Ready, set, Go!

Data-race detection and the Go language[*]

Daniel Schnetzer Fava
danielsf@ifi.uio.no

Martin Steffen
msteffen@ifi.uio.no

Dept. of Informatics, University of Oslo

Abstract. Data races are often discussed in the context of lock acquisition and release, with race-detection algorithms routinely relying on *vector clocks* as a means of capturing the relative ordering of events from different threads. In this paper, we present a data-race detector for a language with channel communication as its sole synchronization primitive, and provide a semantics directly tied to the *happens-before* relation, thus forging the notion of vector clocks.

Keywords: concurrency, synchronization, message passing, data-race detection, golang

1 Introduction

One way of dealing with complexity is by partitioning a system into cooperating sub-components. When these subcomponents compete for resources, coordination becomes a prominent goal. One common programming paradigm is to have threads cooperating around a pool of shared memory. In this case, coordination involves, for example, avoiding conflicting accesses to memory. Two concurrent accesses constitute a *data race* if they reference the same memory location and at least one of the accesses is a write. Because data races can lead to counter intuitive behavior, it is important to detect them.

The problem of data-race detection in shared memory systems is well studied in the context of lock acquisition and release. When it comes to message passing, the problem of concurrent accesses to *channels*, in the absence of shared memory, is also well studied—the goal, in these cases, is to achieve determinism rather than race-freedom [6, 7, 38]. What is less prominent in the race-detection literature is the study of channel communication as the synchronization primitive for shared memory systems. In this paper, we present exactly that; a dynamic data-race detector for a language in the style of Go, featuring channel communication as means of coordinating accesses to shared memory.

We fix the syntax of our calculus in Section 3 and present a corresponding operational semantics. The configurations of the semantics keep track of memory events (i.e. of read and write accesses to shared variables) such that the semantics can be used to detect races. A proper book-keeping of events also involves tracking *happens-before* information. In the absence of a global clock, the happens-before relation is a vehicle

---

for reasoning about the relative order of execution of different threads [18]. We describe the race detection task and present a framework, called GRACE [9], that is based on what we call *happens-before sets*. Different from other race detectors, which often employ vector clocks (VCs) as a mechanism for capturing the happen-before relation, we tie our formalization more closely to the concept of happens-before. The proposed approach, based on what we call happens-before sets, allows for garbage collection of "stale" memory access information that would otherwise be tracked. Although, in the worst case, the proposed detector requires a larger foot-print when compared to VC-based implementations, we conjecture the existence of a hybrid approach that can offer benefits from both worlds.

Our race detector is built upon a previous result [10], where we formalize a weak memory model inspired by the Go specification [12]. The core of the paper was a proof of the DRF-SC guarantee, meaning, we proved that the proposed relaxed memory model behaves Sequentially Consistently (SC) when running Data-Race Free (DRF) programs. The proof hinges on the fact that, in the absence of races, all threads agree on the contents of memory. The scaffolding used in the proof contains the ingredients for the race detector presented in this paper. We should point out, however, that the operational semantics presented here and used for race detection is *not* a weak semantics.[1] Apart from the additional information for race detection, the semantics is "strong" in that it formalizes a memory guaranteeing *sequential consistency*. To focus on a form of strong memory is not a limitation. Since we have established that a corresponding weak semantics enjoys the crucial DRF-SC property [10], the strong and weak semantics agree up to the first encountered race condition. Given that even racy program behaves sequentially consistently up to the point in which the first data-race is encountered, a complete race detector can safely operate under the assumption of sequential consistency.

The remainder of the paper is organized as follows. Section 2 presents background information on data races and synchronization via message passing that are directly related to the formalization of our approach to race detection. Section 3 formalizes race detection in the context of channel communication as sole synchronization mechanism. We turn our attention to the issue of efficiency in Section 4. Section 5 gives a detailed comparison of our algorithm and VC-based algorithms for the acquire-release semantics. Section 6 puts our work in the perspective of trace theory. Section 7 examines related work. Section 8 provides a conclusion and touches on future work.

2   Background

---

[1] Note that while the mentioned semantics of [10] differs from the one presented here, both share some commonalities. Both representations are based on appropriately recording information of previous read and write events in their run-time configuration. In both versions, a crucial ingredient of the book-keeping is connecting events in happens-before relation. The purpose of the book-keeping of events, however, is different: in [10], the happens-before relation serves to operationally formalize the weak memory model (corresponding roughly to PSO) in the presence of channel communication. In the current paper, the same relation serves to obtain a race detector. Both versions of the semantics are connected by the DRF-SC result, as mentioned.

Read and write conflicts. Memory accesses conflict if they target the same location and at least one of the accesses is a *write*—there are no read-read conflicts. A data race constitutes of conflicting accesses that are unsynchronized.

Listing 1.1: Program with race condition. [12]

```
var a string

func main() {
        go func() { a = "hello" }()
        print(a)
}
```

Take the Go code of Listing 1.1 as an example. There, the main function invokes an anonymous function; this anonymous function sets the global variable "a" to "hello". Note, however, that the call is prepended with the keyword **go**. When this keyword is present in a function invocation, Go spawns a new thread (or goroutine), and the caller continues execution without waiting for the callee to return. The main and the anonymous functions access the same shared variable in a conflicting manner (*i.e.* one of the accesses is a write). Since both the main and the anonymous functions run in parallel and no synchronization is used (as evidenced by the lack of channel communication), the two accesses are also concurrent. This allows us to conclude that this program has a race.

A data race *manifests* itself when an execution step is immediately followed by another and the two steps are conflicting. This definition is the closest one can get to a notion of simultaneity in an operational semantics, where memory interactions are modeled as instantaneous atomic steps. While manifest races are obvious and easy to account for, races in general can involve accesses that are arbitrarily far apart in a linear execution. A "memory-less" detector can fail to report races, for example non-manifest races, that could otherwise be flagged by more sophisticated race detectors. The ability to flag non-manifest data-races is correlated with the amount of information kept and the length in which this information is kept for. In general, recording more information and storing it for longer leads to higher degrees of "completeness" at the expense of higher run-time overheads.[2]

We break down the notions of read-write and write-write conflicts into a more fine-grained distinction. Inspired by the notion of data hazards in the computer architecture literature, we break down read-write conflicts into read-after-write (RaW) and write-after-read (WaR) conflicts. To keep consistent with this nomenclature, we refer to write-write conflicts as write-after-write (WaW).[3] Going back to the example in Listing 1.1,

---

[2] It should go without saying that observing one execution as being race free is not enough to assert race-freedom of the program, even if one has observed a complete trace of a terminating run of a program. Completeness can at best be expected with respect to alternative schedules or linearizations of a given execution.

[3] The mentioned "temporal" ordering and the use of the word "after" refers to the occurrence of events in the trace or execution of the running program. It is incorrect to conflate the concept

there are two possible executions: one in which the spawned goroutine writes "hello" to the shared variable after the main function prints it, and another execution in which the print occurs after the writing of the variable. The first execution illustrates a write-after-read race, while the first illustrates a read-after-write. Note that this example does not contain a write-after-write race.

We make the distinction between the detection of after-write races and the detection of write-after-read ones. As we will see in Section 3.3, the detection of after-write races can be done with little overhead. The detection of after-read, however, cannot.

When reading or writing a variable, it must be checked that conflicting accesses *happened-before* the current access. The check must happen from the perspective of the thread attempting the access. In other words, the question of whether an event occurred in the "definite past" (i.e., whether an event is in happened-before relation with "now") is *thread-local*; threads can have different views on whether an event belongs to the past. This thread-local nature is less surprising than it may sound: if one threads executes two steps in sequence, the second step can safely assume that the first has taken effect; after all, that is what the programmer must have intended by sequentially composing instructions in the given program order. Such guarantees hold locally, which is to say that the semantics *respects program order within a thread*. It is possible, however, for steps to not take effect in program order. A compiler or hardware may rearrange instructions, and it often does so in practice. What must remain true is that these reorderings cannot be observable from the perspective of a single thread. When it comes to more than one thread, however, agreement on what constitutes the past cannot be achieved without synchronization. Synchronization and consensus are integrally related.[4] Specifically, given a thread t, events from a different thread t' are not in the past of t unless synchronization forces them to be.

Synchronization via bounded channels. In the calculus presented here, channel communication is the only way in which threads synchronize. Channels can be created dynamically and closed; they are also first-class data, which means channel identifiers can be passed as arguments, stored in variables, and sent over channels. Send and receive operations are central to synchronization. Clearly, a receive statement is synchronizing in that it is potentially blocking: a thread blocks when attempting to receive from an empty channel until, if ever, a value is made available by a sender. Since channels here are bounded, there is also potential for blocking when sending, namely, when attempting to send on a channel that is full.

We can use a channel c to eliminate the data race in Listing 1.1 as follows: the anonymous function sends a message to communicate that the shared variable has been

---

of happens-before with the ordering of occurrences in a trace. For instance, in a RaW situation, the read step occurs after a write in an execution, i.e., the read is mentioned after the write in the linearization. This order of occurrence does *not* mean, however, that the read happens-after the write or, conversely, the write happens-before the read. Actually, for a RaW race (same as for the other kinds of races), the read occurs after the write but the accesses are *concurrent*, which means that they are *unordered* as far as the happens-before relation is concerned.

[4] In the context of channel communication and weak memory, the connection between synchronization and consensus is discussed in a precise manner in our previous work; see the *consensus lemmas* of [10].

set. Meanwhile, the main thread receives from the channel before printing the shared variable.

