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

Daniel S. Fava,[1] Martin Steffen,[1] Volker Stolz[1,2] and Stian Valle[1]

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

A *memory model* dictates the order in which memory operations appear to execute; it also affects how processes communicate in a parallel computing setting with shared variables. The design of a proper memory model is a balancing act. On one hand, memory models should be lax enough to allow common current hardware and compiler optimizations and potential future ones. On the other, the more lax the model, the greater the burden on developers, who need to reason about the correctness of their programs. 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*. These are counter intuitive results that can be justified through circular reasoning. In this work we propose a memory model that precludes out-of-thin-air behavior, is fairly relaxed (allows writes to different memory location to appear out of order), and is based on a calculus with processes and buffered channels.

The calculus we propose is inspired by the Go programming language developed at Google. Go recently gained traction in networking applications, web servers, distributed software and the like. It features goroutines (i.e., asynchronous execution of function calls resembling lightweight threads) and buffered channel communication in the tradition of CSP or Occam. The *happens-before relation* is used in the Go memory model to describe which reads can observe which writes to the same variable. It says, for example, that *within a single goroutine, the happens-before relation boils down to program order.* The language is particularly suited for concurrent programs. 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 [2], where *a send on a channel happens before the corresponding receive from that channel completes.* Synchronization is also imposed by the boundedness of channels: *The k^{th} receive on a channel with capacity C happens before the $(k+C)^{th}$ send from that channel completes.*

Though relaxed, the Go memory model does not preclude out-of-thin-air behavior. Next, we formalize an operational semantics that is less relaxed than Go's, yet it precludes out-of-thin-air behavior.

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. 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). We often use c specifically for references to channels.

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. 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. Therefore, expressions like $x \leftarrow \texttt{load}\, z$ or $x \leftarrow z$ are disallowed.

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

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

Table 1: Abstract syntax

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 `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.

Select-statements, written using the $\sum$-symbol, consists of a finite set of guarded branches. Only communication statements, i.e., channel sending and receiving, or the `default`-keyword are allowed as guards (with at most one `default` per select). A channel can be mentioned in more than one guard; sending and receiving guards can also 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 `let` $r = e$ `in` t combines sequential composition and the use of scopes for local variables r: after evaluating e, the rest t is evaluated where the resulting value of e is handed over using r. The let-construct is seen as a binder for variable r in t. It becomes *sequential composition* when r does not occur free in t. We use semicolon as syntactic sugar in such situations.

Operational semantics with write buffering

Programs consist of the parallel composition of goroutines $\langle \sigma, t \rangle$, write events $n(\!|z{:=}v|\!)$, and channels $c[q_f, q_b]$; with write events and channels being part of the run-time configuration. In the current semantics, read accesses to the main memory cannot be delayed; consequently, there are no read events.

Write events are 3-tuples from $N \times X \times Val$; they records the shared variable being written to and the written value, together with a unique identifier n. A write $n(\!|z{:=}v|\!)$ to variable z is said to be *shadowed* if another write event n' to variable z happened after n but before the current point in time.

In addition to the code t to be executed, goroutines $\langle \sigma, t \rangle$ contain local information about earlier memory interaction. In first approximation, the local state contains information about events which occurred earlier. *Local states* σ are are tuples of type $2^{(N \times X)} \times 2^N$ abbreviated as Σ. We use the notation (E_{hb}, E_s) to refer to the tuples. The first component of the local state E_{hb} contains the identities of all write events that have happened before the current stage of the computation of the goroutine. The second component E_s of the local state represents the set of identities of write events that, at the current point, are shadowed.

From a goroutine's point of view, its reads and writes appear in program order. This is guaranteed by the absence of delayed reads and by disallowing reads from obtaining values of writes that have been shadowed. Also, from a goroutine's point of view, writes from other goroutines appear out of order. This is because writes are placed on a global pool and because subsequent reads can read any write currently in the pool. The reduction rules for reads and writes are on our technical report [1].

Synchronization between goroutines is achieved by communicating via channels as shown in Table 2. A channel is of the form $c[q_f, q_b]$, where c is a name and (q_f, q_b) a pair of queues referred to

$$\frac{\mathit{fresh}(c)}{\langle\sigma, \mathtt{let}\ r = \mathtt{make}\ (\mathtt{chan}\ T, v)\ \mathtt{in}\ t\rangle \to \langle\sigma, \mathtt{let}\ r = c\ \mathtt{in}\ t\rangle \parallel c_f[] \parallel c_b[]}\ \text{R-MAKE}$$

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

$$\frac{|q_1| < \mathit{cap}(c) \qquad \sigma' = \sigma + \sigma''}{c_b[q_2 :: \sigma''] \parallel \langle\sigma, \mathtt{pend}\ c; t\rangle \parallel c_f[q_1] \to c_b[q_2] \parallel \langle\sigma', t\rangle \parallel c_f[q_1]}\ \text{R-PEND}$$

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

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

Table 2: Operational semantics: message passing

as *forward* and *backward* queue respectively. In addition to communicating a value, the queues are managed so that sends and receives trigger exchanges of happened-before and shadowing knowledge between the communicating partners.

Contributions

We propose an operational semantics for a memory model that precludes out-of-thin-air behavior, is fairly relaxed, and is based on a calculus with processes and buffered channels [1]. We have implemented of the calculus in the $\mathbb{K}$ framework [3], an executable semantics framework based on rewriting logic, and we are currently working on a proof that the model exhibits the so called data race free guarantee. The DRF guarantee says that data race free programs (i.e. programs that are properly synchronized) have sequentially consistent behavior. Our goal is to further relax the model by introducing delayed reads, yet, at the same time, keeping out-of-thin-air behavior at bay.

References

[1] Daniel Fava, Martin Steffen, Volker Stolz, and Stian Valle. An operational semantics for a weak memory model with buffered writes, message passing, and goroutines. Technical Report 466, 2017.

[2] Go memory model. The Go memory model. `https://golang.org/ref/mem`, 2016.

[3] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi: 10.1016/j.jlap.2010.03.012.