Finding and fixing a mismatch between the Go memory model and data-race detector.

A story on applied formal methods

Daniel Schnetzer Fava
danielsf@ifi.uio.no

Dept. of Informatics, University of Oslo

Abstract. Go is an open-source programming language developed at Google. In previous works, we presented formalizations for a weak memory model and a data-race detector inspired by the Go specification. In this paper, we describe how our theoretical research guided us in the process of finding and fixing a concrete bug in the language. Specifically, we discovered and fixed a discrepancy between the Go memory model and the Go data-race detector implementation—the discrepancy led to the under-reporting of data races in Go programs. Here, we share our experience applying formal methods on software that powers infrastructure used by millions of people.

1 Introduction

Go is an open-source programming language designed for concurrency. Developed at Google, the language has gained traction in the areas of cloud computing [7], where it is used to implement various client-server applications and container management systems, such as Docker [13] and Kubernetes [2].

One of the language's main features are light-weight threads, called *goroutines*, which are spawned during function invocation. Any function can be made to execute asynchronously by simply prepending the keyword `go` to the function's name during invocation. Go's approach to synchronization also stands out. *Do not communicate by sharing memory; instead, share memory by communicating* [9]—is a catchphrase among Go programmers. The language's feature-mix encourages a style of programming where (1) variables are implicitly owned by goroutines, and (2) variables are shared when this ownership is transferred through direct communication. So, in contrast to locks, which favor synchronization via mutual exclusion, Go has channels, which typically enforce a happens-before relation [11] between a message sender and its receiver.

The discipline prescribed by Go's *share by communicating* slogan is not, however, enforced at compile time.[1] It is, therefore, possible for programs to harbor data races. Since data races often lead to counter intuitive behavior,

[1] There are good reasons why a type checker cannot enforce such a discipline without seriously restricting the language.

the Go programming language comes with a data-race detector built into its toolchain.

The Go memory model is relaxed and its specification describes the behavior of well-synchronized programs. In [4], we gave a small-step operational semantics of a memory model inspired by Go's. There, we proved the DRF-SC guarantee, which states that data-race free (DRF) program executions behave sequentially consistently (SC) under the proposed model. Given the importance of flagging data races, in [5] we explore the use of our semantics for the sake of data-race detection. Armed with these formalisms, we turned our attention to Go's implementation. With that, we discovered that the Go data-race detector was not strictly abiding by the rules of the Go memory model specification. This oversight lead to the under-reporting of data races in Go programs. We then proposed and implemented a fix in conjunction with the Go community. Here, we discuss how the theoretical modeling of the language helped us find and address this issue.

In Sections 2 and 3, we will visit the Go memory model and explore examples of synchronization via channel communication. Having covered this background, we discuss how the Go data-race detector is built into the language (Section 4). In Section 4.1, we show that the detector's implementation inadvertently mismatched rules governing channel communication. We address the issue in Section 5 and share lessons we learned in Section 6.

2 Synchronization via channel communication

Two concurrent memory accesses constitute a data race if they reference the same memory location and at least one of the accesses is a *write*. Data races can be eliminated through synchronization, that is, the enforcement of an order between conflicting memory accesses. In Go, synchronization is performed via channel communication. Go channels assure FIFO communication from a sender to a receiver sharing the channel's reference. Channels can be dynamically created and closed—their type and finite capacity are fixed upon creation.

When attempting to receive from an empty channel, a thread blocks until, if ever, a value is made available by a sender. A thread also blocks when attempting to send on a channel that is full. According to the Go memory model specification [8], the following two main rules govern synchronization. Given a channel c with capacity C:

I. A send on c happens-before the corresponding receive from c completes.

II. The k^{th} receive from c happens-before the $(k + C)^{th}$ send on c completes.

The first rule establishes a causal relationship between a sender and its communicating partner. In contrast, the second rule establishes a relationship between a sender and some past receiver, without there being any message transmission between the two goroutines. Note also that the second rule accounts for channel capacity: a current sender is able to place a new message because some past receiver, by taking an older message out, has made space in the channel's buffer.

Figure 1a is an example of synchronization via rule (I), and Figure 1b is an example via rule (II). Throughout the paper, we will follow the syntax in [4], which closely matches Go's. The term $c \leftarrow v$, with the arrow pointing into c, stand for the sending of value v over channel c. Let $\leftarrow c$, with the arrow pointing away from c, stand for the reception of a value from the channel. Assuming a channel of capacity one, Figure 1a is the classic message passing example, while Figure 1b enforces mutual exclusion.

$T0$	$T1$
$z := 42$	$\leftarrow c$
$c \leftarrow 0$	`load` z

(a) Message passing example.

$T0$	$T1$
$c \leftarrow 0$	$c \leftarrow 0$
$z := 42$	$z := 43$
$\leftarrow c$	$\leftarrow c$

(b) Mutual exclusion example.

Fig. 1: Synchronization via channel communication (channels of capacity one).