Listing 1.2: Repaired program.

```
var a string
var c = make(chan bool, 1);

func main() {
        go func() { a = "hello";  c <- true }()
        <- c
        print(a)
}
```

The happens-before memory model stipulates, not surprisingly, a causal relationship between the communicating partners [12]:

A send on c happens-before the corresponding receive from c completes. (1)

Given that channels have finite capacity, a thread remains blocked when sending on a full channel until, if ever, another process frees a slot in the channel's buffer. In other words, the sender is blocked until another thread receives from the channel. Correspondingly, there is a happens-before relationship between a receive and a subsequent send on a channel with capacity k [12]:

The ith receive from c happens-before the $(i+k)^{th}$ send on c completes. (2)

Interestingly, because of this rule, a causal connection is forged between the sender and *some previous* receiver who is otherwise unrelated to the current send operation. When multiple senders and receivers share a channel, rule (2) implies that it is possible for two threads to become related (via happens-before) without ever directly exchanging a message.[5]

The indirect relation between a sender and a prior receiver, postulated by rule (2), allows channels to be used as locks. In fact, free and taken binary locks are analogous to empty and full channels of capacity one. A process takes and releases locks for the purpose of synchronization (such as assuring mutually exclusive access to shared data) without being aware of "synchronization partners." In the (mis-)use of channels as locks, there is also no inter-process communication. Instead, a process "communicates" with itself: In a proper lock protocol, the process holding a lock (i.e. having performed a send onto a channel) is the only one supposed to release the lock (i.e. performing the corresponding receive). Thus, a process using a channel as lock receives its own previously sent message—there is no direct inter-process exchange. Note, however, synchronization still occurs: subsequent accesses to a critical region are denied

---

[5] Communication means sending a message to or receiving a message from a channel; messages are not addressed to or received from specific threads. Thus, sharing the channel by performing sends and receives does not necessarily make two threads "communication partners." Two threads are partners when one receives a message deposited by the other.

by sending onto a channel and making it full. See Section 3.5.2 for a more technical elaboration.

To establish a happens-before relation between sends and receives, note the distinction, between a channel operation and its *completion* in the formulation of rules (1) and (2). The order of events in a concurrent system is partial; not only that, it is strictly partial since we don't think of an event as happening-before itself. A strict partial order is an irreflexive, transitive, and asymmetric relation. In the case of synchronous channels, if we were to ignore the distinction between an event and its completion, according to rule (1), a send would then happen-before its corresponding receive, and, according to rule (2), the receive would happen-before the send. This cycle breaks asymmetry. Asymmetry can be repaired by interpreting a send/receive pair on a synchronous channel as a single operation; indeed, it can be interpreted as a *rendezvous*.

The distinction between a channel operation and its completion is arguably more impactful when it comes to buffered channels. For one, it prevents sends from being in happens-before with other sends, and receives from being in happens-before with other receives. To illustrate, let $\mathtt{sd}^i$ and $\mathtt{rv}^i$ represent the ith send and receive on a channel. If we remove from rules (1) and (2) the distinction between an operation and its completion, the ith receive would then happens-before the $(i+k)^{th}$ send—based on rule (2)—and the $(i+k)^{th}$ send would happens-before the $(i+k)^{th}$ receive—based on rule (1):

$$\mathtt{rv}^i \ \to_{\mathsf{hb}} \ \mathtt{sd}^{i+k} \ \to_{\mathsf{hb}} \ \mathtt{rv}^{i+k}$$

By transitivity of the happens-before relation, we would then conclude that the ith receive happens-before the $(i+k)^{th}$ receive, which would happen-before the $(i+2k)^{th}$ receive and so on. As a consequence, a receive operation would have a lingering effect through-out the execution of the program—similarly for send operations. This accumulation of effects can be counter intuitive for the application programmer, who would be forced to reason about arbitrarily long histories.

3   Data-race detection

We start in Section 3.1 by presenting the abstract syntax of our calculus and, in Section 3.2, an overview of the operational semantics used for data-race detection. The race detector itself is introduced incrementally. We start in Section 3.3 with a simple detector that has a small footprint but that is limited to detecting after-write races. We build onto this first iteration of the detector in Section 3.4, making it capable of detecting after-write as well as after-read races. The detector's operation is illustrated by examples in Section 3.5. Later, in Section 4, we turn to the issue of efficiency and introduce "garbage collection" as a mean to reduce the detector's footprint. These race detectors can be seen as augmented versions of an underlying semantics without additional book-keeping related to race checking. This "undecorated" semantics, including the definition of internal steps and a notion of structural congruence, can be found in Appendix A.

3.1 A calculus with shared variables and channel communication

We formalize our ideas in terms of an idealized language shown in Figure 1 and inspired by the Go programming language. The syntax is basically unchanged from [10].

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

Fig. 1: Abstract syntax

Values v can be of two forms: r denotes local variables or registers; n is used to denote references or names in general and, in specific, p for processes or goroutines, m for memory events, and c for channel names. We do not explicitly list values such as the unit value, booleans, integers, etc. We also omit compound local expressions like $e_1 + e_2$. Shared variables are denoted by x, z, etc., `load z` represents reading the shared variable z into the thread, and $z := v$ denotes writing to z. References are dynamically created. A new channel is created by `make (chan `T, v`)`, where T represents the type of values carried by the channel and v a non-negative integer specifying the channel's capacity. Sending a value v over a channel c and receiving a value as input from a channel are denoted respectively as $c \leftarrow v$ and $\leftarrow c$. 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.

Starting a new asynchronous activity, called goroutine in Go, is done using the `go`-keyword. In Go, the `go`-statement is applied to function calls only. We omit function calls, asynchronous or otherwise, as they are orthogonal to the memory model's formalization. The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches (or communication clauses in Go-terminology). These branches act as guarded threads. General expressions in Go can serve as guards. Our syntax requires that only communication statements (i.e., channel sending and receiving) and the `default`-keyword can serve as guards. This does not reduce expressivity and corresponds to an A-normal form representation [32]. At most one branch is guarded by `default` in each select-statement. The same channel can be mentioned in more than one guard. "Mixed choices" [27, 28] are also allowed, meaning that sending- and receiving-guards can both be used in the same select-statement. We use `stop` as syntactic sugar for the empty select statement; it represents a permanently blocked thread. The `stop`-thread is also the only way to syntactically "terminate" a thread, meaning that it is the only element of t without syntactic sub-terms.

The `let`-construct `let `$r = e$` in `t combines sequential composition and scoping 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 acts as a binder for variable r in t. When r does not occur free in t, `let` boils down to *sequential composition* and, therefore, is more conveniently written with a semicolon. See also Figure 15 in the appendix for syntactic sugar.

3.2 Overview of the operational semantics

To capture the notion of ordering of events between threads, an otherwise unadorned operational semantics (equation (7)) is equipped with additional information: each thread and memory location tracks the events it is aware of as having happened-before—see the happens-before set E_{hb} in the run-time configurations of equation (3) and (4), this set is present in terms corresponding to threads, $p\langle E_{hb}, t\rangle$, as well as memory locations, $(\!|E_{hb}, z{:=}v|\!)$ or $m(\!|E^r_{hb}, z{:=}v|\!)$. Depending on the capabilities of the race detector, slightly different information is tracked as having happened-before (i.e. stored in a happens-before set).

3.2.1 After-write races When detecting after-write races (i.e. RaW and WaW), in order to know whether a subsequent access to the same variable occurs without proper synchronization, one has to remember additional information concerning past write-events. Specifically, it must be checked that all write events to the same variable *happened-before* the current access. The happens-before set is then used to store information pertaining to write events; read events are not tracked. Also, terms representing a memory location have a different shape when compared to the undecorated semantics. In the undecorated semantics, the content v of a variable z is written as a pair $(\!|z{:=}v|\!)$. When after-write races come into play, it is not enough to store the last value written to each variable; we also need to identify write events associated with the variable. Thus, an entry in memory takes the form $(\!|E_{hb}, z{:=}v|\!)$ where E_{hb} holds identifiers m, m', etc. that uniquely identify write events to z—contrast the run-time configurations in equation (7) and (3). The number of prior write events that need to be tracked can be reduced for the sake of efficiency, in which case the term representing a memory location takes the form $m(\!|E^r_{hb}, z{:=}v|\!)$ where m is the identifier of the most recent write to z. See equation (4).

3.2.2 Write-after-read races Besides the detailed coverage of RaW and WaW races in Section 3.3, we describe the detection of *write-after-read* races in Section 3.4. When it comes to WaR, the race checker needs to remember information about past reads in addition to past write events. Abstractly, a read event represents the fact that a load-statement has executed. Thus, the set E_{hb} of an entry $(\!|E_{hb}, z{:=}v|\!)$ in memory holds identifiers of both read and write events.

In the strong semantics, a read always observes one definite value which is the result of one particular write event. Therefore, the configuration contains entries of the form $m(\!|E^r_{hb}, z{:=}v|\!)$ where m is the identifier of the "last" write event and E^r_{hb} is a set of identifiers of read events, namely those that accumulated after m. Note that "records" of the form $m(\!|E^r_{hb}, z{:=}v|\!)$ can be seen as $n+1$ recorded events, one write event together with $n \geq 0$ read-events. This definition of records with one write per variable stands in

contrast to a weak semantics, where many different write events may be observable by a given read [10].

3.2.3 Synchronization

Channel communication propagates happens-before information between threads, and thus, affects synchronization. In the operational rules, each channel c is actually realized with *two* channels, which we refer to as *forward*, c_f, and *backward*, c_b—see Figure 4. The forward part serves to communicate a value transmitted from a sender to a receiver; it also stipulates a causal relationship between the communicating partners [12]—see rule (1) of page 5. To capture this relationship in the context of race checking, the sender also communicates its current information about the happens-before relation to the receiver. The communication of happens-before information is accomplished by the transmission of E_{hb} over channels; see rule R-REC in Figure 4.

The memory model also stipulates a happens-before relationship between a receive and a subsequent send on a channel with capacity k—see rule (2) of page 5. While we refer to the *forward channel* as carrying a message from a sender to a receiver, the backward part of the channel is used to model the indirect connection between some prior receiver and a current sender; see R-SEND in Figure 4.

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, i.e., the forward channel.[6] Thus, the number of "messages" in the backward channel capture the notion of fullness: a channel is full if the backward channel is empty. See rule R-SEND in Figure 4 or Figure 18 for the underlying semantics without race checking. When a channel of capacity k is created, the forward queue is empty and the backward queue is initialized so that it contains dummy elements $E_{hb\perp}$ (cf. rule R-MAKE). The dummy elements represent the number of empty or free slots in the channel. Upon creation, the number of dummy elements equals the capacity of the channel.

