into concrete values, the semantics must have rules to shorten chains of indirections. For instance, $n_1(\!|\sigma, z := n_2|\!) \parallel n_2[\![v]\!] \rightarrow n_1(\!|\sigma, z := v|\!) \parallel n_2[\![v]\!]$. Additionally, the semantics needs to deal with compound expressions containing future references, which likewise cannot be immediately evaluated. The corresponding dereferencing rules are left out in this discussion. Also left out of

$$\frac{\mathit{fresh}(n)}{p\langle\sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \ \rightarrow\ p\langle\sigma, \texttt{let}\ r = n\ \texttt{in}\ t\rangle \parallel n[\![\sigma, \texttt{load}\ z]\!]}\ \text{R-READ}$$

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

$$\frac{\sigma_1 = (\_, E_s^1) \qquad \sigma_2 = (E_{hb}^2, \_) \qquad n_2 \notin E_s^1 \qquad n_1 \notin E_{hb}^2}{n_1[\![\sigma_1, \texttt{load}\ z]\!] \parallel n_2(\!|\sigma_2, z := v|\!) \ \rightarrow\ n_1[\![v]\!] \parallel n_2(\!|\sigma_2, z := v|\!)}\ \text{R-OBS}$$

Table 3: Operational semantics: shared memory

this note is the exact treatment of *control* dependencies in the presence of delayed reads. When the decision on which branch to take depends on the value of a read that is not yet resolved, the semantics allows execution to continue in a speculative manner, collecting a symbolic representation of the branch condition in the form of a path condition.

Example 1 (Out-of-thin-air). For illustration, let us revisit the litmus test from Figure 5a. Assume that both threads start with a local state of σ_0. We don't show the write events for initialization of the two shared variables. Now, after executing four instructions, the resulting configuration may continue as follows, dereferencing the load reference:

$$\begin{aligned} &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![\sigma_0, \texttt{load}\ x]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![\sigma_0, \texttt{load}\ y]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \ \rightarrow \\ &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![n_2]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![\sigma_0, \texttt{load}\ y]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \ \rightarrow \\ &p_1\langle\sigma_1, n_1\rangle \parallel n_1[\![n_2]\!] \parallel n_1'(\!|\sigma_0, y := n_1|\!) \parallel n_2[\![n_1]\!] \parallel n_2'(\!|\sigma_0, x := n_2|\!) \parallel p_2\langle\sigma_2, n_2\rangle \end{aligned}$$

After these two dereferencing steps, the configuration contains $n_1[\![n_2]\!]$ and $n_2[\![n_1]\!]$, each attempting to solve each other's indirect reference in some form of *deadlock*. Adding a rule that "resolves" such cyclic dependencies by picking a random value (in a form of self-justification) allows the modeling of out-of-thin-air behavior. □

6 Conclusion

We present the ideas behind an *operational* specification for a weak memory model. The semantics is accompanied by an implementation in the $\mathbb{K}$ framework and by several examples and test cases [10]. We plan to use the implementation towards the verification of program properties such as data-race freedom.

Bibliography

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

[2] S. V. Adve and M. D. Hill. Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a):2–14, 1990.

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

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

[5] Programming languages — C++. ISO/IEC 14882:2001, 2011.

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

[7] H.-J. Boehm and B. Demsky. 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, 2014. ACM.

[8] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

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

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

[11] D. Fava, M. Steffen, and V. Stolz. Operational semantics of a weak memory model with channel synchronization. In K. H. et.al., editor, *FM*, volume 10951 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, July 2018.

[12] D. Fava, M. Steffen, and V. Stolz. Operational semantics of a weak memory model with channel synchronization: Proof of sequential consistency for race-free programs. Technical Report 477, University of Oslo, Faculty of Mathematics and Natural Sciences, Dept. of Informatics, Jan. 2018. Available at `https://www.duo.uio.no/handle/10852/61977`.

[13] The Go programming language specification. `https://golang.org/ref/spec`, Nov. 2016.

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

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

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

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

[18] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05*. ACM, Jan. 2005.

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

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

[21] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Q. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.