In the message passing example, the goroutine $T0$ writes to a shared variable z and, by sending a message over a channel, the routine transfers its implicit ownership of z. Goroutine $T1$ blocks until a message is ready to be received. Once a message has been received, $T1$ proceeds to load from z. This program is properly synchronized, which means $T1$ necessarily loads the value of 42 as opposed to potentially observing an uninitialized variable value. Using the happens-before (HB) rules of the Go memory-model specification, we can show that the memory accesses are properly synchronized as follows:

$$
\begin{aligned}
z := 42 &\sqsubset_{hb} c \leftarrow 0 && \text{via program order} && (1)\\
c \leftarrow 0 &\sqsubset_{hb} \leftarrow c && \text{via channel rule (I)} && (2)\\
\leftarrow c &\sqsubset_{hb} \texttt{load}\ z && \text{via program order} && (3)\\
z := 42 &\sqsubset_{hb} \texttt{load}\ z && \text{via (1), (2), (3) and transitivity of HB.}
\end{aligned}
$$

While Figure 1a and rule (I) account for direct communication, Figure 1b relies on rule (II) and the use of channels as locks. The example in Figure 1b involves two threads attempting to write to the same shared variable. Before writing, a thread sends a message onto a channel. Because the channel has capacity one, all subsequent attempts to send again will block until the prior message is received. Therefore, it is not possible for $T0$ and $T1$ to execute their critical sections at the same time. The send is thus analogous to acquiring a lock, and the receive to releasing the lock. Again, we can use the Go memory model to reason about this example. Without loss of generality, assume $T0$ sends its

message first, then

$$z := 42 \quad \sqsubset_{hb} \quad \leftarrow c \text{ by } T0 \qquad \text{via program order} \qquad (4)$$
$$\leftarrow c \text{ by } T0 \quad \sqsubset_{hb} \quad c \leftarrow 0 \text{ by } T1 \qquad \text{via channel rule (II)} \qquad (5)$$
$$c \leftarrow 0 \text{ by } T1 \quad \sqsubset_{hb} \quad z := 43 \qquad \text{via program order} \qquad (6)$$
$$z := 42 \quad \sqsubset_{hb} \quad z := 43 \qquad \text{via (4), (5), (6) and transitivity.}$$

While mutual exclusion is ensured, we cannot ascertain the final value of z. If $T0$ sends a message before $T1$, then z equals 43; otherwise, $z = 42$. Note also that, in this example, rule (I) is obviated by the *program order;* therefore, the rule has no synchronization effect here.[2]

The Go memory model is described plain and succinctly in English [8]. The word "completes," present in both rules (I) and (II), can easily be overlooked. By overlooking this distinction, the Go data-race detector over-synchronizes and, therefore, fails to report certain races. The bug, which we describe in detail in the next section, is related to the following question: Is it possible for the detector to account for the mutex paradigm (Figure 1b) and, at the same time, observe the distinction between a channel operation and its completion?

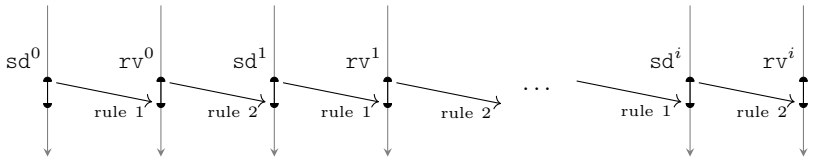
3 The Go memory model: Every word counts

The completion of a channel operation, in addition to the operation itself, is an important part of the Go memory model. In rule (I), it is not the case that a send happens-before the corresponding receive. Instead, the send happens-before the *completion* of the corresponding receive. Similar for rule (II), involving a past receive and the *completion* of a current send. To illustrate, consider Figure 2, where each vertical arrow represents the execution of a thread (flowing from top to bottom). Both the top and the bottom diagrams depicts consecutive send `sd` and receive `rv` operations on a channel of capacity one—the operations are indexed as to show their order of execution.

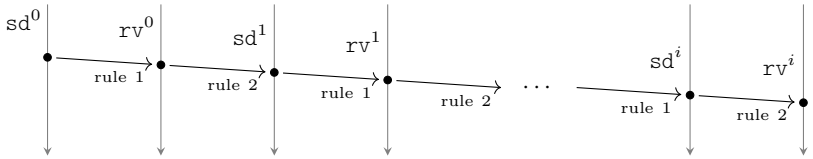
According to the Go memory model, channel operations are related as shown in Figure 2a. The operations are broken into two halves of a circle: the top is the operation and the bottom its completion. The arrows in the diagram represent the happens-before relation—arrows are labeled with the memory-model rule that justify their existence. According to rule (I), the 0^{th} send happens-before the completion of the 0^{th} receive—this relation is captured by the arrow starting at the top half-circle on the far left ($\texttt{sd}^0$) and ending at the bottom half-circle to the right (completion of $\texttt{rv}^0$). The next arrow establishes the happens-before relation between receive $\texttt{rv}^0$ and send $\texttt{sd}^1$ according to rule (II), and so forth. An

[2] The relation below can be derived by both program-order as well as by rule (I). Similar for the send and receive operations performed by $T1$.

$$c \leftarrow 0 \text{ by } T0 \quad \sqsubset_{hb} \quad \leftarrow c \text{ by } T0$$



(a) Depiction of rules (I) and (II) on a channel of capacity one.