As discussed in Section 2, there is a distinction between a synchronization operation and its completion. A send/receive pair on a synchronous channel can be seen as a rendezvous operation; captured in our semantics by the R-REND reduction rule of Figure 4. When it comes to asynchronous communication, the distinction between a channel operation and its completion is handled by the fact that send and receive operations update a thread's local state but do not immediately transmit the updated state onto the channel—see rules R-SEND and R-REC in Figure 4.

---

[6] In the case of lossy channels, backward channels are sometimes used for the purpose of error control and regulating message retransmissions, where the receiver of messages informs the sender about the successful or also non-successful reception of a message. Here, channels are assumed non-lossy and there is no need for error control. In that sense, the term "backward" should not be interpreted as communication *back* to the receiver in the form of an acknowledgment.

3.3 Detecting read-after-write (RaW) and write-after-write (WaW) races

To detect "after-write" races, run-time configurations are given following syntax:

$$R ::= p\langle E_{hb}, t\rangle \mid (\!|E_{hb}^z, z{:=}v|\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R \,. \tag{3}$$

Configurations are considered up-to structural congruence, with the empty configuration $\bullet$ as neutral element and $\parallel$ as associative and commutative. The definition is standard and included in Appendix A.1. Likewise relegated to the appendix are *local* reduction rules, i.e., those not referring to shared variables or channels (see Appendix A.2).

In the configurations, a triple $(\!|E_{hb}^z, z{:=}v|\!)$ not only stores the current value of z but also records the unique identifiers m, m', etc of every write *event* to z in E_{hb}^z.[7] A write to memory updates a variable's value and also generates a fresh identifier m. In order to record the write event, the tuple $(m, !z)$ is placed in the happens-before set of the term representing the memory location that has been written to. The initial configuration starts with one write-event per variable and the semantics maintains this uniqueness as an invariant. In effect, the collection of recorded write events behave as a mapping from variable to values.[8]

A thread t is represented as $p\langle E_{hb}, t\rangle$ at run-time, with p serving as identifier. To be able to determine whether a next action should be flagged as race or not, a goroutine keeps track of happens-before information corresponding to past write events. An event mentioned in E_{hb} is an event of the past, as opposed to being an event that simply occurred in a prior step. An event is "concurrent" if it occurred in a prior step but is not in happens-before relation with the current thread state. Concurrent memory events are potentially in conflict with a thread's next step. More precisely, if the memory record $(\!|E_{hb}^z, z{:=}v|\!)$ is part of the configuration, then it is safe for thread $p\langle E_{hb}, t\rangle$ to write to z if $E_{hb}^z \subseteq E_{hb}$. Otherwise, there exist a write to z that is not accounted for by thread p and a WaW conflict is raised. Similar when reading from a variable.

Data-races are marked as a transition to an exception E—see the derivation rules of Figure 3, and, when write-after-read races are considered, Figure 7. The exception takes as argument a set containing the prior memory operations that conflict and are concurrent with the attempted memory access.

Goroutines synchronize via message passing, which means that channel communication must transfer happens-before information between goroutines. Suppose a goroutine p has just updated variable z thus generating the unique label m. The tuple $(m, !z)$

---

[7] We will later use the term "event" also when talking about histories or traces. There, events carry slightly different information. For instance, being interested in the question whether a history contains evidence of a race, it won't be necessary to mention the actual value being written in the write event in the history. Both notions of events, of course, hang closely together. It should be clear from the context whether we are referring to events as part of a linear history or recorded as part of the configuration. When being precise, we refer to a configuration event as *recorded* event. Since recorded events in the semantics are uniquely labeled, we also allow ourselves to use words like "event m" even if m is just the identifier for the recorded event $m(\!|z{:=}v|\!)$.

[8] The fact that memory behaves like a mapping is consistent with the strong memory assumption.

$$\frac{E_{hb}^z \subseteq E_{hb} \qquad fresh(m') \qquad E'_{hb} = \{(m',!z)\} \cup E_{hb} \qquad E'^z_{hb} = \{(m',!z)\} \cup E_{hb}^z}{p\langle E_{hb}, z := v';t\rangle \parallel (\!|E_{hb}^z, z{:=}v|\!) \to p\langle E'_{hb}, t\rangle \parallel (\!|E'^z_{hb}, z{:=}v'|\!)} \text{ R-WRITE}$$

$$\frac{E_{hb}^z \subseteq E_{hb}}{p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!|E_{hb}^z, z{:=}v|\!) \to p\langle E_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel (\!|E_{hb}^z, z{:=}v|\!)} \text{ R-READ}$$

Fig. 2: Operational semantics augmented for RaW and WaW race detection

$$\frac{E_{hb}^z \nsubseteq E_{hb}}{p\langle E_{hb}, z := v';t\rangle \parallel (\!|E_{hb}^z, z{:=}v|\!) \to \mathsf{E}\left(E_{hb}^z - E_{hb}\right)} \text{ R-WRITE-}\mathsf{E}_{WaW}$$

$$\frac{E_{hb}^z \nsubseteq E_{hb}}{p\langle E_{hb}, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel (\!|E_{hb}^z, z{:=}v|\!) \to \mathsf{E}\left(E_{hb}^z - E_{hb}\right)} \text{ R-READ-}\mathsf{E}_{RaW}$$

Fig. 3: Exception conditions for RaW and WaW data-race detection

is placed in the happens-before set of both the thread p and the memory record associated with z. At this point, p is the only goroutine whose happens-before set contains the label m associated with this write-record. No other goroutine can read or write to z without causing a data-race. When p sends a message onto a channel, the information about m is also sent. Suppose now that a thread p' reads from the channel and receives the corresponding message before p makes any further modifications to z. The tuple $(m,!z)$ is added to p''s happens-before set, so both p and p' are aware of z's most recent write to z. The existence of m in both goroutine's happens-before sets implies that either p or p' are allowed to update z's value. The rules for channel communication are given in Figure 4. They will remain unchanged when we extend the treatment to RaW conflicts. The exchange of happens-before information via channel communication is also analogous to the treatment of the weak semantics in [10].

As in Fava et al. [10], "the R-CLOSE rule closes both sync and async channels. Executing a receive on a *closed* channel results in receiving the end-of-transmission marker $\perp$ (cf. rule R-REC$_\perp$) and updating the local state E_{hb} in the same way as when receiving a properly sent value. The "value" $\perp$ is not removed from the queue, so that all clients attempting to receive from the closed channel obtain the communicated happens-before synchronization information."

Finally, goroutine creation is a synchronizing operation where the child, who is given a unique identifier p', inherits the happens-before set from the parent—see the R-GO rule in Figure 5.

$$\frac{q = [E_{hb\perp}, \ldots, E_{hb\perp}] \qquad |q| = v \qquad fresh(c)}{p\langle E_{hb}, \texttt{let } r = \texttt{make (chan } T, v) \texttt{ in } t\rangle \;\;\rightarrow\;\; \nu c \;(p\langle E_{hb}, \texttt{let } r = c \texttt{ in } t\rangle \parallel c_f[] \parallel c_b[q])} \text{ R-MAKE}$$

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

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

$$\frac{E'_{hb} = E_{hb} + E''_{hb}}{p\langle E_{hb}, \texttt{let } r = \leftarrow c \texttt{ in } t\rangle \parallel c_f[(\perp, E''_{hb})] \;\;\rightarrow\;\; p\langle E'_{hb}, \texttt{let } r = \perp \texttt{ in } t\rangle \parallel c_f[(\perp, E''_{hb})]} \text{ R-REC}_{\perp}$$

$$\frac{E'_{hb} = E_{hb1} + E_{hb2}}{\begin{array}{l} c_b[] \parallel p_1\langle E_{hb1}, c \leftarrow v; t\rangle \parallel p_2\langle E_{hb2}, \texttt{let } r = \leftarrow c \texttt{ in } t_2\rangle \parallel c_f[] \;\;\rightarrow \\ c_b[] \parallel \;\;\; p_1\langle E'_{hb}, t\rangle \;\;\;\; \parallel p_2\langle E'_{hb}, \texttt{let } r = v \texttt{ in } t_2\rangle \;\;\;\; \parallel c_f[] \end{array}} \text{ R-REND}$$

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

Fig. 4: Operational semantics augmented for race detection: channel communication

$$\frac{fresh(p')}{p\langle E_{hb}, \texttt{go } t'; t\rangle \;\;\rightarrow\;\; \nu p' \;(p'\langle E_{hb}, t'\rangle) \parallel p\langle E_{hb}, t\rangle} \text{ R-GO}$$

Fig. 5: Operational semantics augmented for race detection: thread creation

3.4 Detecting write-after-read (WaR) races

In the previous section, the detection of read-after-write and write-after-write races required happens-before sets to contain write labels only. The detection of write-after-read races requires recording read labels, as well. A successful read of variable z causes a fresh read label, say m', to be generated. The pair $(m', ?z)$ is added to the reader's happens-before set as well as to the record associated with z in memory—see the rule R-READ of Figure 6.

