Operational Semantics of a Weak Memory Model
inspired by Go

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 which values may be returned when reading from memory. In a parallel computing setting, the memory model affects how processes communicate through shared memory. The design of a proper memory model is a balancing act. On one hand, memory models must be lax enough to allow common hardware and compiler optimizations. On the other, the more lax the model, the harder it is for developers to reason about their programs. In order to alleviate the burden on programmers, a weak memory model should provide what is called the *data-race freedom guarantee*, which allows reasoning in terms of sequential consistency provided a program is data-race free.

In this paper we present a theory for weak memory with channel communication as the sole synchronization primitive. There are few studies on channel communication as a synchronization primitive for a weak memory model. We formalize the memory model in a small-step operational semantics and implement it in an executable semantics framework [4] from which we obtain an interpreter. Similar to Boudol and Petri, we favor an operational semantics because it allows us to prove the DRF guarantee "at the programming language level." This yields a more concrete interpretation of the DRF guarantee as compared to formalisms in which the notion of a program is abstracted away, often in the form of a graph [1].

The calculus we propose is inspired by the Go programming language developed at Google, which 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 Go memory model

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* and, between goroutines, events can appear to happen out of program order. If the effects of a goroutine are to be observed by another, a synchronization primitive 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 [3]. In particular, it states that *a send on a channel happens before the corresponding receive from that channel completes.* Our semantics incorporates channels for message passing, goroutines for asynchronous code execution, and it allows for out-of-order execution where writes to memory can be arbitrarily delayed.

Abstract syntax

The abstract syntax of the calculus is given in Table 1. *Values* are written generally as v and include booleans, integers, and etc (these more obvious values are not explicitly listed on the table). Note that local variables (or registers) are also counted as values and are denoted r. Names (or references) are also considered values and are denoted n. Names are used, for example, when referring to different channels – when presenting the semantics, we will use c for indicating a reference to a channel.

A new channel is created by `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

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

Table 1: Abstract syntax

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 but is not part of the syntax available to the programmer.

Starting a new asynchronous activity (i.e., goroutine) is done using the `go`-keyword. Select-statements, written using the $\sum$-symbol, consist of a finite set of guarded branches. The `let`-construct $\texttt{let}\, r = e\, \texttt{in}\, t$ combines sequential composition and the use of scopes for local variables r. It becomes *sequential composition* when r does not occur free in t. We use semicolon as syntactic sugar in such situations.

Operational semantics with delayed writes

Programs consist of the parallel composition of goroutines $\langle \sigma, t \rangle$, write events $n(\!|z{:=}v|\!)$, and channels $c[q_f, q_b]$. *Write events* are 3-tuples from $N \times X \times Val$; they record the shared variable being written to and the written value, together with a unique identifier n. In the current semantics, 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. *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 of the local state, E_s, represents the set of identities of write events that, at the current point, are shadowed (i.e., no longer visible to the goroutine).

The reduction rules for reads and writes are given on Table 2. 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. Writes from other goroutines, however, may appear out of order: writes are placed on a global pool and subsequent reads can read any write from the pool as long as the event has not been shadowed from the reader's point of view.

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 as *forward* and *backward* queue. For convenience, we use c_f and c_b when referring to channel's c forward and backward queues. When creating a channel (cf. rule R-MAKE), the forward channel is initially empty but the backward is not: it is initialized by a queue of length $|c|$, which corresponds to the capacity of the channel (the channel is synchronous when capacity is 0). In order to account for the synchronization power of channels, in addition to communicating a value, the queues are managed so that "happened before" and "shadowed" knowledge are also exchanged between communicating partners.

See our technical report for a detailed description of the semantics [2].

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

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

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

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

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

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

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

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

Table 2: Operational semantics: message passing and memory reads/writes

Contributions

- We define a novel semantics for weak memory with channel communication as synchronization primitive;
- We prove that our proposed weak memory upholds the *data-race freedom* guarantee;
- We present our implementation in an executable semantics framework, which allows us to derive an interpreter for programs in the target language.

References

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

[2] 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, University of Oslo, Dept. of Informatics, April 2017.

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

[4] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Methods in Programming*, 79(6):397–434, 2010.