(b) Alternate formulation of rules (I) and (II) with no distinction between an operation and its completion.

Fig. 2: The Go memory model specification, *every word counts*.

operation is related to its immediate predecessor. There is no chain of happens-before starting from the "distant" past. For example, although $\texttt{sd}^0$ is related to the completion of $\texttt{rv}^0$, and $\texttt{rv}^0$ is related to the completion of $\texttt{sd}^1$, it is not the case that $\texttt{sd}^0$ and $\texttt{sd}^1$ are related to each other.

Figure 2b captures an alternative formulation of the happens-before rules (I) and (II) where the word "completes" is left out. Sends and receives are not split into the operation and the operation's completion. Instead, sends and receives happen-before each other. This formulation leads to a chain starting at the very first send, and connecting every send and receive operation ever performed onto the channel. From an application programmer's perspective, this leads to an accumulation of happens-before information: after interacting with a channel, a goroutine's behavior is now dependent, not only on its communicating partner but, on every thread that has previously interacted with the channel. From the point of view of data races, this alternate formulation leads to over-synchronization.

The Go data-race detector's implementation matches the behavior of Figure 2b and, therefore, deviates from the Go memory model specification. Note that the over-synchronization on the part of the detector is not the result of careful deliberation, for example when false-negatives are accepted in exchange for lower runtime overheads. Rather, the implementation springs from an interpretation of synchronization from the perspective of locks rather than of channels. As will be discussed in Section 5, addressing this issue not only eliminates false-negatives but also yields lower runtime overhead.

Listing 1.1: Send.

```
func chansend(c *hchan,
              ep unsafe.Pointer,
              block bool,
              callerpc uintptr)
              bool {
  ...
  lock(&c.lock)
  ...
  if c.qcount < c.dataqsiz {
    qp := chanbuf(c, c.sendx)
    if raceenabled {
      raceacquire(qp)
      racerelease(qp)
    }

    typedmemmove(c.elemtype, qp, ep)


    c.sendx++
    if c.sendx == c.dataqsiz {
      c.sendx = 0
    }
    c.qcount++
    unlock(&c.lock)
    return true
  }
  ...
}
```

Listing 1.2: Receive.

```
func chanrecv(c *hchan,
              ep unsafe.Pointer,
              block bool)
              (selected,
              received bool) {
  ...
  lock(&c.lock)
  ...
  if c.qcount > 0 {
    qp := chanbuf(c, c.recvx)
    if raceenabled {
      raceacquire(qp)
      racerelease(qp)
    }
    if ep != nil {
      typedmemmove(c.elemtype, ep, qp)
    }
    typedmemclr(c.elemtype, qp)
    c.recvx++
    if c.recvx == c.dataqsiz {
      c.recvx = 0
    }
    c.qcount--
    unlock(&c.lock)
    return true, true
  }
  ...
}
```

Fig. 3: Snippets of Go's send and receive operations from `runtime/chan.go`.

4 The Go data-race detector

By adding `-race` to the command line, a Go program can be compiled and run with data-race detection enabled. The Go data-race detector is based on TSan, the Thread Sanitizer library [10]. The library is part of the LLVM infrastructure [12] and was originally designed to find races in C/C++11 applications.

When data-race detection is enabled, each thread becomes associated with an additional data structure. This data structure keeps track of the operations that are in happens-before from a thread's point of view. In most data-race detectors, including TSan, this data structure is a *vector clock* (VC) [11]. Vector-clocks offer a compact representation of the relative order of execution between threads. With this bookkeeping, data-race detectors are able to find synchronization issues in programs—where synchronization means the transfer of happens-before information between threads.

In the setting of locks, a thread performs an *acquire* operation in order to "learn" the happens-before information stored in a lock. By performing a *release* operation, a thread deposits its happens-before information onto a lock. In the setting of channels, we can think of happens-before as being transferred via sends and receives.

Figure 3 contains snippets from Go's implementation of the send and receive operations. Unsurprisingly, Go implements a channel of capacity C as an array

of length C. This array is contained in a struct called `hchan`. Struct member `sendx` is the index where a new message is to be deposited, while `recvx` is the index of the next message to be retrieved. Function `chanbuf` takes a channel struct and an index—the function returns a pointer to the channel's array at the given index. Note from lines 19 to 22 that a channel array is treated as a circular buffer.

When data-race detection is enabled, each channel array entry becomes associated with a vector-clock. Also, when detection is enabled, a send operation (Listing 1.1) generates calls to acquire and release—lines 11 to 14. The acquire causes the sender to "learn" the happens-before (HB) information associated with the channel entry at `c.sendx`. The release causes the thread's HB information to be stored back into that entry.[3] The receive operation is similarly implemented and shown in Listing 1.2.

In light of the implementation described above, we now revisit the message passing and mutual exclusion examples of Section 2. In the case of message passing, a thread sends a message onto a channel of capacity one, then another thread receives this message before accessing a shared resource—see Figure 1a. According to the data-race detector's implementation, the channel array entry at index 0 observes an *acquire* followed by *release* on behalf of the sender. Then, again, a sequence of *acquire* followed by *release* on behalf of the receiver. In effect, the happens-before information of the sender is transferred to the receiver: specifically, the *release* by $T0$ followed by the *acquire* by $T1$ places $T0$'s write operation in happens-before relation with respect to $T1$'s read operation. The message passing example of Figure 1 is thus deemed properly synchronized by the Go data-race detector.