In order for a write to memory to be successful, the writing thread must not only be aware of previous write events to a given shared variable, but must also account for all accumulated reads to the variable. A write-after-read data-race is raised when a write is attempted by a thread and the thread is unaware of some previous reads to z. In other words, there exist some read-label in the happens-before set associated with the variable's record, say $r \in E^z_{hb} \downarrow_?$, that is not in the thread's happen-before set, $r \notin E_{hb}$.

$$\frac{E^z_{hb} \subseteq E_{hb} \quad fresh(m') \quad E'_{hb} = \{(m',!z)\} \cup E_{hb} \quad E'^z_{hb} = \{(m',!z)\} \cup E^z_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to p\langle E'_{hb}, t\rangle \parallel (\!| E'^z_{hb}, z{:=}v' |\!)} \text{ R-WRITE}$$

$$\frac{E^z_{hb} \downarrow_! \subseteq E_{hb} \quad fresh(m') \quad E'_{hb} = \{(m',?z)\} \cup E_{hb} \quad E'^r_{hb} = \{(m',?z)\} \cup E^r_{hb}}{p\langle E_{hb}, \texttt{let } r = \texttt{load } z \texttt{ in } t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to p\langle E'_{hb}, \texttt{let } r = v \texttt{ in } t\rangle \parallel (\!| E'^z_{hb}, z{:=}v |\!)} \text{ R-READ}$$

Fig. 6: Operational semantics augmented for data-race detection

$$\frac{E^z_{hb} \not\subseteq E_{hb} \quad E^z_{hb} \downarrow_? \subseteq E_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to \mathsf{E}\left(E^z_{hb} - E_{hb}\right)} \text{ R-WRITE-E}_{WaW}$$

$$\frac{E^z_{hb} \downarrow_? \not\subseteq E_{hb}}{p\langle E_{hb}, z := v'; t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to \mathsf{E}\left(E^r_{hb} - E_{hb}\right)} \text{ R-WRITE-E}_{WaR}$$

$$\frac{E^z_{hb} \downarrow_! \not\subseteq E_{hb}}{p\langle E_{hb}, \texttt{let } r = \texttt{load } z \texttt{ in } t\rangle \parallel (\!| E^z_{hb}, z{:=}v |\!) \to \mathsf{E}\left(E^z_{hb} - E_{hb}\right)} \text{ R-READ-E}_{RaW}$$

Fig. 7: Exception conditions for WaR data-race detection

The projection $\downarrow_?$ essentially filters out write events from the happens-before set. Under these circumstances, the precondition $E^z_{hb} \downarrow_? \not\subseteq E_{hb}$ of the R-WRITE-E$_{WaR}$ rule is met and a race is reported.

Compared to the detector of Section 3.3, rule R-WRITE-E$_{WaW}$ is augmented with the precondition $E^z_{hb} \downarrow_? \subseteq E_{hb}$. Without this precondition, there would be non-determinism when reporting WaW and WaR races.[9] Note, however, that when both WaW and WaR apply, the read in the WaR race happens-after the write involved in the WaW race. We favor to resolve this non-determinism and to report the most recent conflict.

The detector presented here can flag all races: read-after-write, write-after-write, and write-after-read. In Section 4 we also make the detector efficient by "garbage collecting" stale information. But before then, let us look at a couple of examples that illustrate the detector's operation.

---

[9] Consider the scenario in which p writes to and then reads from the shared variable z. Say the write to z generates a label w and the read generates r. If a thread p' attempts to write to z without first communicating with p, p' will not be aware of the prior read and write events. In other words, the happens-before set of p' will contain neither $(w,!z)$ nor $(r,?z)$. Both rules R-WRITE-E$_{WaW}$ and R-WRITE-E$_{WaR}$ are enabled in this case. However, the read happens-after the write that generated $(w,!z)$.

3.5 Examples

We will look at two examples of properly synchronized programs. The first is a typical usage of channel communication; one in which an action is placed in the past of another. The second example relies on mutual exclusion instead. In this case, we know that actions are not concurrent, but we cannot infer an order between them. By contrasting the two examples in Section 3.5.3, we derive observations related to determinism and constructivism.

3.5.1 Message passing Message passing, depicted in Figure 8, involves a producer writing to a shared variable and notifying another thread by sending a message onto a channel. A consumer receives from the channel and reads from the shared variable.

$$p_1 \langle E_{hb1}, z := 42; \; c \leftarrow 0 \rangle$$
$$p_2 \langle E_{hb2}, \leftarrow c; \; \texttt{load } z \rangle$$

Fig. 8: Message passing example.

The access to the shared variable is properly synchronized. Given the operational semantics presented in this chapter, we can arrive at this conclusion as follows. A fresh label, say m, is generated when p_1 writes to z. The memory record involving z is updated with this fresh label, and the pair $(m, !z)$ is placed into p_1's happens-before set, thus yielding $E_{hb}'_1$. A send onto c sends not only the message value, 0 in this case, but also the happens-before set of the sender, $E_{hb}'_1$, see rule R-SEND. The act of receiving from c blocks until a message is available. When a message becomes available, the receiving thread receives not only a value but also the happens-before set of the sender at the time that the send took place, see rule R-REC. Thus, upon receiving from c, p_2's happens-before set is updated to contain $(m, !z)$. Receiving from the channel places the writing to z by p_1 into p_2's definite past. The race-checker makes sure of this fact by inspecting p_2's happens-before set when p_2 attempts to load from z. In other words, the race-checker checks that the current labels associated with z in the configuration are also present in the happens-before set of the thread performing the load.

The message passing example illustrates synchronization as imposing of an order between events belonging to different threads. The message places the producer's write in the past of the consumer's read. Next, we will look into an example in which synchronization is achieve via mutual exclusion. Two threads, p_1 and p_2, are competing to write to the same variable. We will not be able to determine which write happens-before the other. Even though we cannot infer the order, we can determine that a happens-before order exists and, therefore, that the program is properly synchronized.

3.5.2 Mutual exclusion Figure 9 shows a typical mutual exclusion scenario. It involves two threads writing to a shared variable z. Before writing, a thread sends a message onto a channel c which capacity $|c| = 1$. After writing, it receives from c.[10]

$$p_1 \langle c \leftarrow 0;\ z := 17;\ \leftarrow c \rangle$$
$$p_2 \langle c \leftarrow 0;\ z := 42;\ \leftarrow c \rangle$$

Fig. 9: Mutual exclusion example.

A send and its corresponding receive do not directly contribute to synchronization in this example. The send is matched by a receive from the same thread; nothing new is learned from this exchange. To illustrate this point, which may come as a surprise, let us look at an execution. Say p_1 is the first to send 0 onto c. Then p_1's happens-before set E_{hb1} is placed onto the channel along with the value of 0. The thread then proceeds to write to z, which generates a fresh label, say m'; the pair $(m', !z)$ is placed on p_1's happens-before set. When receiving from c, p_1 does not learn anything new! It receives the message 0 and a "stale" happens-before set E_{hb1}. The receiver's happens-before set, E_{hb1}', is updated to incorporate the stale happens-before set, but this "update" causes no effective change:

$$E_{hb1}' \cup E_{hb1} = (E_{hb1} \cup \{(m', !z)\}) \cup E_{hb1}$$
$$= E_{hb1} \cup \{(m', !z)\}$$
$$= E_{hb1}'$$

The explanation for why the program is synchronized, in this case, is more subtle. It involves reasoning about the channel's capacity. Recall that, according to rule (2) on page 5, the ith receive from a channel with capacity k happens before the $(i+k)^{th}$ send onto the channel completes. Since channel capacity is 1 in our example, rule (2) implies that the first receive from the channel happens-before the second send completes. If p_1 is the first to write to z, then p_1 is also the first to receive from c. Receiving from c places p_1's happens-before set onto the backward channel (see rule R-REC). This happens-before set contains the entry $(m', !z)$ registering p_1's write to z. Upon *sending* onto c, p_2 receives from the backward channel and learns of p_1's previous write. Thus, by the time p_2 writes to z, the write by p_1 has been-placed onto p_2's definite past. Since no concurrent accesses exist, the race checker does not flag this execution as racy.

---

[10] Note that the channel is being used as a semaphore [8]. Sending on the channel is analogous to a semaphore `wait` or P operation. Receive is analogous to `signal` or V. The `wait` decrements the value of the semaphore and, if the new value is negative, the process executing the `wait` is blocked. A `signal` increments the value of the semaphore variable, thus allowing another process (potentially coming from the pool of previously blocked processes) to resume. Similarly, a send operation decrements the number of available slots in the channel's queue, while a receive increments it. Sending on a channel with capacity 1 can only take place if the channel is empty; meaning, all previous sends are matched with a corresponding receive.

Similarly, p_2 could first send onto c and write to z. The argument for the proper synchronization of this alternate run would proceed in the same way. Therefore, even though it is not possible to infer who, among p_1 and p_2, writes to z first, we know that one of the writes is in a happens-before relation with the other. This knowledge is enough for us to conclude that the program is properly synchronized.

This example shows that channels are excessively powerful when it comes to implementing mutual exclusion, as evidenced by the fact that the forward queue associated with the channel is not utilized. When it comes to mutual exclusion, a more parsimonious synchronization mechanism suffices. Indeed, the *acquire* and *release* semantics associated with locks is a perfect fit. When acquiring a lock, a thread *learns* about the memory operations that precede the lock's release. In other words, memory operations preceding a lock's release are put in happens-before with respect to a thread that acquires the lock. Assuming a lock l starts with empty happens-before information, say $l[\emptyset]$, the rules ACQUIRE and RELEASE capture a lock's behavior.

$$\frac{E'_{hb} = E_{hb} \cup E''_{hb}}{p\langle E_{hb}, \mathtt{acq}\,(l);t\rangle \parallel l[E''_{hb}] \;\;\rightarrow\;\; p\langle E'_{hb}, t\rangle \parallel l[]} \;\; \text{ACQUIRE}$$

$$\frac{E'_{hb} = E_{hb} \cup E''_{hb}}{p\langle E_{hb}, \mathtt{rel}\,(l);t\rangle \parallel l[] \;\;\rightarrow\;\; p\langle E_{hb}, t\rangle \parallel l[E'_{hb}]} \;\; \text{RELEASE}$$

Note that an acquired lock, represented by $l[]$, cannot be re-acquired without a prior release, and that a released lock, meaning $l[E_{hb}]$, cannot re-released without a prior acquire.[11] While a thread's happens-before is updated on both sends and receives, with locks, only the acquisition updates a thread's happens-before information. Surrounding code with a call to acquire at the beginning and release at the end is sufficient for ensuring mutual exclusion. The full generality of channels is not required.

3.5.3 Determinism, confluence, and synchronization In the message passing example of Section 3.5.1, we are able to give a constructive proof-sketch of the synchronization between p_1 and p_2; the "proof" puts an event from p_1 in the past of p_2. In the mutual exclusion example of Section 3.5.2, no such guarantee is possible. Instead, we give a non-constructive "proof" that p_1 and p_2 are synchronized by arguing that either p_1's actions are in the past of p_2's or vice versa. The *law of excluded middle* is used in this non-constructive argument.

The absence of constructivism is tied to the absence of determinism. While in the message passing example the program is deterministic, in the mutual exclusion example it is not. There is no *data* race in the mutual exclusion example, but there is still a "race" insofar as the two threads compete for access to a shared resource. The resource, in this case, is the channel, which is being used as a lock. The two threads race towards

---

[11] When releases are matched by an prior acquire from the same thread, then happens-before information accumulates monotonically, meaning, a thread learns about all previous releases, not just the most recently occurring one.

acquiring the lock (i.e. sending onto the channel) first. The initial configuration has two transitions, one in which p_1 acquires the lock first and one in which p_2 does. These transitions are non-confluent.

When it comes to reasoning about programs that model hardware, the lack of constructivism and the non-confluence in the use of channels as locks is a hindrance. Deterministic languages and constructive logics are needed in order to rule out scenarios in which two logic gates attempt to drive the same *via* with different logic values (i.e. a short circuit) [2]. In the case of channel communication and in the absence of shared memory, determinism can be achieved by enforcing ownership on channels; for example, by making sure a single thread can read and a single thread can write on a given channel at any given point in the execution [37]. It is possible for the ownership on channels to be passed around the threads in a way that preserves determinism [38].

The examples show that the absence of absence of data races is not enough to ensure determinism. In general, however, determinism is not a requirement. Many applications require "only" data-race freedom.

4   Efficient data-race detection

We have been gradually introducing a data-race checker. In Section 3.3, we presented a simple checker that flags after-write races (WaW and RaW) but is not equipped for write-after-read (WaR) detection. In Section 3.4, we augmented the detector to handle WaR. Here, we discuss how these detectors can be implemented efficiently; where efficiency is gained by employing "garbage collection" to reduce the detector's memory footprint. Note that keeping one record per variable is already a form of efficiency gain. In a relaxed memory model, since there may be more than one value associated with a variable at any point in the execution, one might keep one record per memory event [10]. The first step towards a smaller footprint is to realize that, if the underlying memory model supports the DRF-SC guarantee, a data-race detector can be built assuming sequential consistency. The reason being that, when a data race is flagged, execution stops at the point in which the weak and strong memory models' executions would diverge.

Knowing that memory events can overtake each other, in this section we discuss how stale or redundant information can be garbage collected. More precisely, we show how to garbage collect the data structures that hold happens-before information, that is, the thread-local happens-before set and the per-memory-location one.

4.1   Most recent write

Terms representing a memory location have taken different shapes when compared to the undecorated semantics. In the undecorated semantics, the content v of a variable z is written as a pair $(\!|z{:=}v|\!)$. For after-write race detection, an entry in memory took the form of $(\!|E_{hb}, z{:=}v|\!)$ with E_{hb} holding information about prior write events. Our first optimization comes from realizing that we do not need to keep a set of prior write events. We can record only the most recent write and still be able to flag all after-write racy *executions*. With this optimization, we may fail to report all *accesses* involved in

the race, but we will still be able to report the execution as racy and to flag the most recent conflicting write event. This optimization is significant; it reduces the arbitrarily large set of prior write events to a single point.

An intuitive argument for the correctness of the optimization comes from noticing that a successful write to a variable can be interpreted as the writing thread taking *ownership* of the variable. Suppose a goroutine p has just updated variable z. At this point, p is only goroutine whose happens-before set contains the label, say m, associated with this write-record. The placement of the new label into p's happens-before set can be seen as recording p's *ownership* of the variable: a data-race is flagged if any other thread attempts to read or write to z without first synchronizing with p—see the check $(m, !z) \in E_{hb}$ in the premise of the R-WRITE and R-READ rules of Figure 10.

When p sends a message onto a channel, the information about m is also sent. Suppose now that a thread p' reads from the channel and receives the corresponding message before p makes any further modifications to z. The tuple $(m, !z)$ containing the write-record's label is added to p''s happens-before set. Now both p and p' are aware of z's most recent write to z. The existence of m in both goroutine's happens-before sets imply that either p or p' are allowed to update z's value. We can think of the two goroutines as *sharing z*. Among p and p', whoever updates z first (re)gains the *exclusive* rights to z.

It may be worth making a parallel with hardware and cache coherence protocols. Given the derivation rules, we can write a race detector as a state machine. Compared to the Modified-Exclusive-Shared-Invalid protocol (MESI), our semantics does not have the *modified* state: all changes to a variable are immediately reflected in the configuration, there is no memory hierarchy in the memory model. As hinted above, the other states can be interpreted as follows: If the label of the most recent write to a variable is only recorded in one goroutine's happens-before set, then we can think of the goroutine as having *exclusive* rights to the variable. When a number of goroutines contain the pair $(m, !z)$ in their happen-before set with m being the label of the most recent write, then these goroutines can be thought to be *sharing* the variable. Other goroutines that are unaware of the most recent write can be said to hold *invalid* data.

4.2   Runtime configuration and memory related reduction rules

Given the "most recent write" optimization above, and, if we were satisfied with after-write conflicts, an entry in memory would take the form of $m(\!|z{:=}v|\!)$, with the label m uniquely identifying the event associated with v having been stored into z. Being able to flag after-write but not write-after-read races may be an adequate trade-off between completeness and efficiency. By not having to record read events, a simplified detector tailored for after-write race detection has a much smaller footprint than when read-after-write conflicts are also taken into account. Besides, a write-after-read race that is not flagged in an execution may realize itself as a read-after-write race in another run, and then be flagged by the simplified detector.[12]

---

[12] Intuitively, say $S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} S_n$ is a run starting from an initial configuration S_0. Let $\bowtie$ be an independence relation on events, meaning, given $S_i \xrightarrow{e_i} S_{i+1} \xrightarrow{e_{i+1}} S_{i+2}$, we say that

In contrast, the detection of write-after-read races requires more book-keeping: we need read- in addition to write-labels. This addition is required because a WaR conflict can ensue between an attempted write and *any* previous unsynchronized read to the same variable. Therefore, the race-checker is made to remember all such potentially troublesome reads.[13] The runtime configuration is thus modified, this time as to contain entries of the form $m(\!|E_{hb}^r, z{:=}v|\!)$. The label m identifies of the most recent write event to z and the set E_{hb}^r holds-read event identifiers, namely, the identifiers of reads that accumulated after m.

$$R ::= p\langle E_{hb}, t\rangle \mid m(\!|E_{hb}^r, z{:=}v|\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R\,. \tag{4}$$

Note that *records* of the form $m(\!|E_{hb}^r, z{:=}v|\!)$ can be seen as $n+1$ recorded events: one write together with $n \geq 0$ read events.

The formal semantics maintains the following invariants. First, the happens-before information E_{hb}^r in $m(\!|E_{hb}^r, z{:=}v|\!)$ contains information of the form $(m', ?z)$ only, i.e., there are no write events and all read-events concern variable z. Also, the event labels are unique for both reads and writes. In an abuse of notation, we may refer to m being in E_{hb}^r and write $m \in E_{hb}^r$ meaning, more precisely, $(m, ?z) \in E_{hb}^r$.

$$\frac{(m, !z) \in E_{hb} \qquad E_{hb}^r \subseteq E_{hb} \qquad fresh(m') \qquad E_{hb}' = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow z)}{p\langle E_{hb}, z := v'; t\rangle \parallel m(\!|E_{hb}^r, z{:=}v|\!) \rightarrow p\langle E_{hb}', t\rangle \parallel m'(\!|\emptyset, z{:=}v'|\!)} \text{ R-Write}$$

$$\frac{E_{hb}''^r = \{(m', ?z)\} \cup \left(E_{hb}^r - E_{hb} \downarrow z\right)}{(m, !z) \in E_{hb} \quad fresh(m') \quad E_{hb}' = \{(m', ?z)\} \cup (E_{hb} - E_{hb} \downarrow z) \cup \{(m, !z)\}}{p\langle E_{hb}, \mathtt{let}\, r = \mathtt{load}\, z\, \mathtt{in}\, t\rangle \parallel m(\!|E_{hb}^r, z{:=}v|\!) \rightarrow p\langle E_{hb}', \mathtt{let}\, r = v\, \mathtt{in}\, t\rangle \parallel m(\!|E_{hb}''^r, z{:=}v|\!)} \text{ R-Read}$$

Fig. 10: Operational semantics augmented for efficient data-race detection

4.3 Garbage collection of happens-before sets

Knowledge of past events contained in a happens-before set E_{hb} is naturally monotonically increasing. For example, each time a goroutine learns about happens-before

---

$e_i \bowtie e_{i+1}$ if there exist S' such that $S_i \xrightarrow{e_{i+1}} S' \xrightarrow{e_i} S_{i+2}$. The independence relation induces an equivalence relation on traces, namely, traces are equivalent if they can be derived from one another via the permutation of independent events. It can be shown that if $S_0 \xrightarrow{h} S_n$ is a run containing a write-after-read race, the exist an equivalent run in which the race materializes as a read-after-write race.

[13] Since depending on scheduling, a WaR data-race can manifest itself as RaW race, one option would be not add instrumentation for WaR race detection and, instead, hope to flag the RaW manifestation instead. Such practical consideration illustrates the trade-off between completeness versus run-time overhead.

information, it adds to its pool of knowledge. In particular, events that are known to have "happened-before" cannot, by learning new information, become "concurrent." An efficient semantics, however, does not accumulate happens-before information indiscriminately; instead, it purges redundant information. We say "redundant" for the purpose of flagging racy executions, but leaving out conflicting accesses that have been overtaken by more recent memory events.