We can reason about the mutual exclusion example of Figure 1b in similar terms. A thread sends onto a channel, accesses a shared resource, and then receives from the channel. With the receive operation, this thread deposits its happens-before information onto the channel—line 13 of Listing 1.2. The second thread then acquires this happens-before information when it sends onto the channel—line 12 of Listing 1.1. Again, the Go data-race detector's implementation correctly deems the example as properly synchronized.

4.1 The bug

Although the Go data-race detector behaves correctly on the message-passing and mutual-exclusion examples, the detector's implementation does not reflect the Go memory model specification. The acquire/release sequence performed on behalf of send and receive operations follows the typical lock usage. Channel

[3] In the implementation of the send operation, a message is moved from the sender's buffer to a receiver's buffer `ep` on line 16. The index `c.sendx` is incremented in line 19 and the increment wraps around based on the length of the array—lines 20 to 22. The number of elements in the array is incremented, the lock protecting the channel is unlocked and the function returns—lines 23 to 25.

programming is, however, different from lock programming. The current implementation of the detector leads to an accumulation of happens-before information associated with channel entries. This monotonic growth of happens-before information, however, is not prescribed by the Go memory model.

In the example that follows, we illustrate the mismatch between (1) the implementation of the data-race detector and (2) the memory model specification. We show how this mismatch leads to over-synchronization and the under reporting of data races.

$T0$	$T1$	$T2$
$c \leftarrow 0$	$c \leftarrow 0$	$\leftarrow c$
$z := 42$		$\texttt{load}\ z$
$\leftarrow c$		

Fig. 4: Example that highlights a mismatch between the Go memory model and the Go data-race detector implementation. (Capacity of channel c equals one).

Let c in Figure 4 be a channel of capacity one. The example is then a mix of mutual exclusion and message passing: $T0$ is using the channel as a lock in an attempt to protect its access to a shared variable,[4] and we can interpret $T1$ as using the same channel to communicate with $T2$.[5] Now, consider the interleaving in which $T0$ runs to completion, followed by $T1$, then $T2$—shown in Trace 7. Is the write to z by $T0$ in a race with the read of z by $T2$?

$$(c \leftarrow 0)_{T0} \quad (z := 42)_{T0} \quad (\leftarrow c)_{T0} \quad (c \leftarrow 0)_{T1} \quad (\leftarrow c)_{T2} \quad (\texttt{load}\ z)_{T2} \qquad (7)$$

The original Go data-race detector does not flag these accesses as racy:[6] $T0$ releases its happens-before (HB) by sending on the channel. This HB is stored in the vector-clock associated with c's 0^{th} array entry. The send by $T1$ performs an acquire followed by a release, at which point the VC associated with the entry contains the union of $T0$'s and $T1$'s happens-before. Finally, the receive by $T2$ performs an acquire and a release, causing $T2$ to learn the happens-before of $T0$ and $T1$. More formally, the data-race detector derives a happens-before relation between the write and the read as follows:

$$\begin{aligned}
z := 42 &\sqsubset_{hb} \leftarrow c \text{ by } T0 && \text{via program order} \\
\leftarrow c \text{ by } T0 &\sqsubset_{hb} c \leftarrow 0 \text{ by } T1 && \text{release by } T0\text{, acquire by } T1 \\
c \leftarrow 0 \text{ by } T1 &\sqsubset_{hb} \leftarrow c \text{ by } T2 && \text{release by } T1\text{, acquire by } T0 \\
\leftarrow c \text{ by } T2 &\sqsubset_{hb} \texttt{load}\ z && \text{via program order} \\
z := 42 &\sqsubset_{hb} \texttt{load}\ z && \text{via transitivity of HB}
\end{aligned}$$

[4] The send operation by $T0$ is analogous to acquire and the receive to release.

[5] Recall that the mutual exclusion and message passing patterns were introduced in Figure 1 and discussed in Section 2.

[6] GitHub issue `https://github.com/golang/go/issues/37355`

According to the Go memory model specification, however, the receive from c in $T0$ is not in happens-before relation to the send in $T1$. Instead, the receive is in happens-before relation to the *completion* of the send! Information about the write to z by thread $T0$ is transmitted to $T1$, but this information is only incorporated into $T1$ *after* the thread has transmitted its message to $T2$. Therefore, $T2$ does not receive $T0$'s happens-before information. In other words, according to the Go memory model, there is no chain of happens-before connecting $T0$ to $T2$. The trace captured by equation (7) is thus racy, with the race depicted in Figure 5. Specifically, the race is captured by the absence of a path between the write to z in $T0$ and the load of z in $T2$.