4.3.1   Garbage collection on *writes* For a thread t to successfully write to z, all previously occurring accesses to z must be in happens-before with the thread's current state. One optimization comes from realizing that we can purge all information about prior accesses the variable z from the happens-before set of the writing thread t. We call these prior accesses *redundant* from the point of view of flagging racy executions. The reason for the correctness of this optimization is as follows: All future access of t to z are synchronized with the redundant accesses, after all, the accesses are recorded in t's happens-before set. Therefore, from the perspective of t, these accesses do not affect data-race detection. For the same reason, if a thread t' synchronizes with t, there is no race to report if and when t' accesses memory—the absence of these redundant accesses from t''s happens-before is, therefore, inconsequential. Finally, if t' does not synchronize with t, then an access to z is racy because it is unsynchronized with t's most recent write, regardless of the redundant prior accesses. Note that this optimization allows us to flag all racy *executions* even if we fail to report some of the accesses involved in the race.

Rule R-WRITE of Figure 10 embodies this discussion. Before writing, the rule checks that the attempted write happens-after all previously occurring accesses to z. This check is done by two premises: premise $(m, !z) \in E_{hb}$ makes sure that the most recent write to z, namely, the one that produced event $(m, !z)$, is in happens-before with the current thread state E_{hb}. As per discussion in Section 4.1, being synchronized with the most recent write means the thread is synchronized with all writes up to that point in the execution. The other premise, $E_{hb}^r \subseteq E_{hb}$, makes sure that the attempted write is in happens-after read accesses to z. If these two premises are satisfied, the write can proceed and prior accesses to z are garbage collected from the point of view of t. The filtering of redundant accesses is done by subtracting $E_{hb} \downarrow_z$ in

$$E_{hb}' = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow_z)$$

where $\downarrow_z$ projects the happens-before set down to operations on variable z. Finally, the write rule also garbage collects the in-memory record E_{hb}^r by setting it to $\emptyset$,[14] meaning that no read event have accumulated after the write yet.

4.3.2   Garbage collection on *reads* We also garbage collect on load operations. Say t reads from z, thus generating event $(m', ?z)$. Let us call *redundant* the memory accesses

---

[14] As per discussion in Section 4.1, a term representing a memory location $m(\!|E_{hb}^r, z\!:=\!v|\!)$ records in E_{hb}^r all the reads to z that have accumulated after the write that generated the write label m. When a new write m' of value $z := v'$ ensues, we update the memory term to record this new write and we reset its corresponding E_{hb}^r to $\emptyset$.

to z in t's happens-before set at the time event $(m', ?z)$ takes place, with the exception of $(m, !z)$. A read operation can only conflict with a future write; there are not read-read conflicts. For a future write to take place, the writing thread will need to synchronize with a thread that "knows" about the read m'.[15] Any thread that knows of m' would also know about the redundant access to z and know of $(m, !z)$. In other words, m' and m subsume all happened-before accesses of z from the perspective of t. Therefore, we can garbage collect all such accesses by filtering them out of the thread's happen-before set, as in

$$E'_{hb} = \{(m', ?z)\} \cup (E_{hb} - E_{hb} \downarrow_z) \cup \{(m, !z)\}.$$

These redundant accesses are also filtered out of the in-memory happens-before set:

$$E''^r_{hb} = \{(m', ?z)\} \cup (E^r_{hb} - E_{hb} \downarrow_z).$$

4.3.3   Off-line garbage collection and channel communication   The garbage collector rules of Figure 11 can be run non-deterministically during the execution of a program. Rule R-GC eliminates stale entries from the happens-before set of a thread. It can be sensible to perform garbage collection also *after* a thread interacts with a channel, as happens-before information communicated via channels are likely to become stale. For example, suppose a thread, whose happens-before set does not contain stale entries, sends onto a channel and continues executing. By the time a receive takes place, the happens-before set transmitted via the channel may have become stale. Similarly for happens-before transmitted between receives and prior sends via the backward channel. Alternatively, we may choose an implementation in which the happens-before of in-flight messages are also garbage collected, in which case we would process the happens-before sets in a channel's forward and backward queues.

$$\frac{\begin{array}{c} E'_{hb} = E_{hb} - \{(\hat{m}, !z) \mid (\hat{m}, !z) \in E_{hb} \ \wedge \ \hat{m} \neq m\} \\ - \{(\hat{m}, ?z) \mid (\hat{m}, ?z) \in E_{hb} \ \wedge \ (\hat{m}, ?z) \notin E^r_{hb}\} \end{array}}{p\langle E_{hb}, t\rangle \parallel m(\!|E^r_{hb}, z := v|\!) \rightarrow p\langle E'_{hb}, t\rangle \parallel m(\!|E^r_{hb}, z := v|\!)} \text{ R-GC}$$

Fig. 11: Off-line garbage collection

---

[15] "Knowing about the read m'" is a necessary condition for a thread to successfully write to z, but it is not a sufficient one. There may exist other reads, say m'', m''', etc that are concurrent with m'. A thread needs to synchronize with all such concurrent reads before it can successfully write to z.

5 Comparison with vector-clock based race detection

Vector clocks (VCs) are a mechanism for capturing the happen-before relation over events emanating from a program's execution [22]. A *vector clock* $\mathbb{V}$ is a function `Tid→Nat` which records a clock, represented by a natural number, for each thread in the system. "VCs are partially-ordered ($\sqsubseteq$) in a pointwise manner, with an associated join operation ($\sqcup$) and minimal element ($\perp_V$). In addition, the helper function inc_t increments the t-component of a VC" [11].

$$\mathbb{V}_1 \sqsubseteq \mathbb{V}_2 \;\; \text{iff} \;\; \forall t.\, \mathbb{V}_1(t) \leq \mathbb{V}_2(t)$$
$$\mathbb{V}_1 \sqcup \mathbb{V}_2 \;\; = \;\; \lambda t.\, max(\mathbb{V}_1(t), \mathbb{V}_2(t))$$
$$\perp_V \;\; = \;\; \lambda t.\, 0$$
$$inc_t(\mathbb{V}) \;\; = \;\; \lambda u.\, \text{if } u = t \text{ then } \mathbb{V}(u) + 1 \text{ else } \mathbb{V}(u)$$

Using vector clocks, Pozniansky and Schuster [29] proposed a data-race detection algorithm referred to as DJIT.[+] Their algorithm works as follows. Each thread t is associated with a vector clock $\mathbb{C}_t$. Entry $\mathbb{C}_t(t)$ stores the current time at t, while $\mathbb{C}_t(u)$ for $u \neq t$ keeps track of the time of the last operation "known" to t as having been performed by thread u.

The algorithm also keeps track of memory operations. Each memory location x has two vector clocks, one associated with reads, $\mathbb{R}_x$, and another with writes, $\mathbb{W}_x$. The clock of he last read from variable x by thread t is recorded in $\mathbb{R}_x(t)$; similar for $\mathbb{W}_x(t)$ and writes to x by t. When it comes to reading from memory, a race is flagged when a thread t attempts to read from x while being "unaware" of some write to x. Precisely, a race is flagged when t attempts to read from x and there exists a write to x by thread u, $\mathbb{W}_x(u)$, that is not accounted for by t, meaning $\mathbb{W}_x(u) \geq \mathbb{C}_t(u)$, or, equivalently, $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$. If t succeeds in reading from x, then $\mathbb{R}_x(t)$ is updated to the value of $\mathbb{C}_t(t)$. When it comes to writing to memory, a race is flagged when t attempts to write to x while being unaware of some read or write to x, meaning either $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$ or $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$. If t succeeds in writing to x, then $\mathbb{W}_x(t)$ is updated to $\mathbb{C}_t(t)$.

A thread's clock is advanced when the thread executes synchronization operations, which have bearing on the happens-before relation. The algorithm was proposed in the setting of locks; each lock m is associated with a vector clock $\mathbb{L}_m$. When a thread t acquires m, then $\mathbb{C}_t$ is updated to $\mathbb{C}_t \sqcup \mathbb{L}_m$. Acquiring a lock is analogous to receiving from a channel with buffer size one: the receiving thread updates its vector clock by incorporating the VC previously "stored" in the lock. When a thread t releases a lock m, the vector clock $\mathbb{L}_m$ is updated to $\mathbb{C}_t$ and thread's clock is advanced, meaning $\mathbb{C}_t := inc_t(\mathbb{C}_t)$. We can think of lock release as placing a message, namely the vector clock associated with the releasing thread, into a buffer of size one. Thus, in comparison with the approach presented in our paper, lock operations are a special case of buffered channel communication. Our paper deals with channels of arbitrary size and their capacity limitations.

A significant difference between our approach and DJIT[+] is that we dispense with the notion of vector clocks. Vector clocks are a conceptual vehicle to capturing partial order of events. Instead of relying on VCs, our formalization is tied directly to the concept of happens-before. Vector clocks are expensive. VCs require $O(\tau)$ storage space

and common operations on VCs consume $O(\tau)$ time where τ is the number of entries in the vector. In the case of race detection, τ is the number of threads spawn during the execution of a program. It turns out that not all uses of VCs in DJIT$^+$ are strictly necessary. In fact, Flanagan and Freund [11] introduce the concept of *epoch*, which consists of a pair $c@t$ where c is a clock and t a thread identifier. They then replace $\mathbb{W}_x$, the vector clock tracking writes to x, with a single epoch. This epoch captures the clock and thread identity associated with the most recent write to x. Similarly, in our approach, a memory location is associated with the identifier of only the most-recent write to that location. Any thread who is "aware" of this identifier is allowed to read from the corresponding variable.

FASTTRACK also reduces the dependency on vector clocks by replacing $\mathbb{R}_x$ with the epoch of the most recent read to x. However, since reads are not totally ordered, FASTTRACK dynamically switches back to a vector clock representation when needed. Similar to FASTTRACK, we record the most recent (unordered) reads which, in the best case, involves an $O(1)$-memory footprint and $O(\tau)$ at the worst.

When it comes to per-thread memory consumption, however, our approaches look very different. While DJIT$^+$'s and FASTTRACK's worst-case memory consumption per thread is $O(\tau)$, our is $O(v\tau)$ where v is the number of shared variables in a program.[16] Vector clocks' memory efficiency, when compared to happens-before sets, come from VC's ability to succinctly capture the per-thread accesses that take place in between advances of a clock. A thread's clock is advanced when the thread releases a lock.[17] All accesses made by a thread t in a given clock c are captured by the clock: if another thread u "knows" the value c of t's clock, then u is in happens-after with *all* accesses made by t—that is, all accesses up to when t's clock was advanced to $c+1$. In contrast, the happens-before set representation is much more coarse. We keep track of individual accesses, as opposed to lumping them together into a clock number. This coarseness explains the extra factor of v in the worst-case analysis of the happens-before set solution. Although being a disadvantage in the worst case scenario, it does provide benefits, as we discuss next.