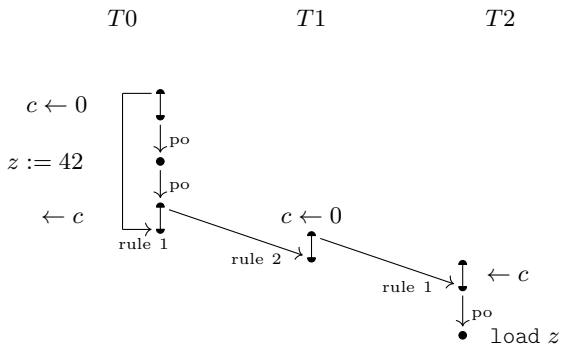


Fig. 5: Partial order on events according to the Go memory model. The HB relation is represented by arrows labeled with the Go memory model rule justifying the arrow's existence. The top part of the half-circle corresponds to a channel operation and the bottom to its completion.

The Go memory model calls for a mix between channel communication as described by Lamport [11] and lock programming. Lamport [11] was studying distributed systems in the absence of shared memory: the shared resources were the channels themselves, and the absence of races (of channel races) was related to determinism. In contrast, Go employs channels as a primitive for synchronizing *memory accesses*. In Go, some happens-before relations are forged solely between communicating partners—these relations are derived from rule (I), which is also present in [11]. Similar to lock programming, some happens-before relations are the result of an accumulation of effects from past interactions on a channel. This accumulation occurs when we incorporate rule (II), which is not present in [11]. So, while the language favors a discipline where an implicit notion of variable ownership is transferred via direct communication, as prescribed by rule (I), by incorporating rule (II), Go also supports the use of channels as locks.

5 The fix: capturing the semantics of channels

The repair to the Go data-race detector's deviation from the memory model specification comes from acknowledging that a primitive, different from acquire and release, can better fit the semantics of synchronization via channel communication. We propose the primitive depicted in Figure 6, which we call *release-acquire-exchange* or `rea`. Let $\mathbb{C}_t$ be the happens-before information of thread t, m be the channel entry where a message will be deposited or retrieved, and $\mathbb{L}_m$ be the happens-before information associated with m. The primitive is implemented with a thread releasing onto a place-holder and then acquiring from $\mathbb{L}_m$. The happens-before in $\mathbb{L}_m$ is then overwritten with the HB information from the place-holder.[7] We added this new synchronization primitive into TSan, the data-race detection library that powers the Go data-race detector. With the new primitive in place, changes to the Go sources became trivial:[8] it involved changing sequences of acquire/release calls with a call to *release-acquire-exchange*.

$$\frac{\mathbb{C}'_t := \mathbb{C}_t \sqcup \mathbb{L}_m \qquad \mathbb{L}'_m := \mathbb{C}_t}{(\mathbb{C}_t, \mathbb{L}_m) \Rightarrow^{\mathtt{rea}(t,m)} (\mathbb{C}'_t, \mathbb{L}'_m)}$$

Fig. 6: Semantics of "release-acquire-exchange," a new primitive added to TSan.

Given the addition of `rea` into TSan, let us revisit trace (7). While the original implementation of the Go data-race detector did not flag this trace as racy, the updated version does. Given the detector's updated implementation, we can reason about the race as follows. Let $\mathbb{C}_{T0}$, $\mathbb{C}_{T1}$, and $\mathbb{C}_{T2}$ be data-structures storing happens-before information of threads $T0$, $T1$, and $T2$. Let $\mathbb{L}_{c[0]}$ be the happens-before associated with the 0^{th} array entry of channel c. We denote the write event to z as $!z$ and, for simplicity, we represent happens-before information as a set of memory events. The race-detector state is then the tuple $[\mathbb{C}_{T0}, \mathbb{C}_{T1}, \mathbb{C}_{T2}, \mathbb{L}_{c[0]}]$, with initial state $[\{\}, \{\}, \{\}, \{\}]$. The data-race detector

[7] The place-holder is a variable local to a function in TSan, as opposed to an extra memory region allocated in Go.

[8] Changes in Go `https://golang.org/cl/220419` and TSan `https://reviews.llvm.org/D76322`

performs the following transitions as the program executes:

$$
\begin{array}{llll}
\mathbb{C}_{T0} & \mathbb{C}_{T1} & \mathbb{C}_{T2} & \mathbb{L}_{c[0]} \\
[\ \{\}, & \{\}, & \{\}, & \{\}\ ] \Rightarrow^{(c \leftarrow 0)_{T0}} \\
[\ \{\}, & \{\}, & \{\}, & \{\}\ ] \Rightarrow^{(z:=42)_{T0}} \quad (8) \\
[\{!z\}, & \{\}, & \{\}, & \{\}\ ] \Rightarrow^{(\leftarrow c)_{T0}} \quad (9) \\
[\{!z\}, & \{\}, & \{\}, & \{!z\}] \Rightarrow^{(c \leftarrow 0)_{T1}} \quad (10) \\
[\{!z\}, & \{!z\}, & \{\}, & \{\}\ ] \Rightarrow^{(\leftarrow c)_{T2}} \quad (11) \\
[\{!z\}, & \{!z\}, & \{\}, & \{\}\ ] \Rightarrow^{(\texttt{loadz})_{T2}} \quad (12)
\end{array}
$$

The write to z by $T0$ places $!z$ into $\mathbb{C}_{T0}$—transition from equation (8) to (9). Sends and receives are interpreted according to their formal semantics in Figure 6. The receive by $T0$ places the write event into the channel-entry's happens-before—equations (9) and (10). The send by $T1$ places the write event into the thread's happens-before and overwrites the channel-entry's happens-before with the empty set—equations (10) and (11). The receive by $T2$ retrieves the empty happens-before information—equations (11) and (12). Therefore, at the time $T2$ loads from the shared variable, the write to z by $T0$ is not in happens-before with respect to $T2$. In conclusion, the execution is racy.