Note that the vector-clocks associated with threads and locks grow monotonically. By *growing monotonically* we do not mean that time marches forward to increasing clock values. Instead, we mean that the number of clocks in a vector grows without provisions for the removal of entries. This growth can lead to the accumulation of "stale" information, where by stale we mean information that is not useful from the point of view of race detection. This growth stands in contrast to our approach to garbage collection. Stale information is purged from happens-before sets, which means they can shrink back to size zero after having grown in size.

Let us look at an example that illustrates this difference in treatment of stale information. Consider the producer/consumer paradigm, where a thread produces information to be consumed by other threads. Say p_0 produces information by writing to the

---

[16] We believe the worst case is a degenerate case unlikely to happen: it involves every thread reading from every shared variable and then exchanging messages as to inform everyone else about their read events.

[17] If channels were used instead of locks, the advance would take place when a thread sends onto or receives from a channel.

shared variable z. The thread then notifies consumers, p_1 and p_2, by sending a message on channel c. The consumers read from z and signal the fact that they are done consuming by sending onto channel d. The producer p_0 writes to z again once it has received the consumers' messages.

Producer	Consumers	
p_0	p_1	p_2
$z := 42;$	$\leftarrow c;$	$\leftarrow c;$
$c \leftarrow 0;\ c \leftarrow 0;$	$\texttt{load } z;$	$\texttt{load } z;$
$\leftarrow d;\ \leftarrow d;$	$d \leftarrow 0$	$d \leftarrow 0$
$z := 43$		

Let us run this example against a prototype implementation [9] of our proposed race detector, called GRACE, and against FASTTRACK. Consider the point in the execution after p_0 has written to z, the consumers have read from z and notified p_0, and p_0 is about to write to z again. Below is the state of the detectors at this point. The information contained in the happens-before sets and the vector-clocks is very similar. There are three entries for p_0, and two entries for p_1 and p_2 each.

Happens-before sets	Vector-clocks
$E_{hb}^{p_0} = \{(m_0, !z), (m_1, ?z), (m_2, ?z)\}$	$\mathbb{C}_{p_0} = \perp[p_0 \mapsto 6, p_1 \mapsto 1, p_2 \mapsto 1]$
$E_{hb}^{p_1} = \{(m_0, !z), (m_1, ?z)\}$	$\mathbb{C}_{p_1} = \perp[p_0 \mapsto 2, p_1 \mapsto 2]$
$E_{hb}^{p_2} = \{(m_0, !z), (m_2, ?z)\}$	$\mathbb{C}_{p_2} = \perp[p_0 \mapsto 3, p_2 \mapsto 2]$

The happens-before set $E_{hb}^{p_0}$ show the reads by p_1 and p_2 as being in happens-before with respect to p_0, along with p_0's own write to z. It also shows p_1 and p_2 as being "aware" of p_0's write to z, as well as being "aware" of their own reads to z. The same information is captured by the vector clocks. Recall that the bottom clock, $\perp$, maps every process-id to the clock value of 0. Thus, the VC associated with p_0 contains p_0's clock (which happens to be 6) as well as the clock associated with the reads to z by p_1 and p_2. In this execution, p_0's clock was 2 when the thread wrote to z. Thus, the entry $p_0 \mapsto 2$ in $\mathbb{C}_{p_1}$ and the entry $p_0 \mapsto 3$ in $\mathbb{C}_{p_2}$ place the write to z by p_0 in p_1's and p_2's past.

The difference between our approach the VC based approach is evidenced in the next step of execution, when p_0 writes to z for the second time. This write subsumes all previous memory interactions on z. In other words, this write is in happens-after with respect to all reads and writes to z up to this point in the execution of the program. Therefore, it is sufficient for a thread to synchronize with p_0 before issuing a new read or write to z; also, it is no longer necessary to remember the original write to z and the reads from z by p_1 and p_2. Here are the happens-before sets and vector-clocks in the next step of execution, meaning, after p_0 writes to z the second time:

Happens-before sets	Vector-clocks
$E_{hb}^{p_0} = \{(m_3, !z)\}$	$\mathbb{C}_{p_0} = \bot[p_0 \mapsto 6, p_1 \mapsto 1, p_2 \mapsto 1]$
$E_{hb}^{p_1} = \{\}$	$\mathbb{C}_{p_1} = \bot[p_0 \mapsto 2, p_1 \mapsto 2]$
$E_{hb}^{p_0} = \{\}$	$\mathbb{C}_{p_2} = \bot[p_0 \mapsto 3, p_2 \mapsto 2]$

The happens-before sets are mostly empty; the only entry corresponds to the most recent write to z, which is known to p_0. Meanwhile, the vector clocks are unchanged. Note, however, that every entry with the exception of $p_0 \mapsto 6$ in $\mathbb{C}_{p_0}$ is stale. In other words, with the exception of $p_0 \mapsto 6$, the presence or absence of all other entries does not alter a thread's behavior. To illustrate this point, take entry $p_0 \mapsto 2$ in $\mathbb{C}_{p_1}$ as an example: if p_1 were to attempt to access z, a data race will ensue regardless of whether or not the entry $p_0 \mapsto 2$ is in p_1's vector clock. Therefore, ideally, we would want these stale entries purged from the vector-clocks of p_0, p_1, and p_2. Concretely, we would want $\mathbb{C}_{p_0} = \bot[p_0 \mapsto 6]$ and $\mathbb{C}_{p_1} = \mathbb{C}_{p_2} = \bot$.

Similar unbounded growth occurs in the VCs associated with locks,[18] thus also leading to the accumulation of stale information. We conjecture that an approach that purges stale information from VCs, similar to our notion of garbage collection, would be highly be beneficial. VC-based implementations are very efficient in managing the memory overhead associated with variables. For example TSan, a popular race-detection library based on vector clocks and that comes with the Go tool chain, stores one write and a small number of reads per memory location (the number of reads stored is 4 in the current implementation) [15]. Capping the number of tracked read events leads to false negatives; the cap a fair compromise between *recall* and memory consumption. In order to further reduce the memory foot-print of modern race detection implementations, we are thus left with devising approaches to managing threads' and locks' memory overhead.

Unfortunately, reducing memory pressure on vector-clocks associated with threads and locks is arguably more difficult than reducing memory pressure on VCs associated with shared variables. In one hand, if a variable does not "remember" a read or write to itself as having happened-before, then the variable becomes more permissive from the point of view of race detection; meaning, more threads would be able to interact with this variable without raising a data-race, even when races should have been reported. On the other hand, if a *thread* "forgets" about some prior read or write access that have taken place on a variable, a spurious data race may be raised. Thus, while dropping clock entries in the VCs associated with variables can introduce false negatives, dropping clock entries from VCs associated with threads and locks introduce false positives. From a practical perspective, false negatives are acceptable and can even be mitigated,[19] however, being warned of non-existing races is overwhelming to the application programmer, which means false positives are generally not tolerated.

---

[18] The *acquire* grows the VC associated with the acquiring thread; the *release* sets the VC of the corresponding lock to the VC of the acquiring thread.

[19] Provided we run a program enough times, we can randomly evict entries from a VC or happens-before set associated with a variable such that we eventually flag all existing races of the program.

6 Connections with trace theory

Our operational semantics mimics the Go memory model in defining synchronization in terms of channel communication. Specifically, we abide by rules (1) and (2), which establish a happens-before relation between a send and the completion of its corresponding receive, and, due to the boundedness of channels, between a receive and the completion of a future send. However, these are not the only imposition by the semantics on the order of events. Channels act as FIFO queues in both Go [4] as well as in our operational semantics. However, neither Go nor our operational semantics establish a happens-before relation between consecutive sends or consecutive receives. For example, the ith send on a channel c does not happens-before the $(i+1)^{th}$ send on c. Therefore, there exist events that are necessarily ordered, but that are not in happens-before relation.

It is tempting to think of happens-before in terms of observations, where a and b are in happens-before if and only if we observe a followed by b, and never the other way around. This intuition is captured by the following tentative definition:

Let $\texttt{idx}\,(a,h)$ *be the index of event a in a run h. Given the set of runs H starting from an initial configuration, we say that event a* happens-before *b if-and-only-if, for all runs $h \in H$ such that $a,b \in h$,* $\texttt{idx}\,(a,h) < \texttt{idx}\,(b,h)$.

When it comes to weak memory systems, there exist events that are ordered according to the above tentative definition but that are not in happens-before relation. Take the improperly synchronized message-passing example of Figure 12 as an example. In this example, a thread p_0 writes to a shared variable z and sets a flag; another thread, p_1, checks the flag reads from z if the flag has been set.

p_0		p_1	
`z    := 42;`	(A)	`r = load done;`	(C)
`done := true;`	(B)	`if r then`	
		`     load z`	(D)

Fig. 12: Message passing example.

If A and B are the first and second instructions in thread p_0, and C and D are the loads of the flag and of the shared variable z in p_1, then program order gives rise to $A \to_{\mathsf{hb}} B$ and $C \to_{\mathsf{hb}} D$. We also have that the load of z in D only occurs if the value of the flag observed by thread p_1 is `true`, which means it was previously set by thread p_0 in B. Therefore, in all runs in which D is observed, B necessarily occurs earlier in the execution. This necessity does not, however, place B and D in happens-before relation. Under many flavors of weak memory, the memory accesses between the two threads are not synchronized. As the example shows, our tentative definition of happens-before as always-occurring-before or necessarily-occurring-before does not work for weak memory systems. How about for sequential consistent ones?