Note that the fix to the Go data-race detector does not invalidate the use of channels as locks. Without loss of generality, let the trace below be an execution of the mutual exclusion example of Figure 1b.

$$(c \leftarrow 0)_{T0} \quad (z := 42)_{T0} \quad (\leftarrow c)_{T0} \quad (c \leftarrow 0)_{T1} \quad (z := 43)_{T1} \quad (\leftarrow c)_{T1} \quad (13)$$

The detector's execution, from initial state $[\mathbb{C}_{T0}, \mathbb{C}_{T1}, \mathbb{L}_{c[0]}] = [\{\}, \{\}, \{\}]$, is

$$
\begin{array}{lll}
\mathbb{C}_{T0} & \mathbb{C}_{T1} & \mathbb{L}_{c[0]} \\
[\ \{\}, & \{\}, & \{\}\ ] \Rightarrow^{(c \leftarrow 0)_{T0}} \\
[\ \{\}, & \{\}, & \{\}\ ] \Rightarrow^{(z:=42)_{T0}} \\
[\{!z\}, & \{\}, & \{\}\ ] \Rightarrow^{(\leftarrow c)_{T0}} \\
[\{!z\}, & \{\}, & \{!z\}] \Rightarrow^{(c \leftarrow 0)_{T1}} \\
[\{!z\}, & \{!z\}, & \{\}\ ]
\end{array}
$$

Note that the event $!z$ capturing the write by $T0$ is contained in $\mathbb{C}_{T1}$ before $T1$ attempts to write to z. In other words, the writes are ordered by happens-before and the execution is properly synchronized. Thus, the answer to the question raised at the end of Section 2, *"is it possible to support the use of channels as locks (as in the mutex example) and still avoid over-synchronization?"* is yes.

We implement the new synchronization primitive `rea` in TSan with one pass, as opposed to two passes, over the data-structure storing happens-before information. Therefore, the updated data-race detector implementation provides

better performance than the original sequence of *acquire* followed by *release*. Another consequence of our fix is a potential reduction in the memory footprint associated with data-race detection. This savings comes from the fact that vector clocks associated with channel entries no longer observe as large of an accumulation of happens-before information—this point was previously touched upon in Section 3, Figure 2. We include a short experimental evaluation in Appendix A.

5.1 From small-step operational semantics to `rea`

The *release-acquire-exchange* primitive comes from our previous formalizations of Go channels. It is conceptually useful to distinguish between happens-before information transmitted on behalf of rule (I) versus (II). In our earlier formalization of a memory model inspired by the Go specification [4], a channel c is split into two: a forward and a backward one. The forward channel c_f holds messages and thread-local information to be transmitted, as prescribed by rule (I), from a sender to its corresponding receiver. The backward channel c_b, which flows from a prior receiver to a current sender, captures rule (II) of the memory model.[9]

In [4], threads or goroutines $p\langle\sigma, t\rangle$ have a unique identifier p, contain thread-local information σ, and a term t corresponding to the program under execution. When it comes to data-race detection, thread-local information σ is composed of *happens-before* data. This data could be stored in a vector-clock or, more simply, it could be a set of read- and write-events that are in happens-before with respect to the thread. Synchronization, therefore, entails the exchange of thread-local information σ via channel communication.

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

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

Fig. 7: Send and receive reduction rules in the calculus of [4].

Configurations consist of the parallel composition of goroutines, memory events, and channels. The semantics of [4] is operational. We give the reduction rules for sends and receives in Figure 7—other rules are omitted and can

[9] As noted in [5], "the interplay between forward and backward channels can also be understood as a form of flow control. Entries in the backward channel's queue are not values deposited by threads. Instead, they can be seen as tickets that grant senders a free slot in the communication channel."

be found in the original paper. The `let`-construct in R-Rec is a binder for the local variable r in a term t. In the case of R-Rec, the `let`-construct allows t to refer to the value obtained when receiving from a channel.

According to reduction rule R-Send, when a thread sends a value v, the thread's local state σ is placed on the forward channel alongside v. The transmission of σ models rule (I) of the Go memory model: a receiver who receives (v, σ) will learn about the sender's actions up to the given send operation. The rule R-Send captures the placement of the message (v, σ) onto the forward channel as follows: if q_2 is the content of the forward channel before transmission, $(v, \sigma) :: q_2$ are the contents after.

Besides transmitting, a sender also learns HB information in accordance to rule (II). Precisely, the $(k + C)^{th}$ sender obtains, from the backward channel, thread-local state from the k^{th} receiver. This is captured by the update $\sigma' = \sigma + \sigma''$ with the state σ'' coming from the backward channel. Thus, if the contents of the backward channel were $q_1 :: \sigma''$ before the send, the channel is left with q_1 after the send. Note that the update to the sender state occurs *on completion* of the send operation: the update "occurs after" the sender has deposited its message onto the forward channel—concretely, the send transmits the thread state σ as opposed to the updated thread state σ'.

When receiving, a goroutine obtains a value v as well as a state σ'' from a sender. As dictated by rule (I), the receiver updates its state given the corresponding sender state: $\sigma' = \sigma + \sigma''$. The sender also deposits its state onto the backward channel. Similar to R-Send, the original thread state σ is deposited as opposed to the updated thread state σ'.

For both reduction rules R-Send and R-Rec, local thread state σ is deposited onto a channel as opposed to the update thread state σ'. This discipline creates a distinction between an operation and its completion. In effect, the reduction rules do not cause the over-synchronization observed by the Go data-race detector.

5.2 Why not *acquire* and *release*?

The formalization in [4] speaks of synchronization in terms of channel communication. Since TSan operates at the level of locks, we might be tempted to implement the reduction rules with acquire and release operations. The reduction rule R-Send could be implemented with a thread releasing its happens-before information into the forward queue, and *then* acquiring happens-before information from the backward queue. Similarly, R-Rec can be implemented with a release to the backward queue, followed by an acquire from the forward queue.

One invariant of the semantics in [4] is that the number of elements in the forward and backward queues equals the capacity of the channel. Since a thread must first release its HB into the channel before acquiring from the channel, there would be more than C entries in the queues while a send or receive is in-flight. In fact, when using acquire and release operations as primitives, the Go data-race detector would need to allocate an array of length $C + 2$ for a channel of capacity C. Given such an array, sends and receives can be implemented

with acquire/release operations as shown on Listings 1.3 and 1.4. Recall that `c.sendx` and `c.recvx` are the indices into the array where the next message is to be deposited and retrieved respectively. Recall also that `chanbuf` returns a pointer to a channel's array at a given index.

Listing 1.3: Implementation of send.

```
1  idx := c.sendx+1
2  if idx == C+2
3    idx = 0
4  qf := chanbuf(c, c.sendx)
5  qb := chanbuf(c, idx)
6  racerelease(qf)
7  raceacquire(qb)
```

Listing 1.4: Implementation of receive.

```
1  idx := c.recvx−1
2  if c.recvx == 0
3    idx = C+1
4  qf := chanbuf(c, c.recvx)
5  qb := chanbuf(c, idx)
6  racerelease(qb)
7  raceacquire(qf)
```

Although correct, there are major downsides to the solution of Listings 1.3 and 1.4. First, it requires additional memory allocation. Second, because the Go runtime expects a channel of capacity C to be implemented with an array of length C, the solution would require intrusive changes to the implementation. Third, from a timing perspective, in order to implement a single channel operation, the solution performs two passes over the data-structure storing happens-before information—we want a solution that performs less passes.

Compared to *acquire* and *release*, the *release-acquire-exchange* primitive requires no additional allocation in Go, involves minimal changes to the Go runtime, and has lower overhead.

6 Lessons learned

When we started looking at TSan's source code, our goal was to improve Go's data-race detector by expressing synchronization in terms of channels as opposed to locks [5]. We began reading the source code of Go and TSan in November 2019. In January 2020, we started experimenting by compiling the projects from source and making modifications in order to gain experience and intuition. This tinkering lead us to find, in early and mid February, a small Go compiler bug[10] and a small performance bug in TSan.[11] Shortly after, around late February, we found the bug described in this paper.

Given our experience formalizing the calculus in [4], we could see similarities between our reduction rules and the Go implementation.[12] The implementations of send and receive, however, stood out. The bug was thus found by inspection. We created a test (Figure 4) to showcase what we believed was discrepancy

[10] `https://github.com/golang/go/issues/37012`

[11] `https://reviews.llvm.org/D74831`

[12] For example, the closing of channels in both [4] and in Go cause happens-before information to be deposited onto the channel, regardless of whether the channel is full.

between the detector and the memory model. From there, we filed an issue on GitHub and started interacting with the Go community. With this interaction, which went until May, an initial patch was iteratively improved until being accepted for future release.

In this section, we collect insights drawn from our experience in both (1) formalizing aspects of the Go programming language and (2) interacting with the TSan and Go communities.

Models do not have to be right, they have to be useful. In [4], we developed a memory model based on the Go specification. Before embarking on studying the Go source code, we found ourselves at cross-roads. Since our model is not as relaxed as Go's, more theoretical research remained to be done. We pondered whether to continue working on formalizations or whether to investigate how the current model fits the "real world." Both avenues are interesting to us. By taking, for now, the second avenue, we learned that *models do not have to be right, they have to be useful.* Our memory model formalization in [4] *is not* the memory model of Go, but it is close enough to allow us to reason about Go and its implementation.

Mind the gap. In one hand, we have the concept of a data-race according to the synchronization rules of the Go memory model specification. The specification is expressed in English. On the other hand, we have the Go data-race detector implementation, with thousands of lines of code spawning different projects and repositories and involving at least three languages (Go, C/C++, and assembly). These are two ends of a spectrum. Our model was useful, in part, because of where it sits in this spectrum. When developing the model in [4], we followed the English text of the Go memory model specification very closely. Our model, however, is expressed in structural operational semantics—its rules form an executable implementation. Our calculus, therefore, forms a bridge between source code and the specification expressed in natural language.[13]

Bad news is good news. The effort in formalizing and proving a nontrivial property of a software system is often high. Before finding the issue described in this paper, we had been working on formalisms related to Go for over two years. This high barrier of entrance is both good and bad. It is good, less obviously so, because it opens opportunities for collaboration between industry and academia. While industry excels at delivering software, academia can provide artifacts, such as formalisms and proofs, which are still not as commonly produced in industry.[14]

[13] Our observation about the representational different between specification and implementation is not new. The idea of bridging specification and implementation has been tackled by many fronts, for example [1].

[14] Because of stigma, the "formal" qualifier has been de-emphasized when disseminating formal methods in industry [14]. This stance has shifted dramatically [3].

7 Conclusion

The bug described in this paper evaded skilled developers for about six years, nearly since the data-race detector was bolted onto the Go runtime. In this paper, we share how formal methods played an integral role in bringing the issue to light, and giving it closure.

Acknowledgments. I would like to thank Martin Steffen for his feedback on this manuscript, Dmitry Vyukov for his feedback and guidance on incorporating the proposed changes into Go and TSan, Keith Randall for rebuilding the TSan library files that ship with Go, and everyone who has given constructive feedback during the code review process. I would also like to thank the reviewers for their comments on this manuscript, and reviewer 2 in particular.

Bibliography

[1] Back, R. and von Wright, J. (1998). *Refinement Calculus – A Systematic Introduction*. Graduate Texts in Computer Science. Springer.

[2] Brewer, E. A. (2015). Kubernetes and the path to cloud native. In Ghandeharizadeh, S., Barahmand, S., Balazinska, M., and Freedman, M. J., editors, *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, page 167. ACM.

[3] Cook, B. (2018). Formal reasoning about the security of amazon web services. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 38–47. Springer.

[4] Fava, D., Steffen, M., and Stolz, V. (2019). Operational semantics of a weak memory model with channel synchronization. *Journal of Logical and Algebraic Methods in Programming*, 103:1 – 30. An extended version of the FM'18 publication with the same title.

[5] Fava, D. S. and Steffen, M. (2020). Ready, set, Go! Data-race detection and the Go language. *Science of Computer Programming*, 195:102473.

[6] Go channels (2020). Channel types. `https://golang.org/ref/spec#Channel_types`. Version of March 19, 2020.

[7] Go developer survey (2020). Go developer survey 2019 results. `https://blog.golang.org/survey2019-results`.

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

[9] Go share memory by communicating (2010). The Go blog. `https://blog.go-lang.org/codelab-share`.

[10] google.thread.sanitizer (2015). `https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm`.

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

[12] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

[13] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[14] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73.

A Memory footprint

Here we illustrate how our fix to the Go data-race detector leads to a smaller memory foot-print. Consider an in-place parallel sorting algorithm where an array is recursively split, up to some depth, in approximately half. Each region of the array is assigned to a thread for sorting. When a thread completes sorting, it signals its parent. The parent merges, in-place, the consecutive array regions previously assigned to its children.

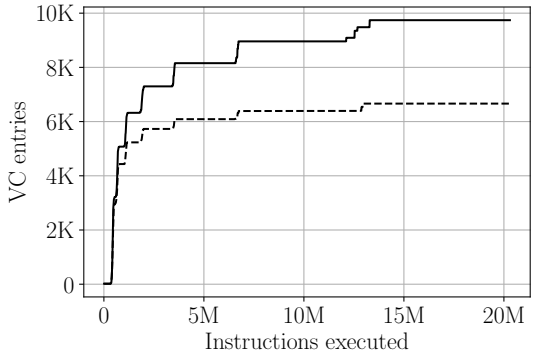


Fig. 8: Number of VC entries associated with channels during the execution of an in-place parallel sorting algorithm: before (solid line) and after (dashed line) the introduction of release-acquire-exchange.

We tracked the number of entries in the vector-clocks associated with channel array entries. Measurements of the number of VC entries were taken multiple times during the program's execution. For ease of collecting and plotting the data, we modified TSan to call out to a reference data-race detector implemented in Python.[15][16] Figure 8 shows the number of VC entries before and after the fix to the data-race detector—meaning, with a race detector that performed an *acquire* followed by *release* versus a race detector that implements the *release-acquire-exchange* primitive. The x-axis is the number of instructions executed, the y-axis is the number of vector-clock entries consumed so far in the execution. As the program makes progress, more entries accumulate in the vector-clocks associated with channel entries. This accumulation is much more accentuated before the fix to the data-race detector. In fact, for this workload, the fix lead to larger than 30% reduction in the number of VC entries after 12.5M instructions were executed.

[15] `https://github.com/dfava/paper.go.mm.drd`

[16] Because of differences in how vector clocks are allocated and managed, the memory gains reported by the reference data-race detector may be different from TSan's.