In the program of Figure 13, thread p_0 sends values 0 and 1 into channel c consecutively. Concurrently, thread p_1 writes 42 to a shared variable z and receives from the channel, while thread p_2 first receives from the channel and conditionally reads from z. From this program, we construct an example in which events are necessarily ordered but are not in happens-before—even if we assume sequential consistency. To illustrate

$$p_0\langle c \leftarrow 0; c \leftarrow 1\rangle$$
$$p_1\langle z := 42; \leftarrow c\rangle$$
$$p_2\langle \text{let } r := \leftarrow c \text{ in if } r = 1 \text{ then load } z\rangle$$

Fig. 13: Conditional race example.

this point, let us consider an execution of the program. Let $(o)_p$ be a trace event capturing the execution of operation o by threads p. Let also $z!$ and $z?$ represent a write and read operation on the shared variable z, and $\text{sd } c$ and $\text{rv } c$ represent send and receive operations on channel c. Assuming channel capacity $|c| \geq 2$, the sequence below is a possible trace obtained from the execution of the program. Note that the if-statement's reduction is interpreted as an internal or silent transition:

$$(\text{sd } c)_{p_0} \; (\text{sd } c)_{p_0} \; (z!)_{p_1} \; (\text{rv } c)_{p_1} \; (\text{rv } c)_{p_2} \; (z?)_{p_2} \tag{5}$$

Given that p_1 receives from c before p_2 does, the value received by p_2 must be 1 as opposed to 0. Therefore, p_2 takes the branch and reads from the shared variable z. Figure 14 shows the partial order on events for this execution. Program order is captured
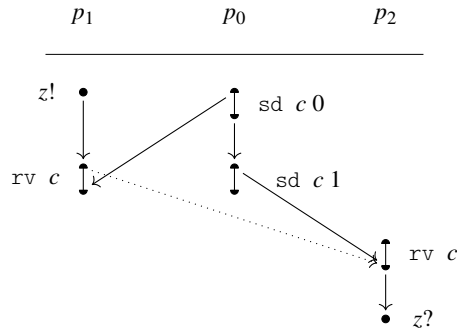

Fig. 14: Partial order on conditional-race example.

by the vertical arrows in the diagram; channel communication is captured by the solid diagonal arrows. As per discussion in Section 3.2.3, we make the distinction between

a channel operation and its completion. A channel operations is depicted as two half-circles; the operation's completion is captured by the bottom half-circle. That way, a send (top of the half-circle) happens-before its corresponding receive completes (bottom half).

Now, given that the send operations are in happens-before, meaning $(\texttt{sd}\ c\ 0)_{p_0} \rightarrow_{\mathsf{hb}}$ $(\texttt{sd}\ c\ 1)_{p_0}$, and that channels are First-In-First-Out (FIFO), the reception of value 0 from c must occur before the reception of 1. This requirement is captured by the dotted arrow in the diagram. However, according to the semantics of channel communication (i.e. rules (1) and (2) of page 5), this order does not impose a happens-before relation between the receiving events. In other words, there exist events that are necessarily ordered, but not in happens-before relation to one another.

The failure of our tentative definition of happens-before as necessarily-occurring-before, given early in this section, has subtle implications as discussed next.

6.1 Happens-before, traces, and commutativity of operations

Traces come from observing the execution of a program and are expressed as strings of events. In a concurrent system, however, events may not be causally related, which means that the order of some events is not pre-imposed. In reality, instead of sequences, events in a concurrent system form a partially ordered set (see Figure 14 for an example). As advocated by Mazurkiewicz [23], it is useful to combine sequential observations with a dependency relation for studying "the nonsequential behaviour of systems via their sequential observations." By defining an *independence relation* on events, it is possible to derive a notion of equivalence on traces: two traces are equivalent if it is possible to transform one into the other "by repeatedly commuting adjacent pairs of independent operations" [17].

One way to define independence is as follows: Given a run $R_i \xrightarrow{a} \cdot \xrightarrow{b} R$, we say that a and b are independent if $R_i \xrightarrow{b} \cdot \xrightarrow{a} R$, meaning,

- b is enabled at R_i,
- a is enabled at $R_i \xrightarrow{b} \cdot$, and
- there exists an R' such that $R_i \xrightarrow{b} R' \xrightarrow{a} R$.

Clearly, if a happens-before b, then a and b cannot be swapped in a trace. So, independence between two events means (at least) the absence of happens-before relation between them. But happens-before is not all that needs to be considered in the definition of independence.

When translating a partial order of events to a trace, not every linearization that respects the happens-before relation is a valid trace. Some linearizations of the partial order may not be "realizable" by the operational semantics. In other words, there can be traces that abide by the happens-before relation but that cannot be generated from the execution of a program. For example, we can obtain the following linearization given the partial order of Figure 14:

$$(\texttt{sd}\ c\ 0)_{p_0}\ (\texttt{sd}\ c\ 1)_{p_0}\ (\texttt{rv}\ c)_{p_2}\ (z!)_{p_2}\ (z?)_{p_1}\ (\texttt{rv}\ c)_{p_1}. \tag{6}$$

This linearization respects the partial order based on the happens-before relation: program order is respected, so is the relation between sends and their corresponding receives. However, this linearization breaks the first-in-first-out assumption on channels. FIFO is broken because, in order for p_2 to read from z, it must be that it received the value of 1 from the channel. But p_2 is the first thread to receive from the channel and, since 0 was the first value into the channel, it must also have been the first value read from the channel. Therefore, the linearization in Trace 6 is not "realizable" by the operational semantics. While happens-before restricts the commutation of trace operations, there exist other operations that are ordered (though not ordered by happens-before) and that, consequently, must not commute.

The difficulty in conciliating the commutativity of trace events with the happens-before relation remains counterintuitive today, even though its origins are related to an observation made years ago in a seminal paper by Lamport [18]. In the paper, Lamport points out that "anomalies" can arise when there exist orderings that are external to the definition of happens-before—see the "Anomalous Behavior" section of [18]. In order to avoid these anomalies, one suggestion from the paper is to expand the notion of happens-before so that, if a and b are necessarily ordered, then a and b are also in happens-before.

Let us analyze the consequences of rolling FIFO notions into the definition of happens-before. Given the example of Figure 13, since the sends are ordered in a happens-before relation, and the channel is FIFO, one can argue that the receive events should also be ordered by happens-before. According to this argument, we ought to promote the dotted line in Figure 14 to a solid $\rightarrow_{hb}$ arrow. This modification would make the example well-synchronized. In one hand, given that the write to z by p_1 and the read from z by p_2 are *always* separated by events (by the two receive events in specific), interpreting the two memory accesses as being synchronized seems rather fitting: the two memory accesses cannot happen simultaneously, nor can they exist side-by-side in a trace.

There are downsides to this approach. For one, the resulting semantics deviates from Go's, but, more importantly, such a change does impact synchronization in counter intuitive ways. Specifically, making the dotted arrow a happens-before arrow would imply that a receiver (in this case p_2) can learn about prior events that are not known by the corresponding sender. If the dotted arrow is promoted to a synchronization arrow, the write $(z!)_{p_1}$ is communicated to p_2 via p_0 without p_0 itself being "aware" of the write. In other words, the write identifier is transmitted via p_0 but is not present in p_0's happens-before set.

We follow Go and allow for some events to always occur in order without affecting synchronization. Consequently, such ordered events are not considered to be in *happens-before* order. A less clear consequence, however, is that races can longer be defined as simultaneous (or side-by-side) accesses to a shared variable. This point is explored next.

6.2 Manifest data races

Section 2 mentioned the concept of manifest data race; below we give a concrete definition.

Definition 1 (Manifest data race). *A well-formed configuration R contains a* mani-fest data race *if either hold:*

$$R \xrightarrow{(z!)_{p_1}} \text{ and } R \xrightarrow{(z!)_{p_2}} \qquad\qquad \text{(manifest write-write race on z)}$$

$$R \xrightarrow{(z?)_{p_1}} \text{ and } R \xrightarrow{(z!)_{p_2}} \qquad\qquad \text{(manifest read-write race on z)}$$

for some $p_1 \neq p_2$.

Manifest data races can also be defined on traces.

Definition 2 (Manifest data race). *A well-formed trace h contains a* manifest data race *if either*

$$(z!)_{p_1} \ (z!)_{p_2} \qquad\qquad \text{(manifest write-after-write)}$$
$$(z!)_{p_1} \ (z?)_{p_2} \qquad\qquad \text{(manifest read-after-write)}$$
$$(z?)_{p_1} \ (z!)_{p_2} \qquad\qquad \text{(manifest write-after-read)}$$

are a sub-sequence of h and where $p_1 \neq p_2$).

While manifest races are obvious, races in general may involve accesses that are arbitrarily "far apart" in a linear execution. By bring conflicting accesses side-by-side, we could show irrefutable evidence of a race that, otherwise, may be obscured in a trace. Let $h \sqsubseteq h'$ represent the fact that h' is derivable from h by the repeated commutation of adjacent pairs of independent operations. If $h \sqsubseteq h'$ and h' contains a manifest data race, then we say h contains a data-race. This definition of races seems unequivocal. From here, soundness and completeness of a race detector may be defined as such:

Theorem 3. *(Soundness) If $S_0 \xrightarrow{h}$ is a run flagged by a data-race detector, then $h \sqsubseteq h_{dr}$ with h_{dr} containing a manifest data-race.*

Theorem 4. *(Completeness) Let $S_0 \xrightarrow{h}$ be a run such that $h \sqsubseteq h_{dr}$ and h_{dr} contains a manifest race. Then $S_0 \xrightarrow{h}$ is flagged by the data-race detector.*

Theorems 3 and 4 are also clear and unequivocal. More importantly, they link two world views: the view of races as unsynchronized accesses with respect to the happens-before relation and a view of races in terms of commutativity of trace events *à la* Mazurkiewicz. The problem with the concept of manifest data race and Theorems 3 and 4, however, is that when the definition of independence is made to respect FIFO or-der as well as the happens-before relation, the notion of manifest data race is no longer attainable. In other words, given a definition of independence which respects FIFO and happens-before, there exist racy traces from which a manifest data race is not derivable.

The program of Figure 13 gives rise to such an example. The access to z by p_2 only occurs if p_2 receives the second message sent on the channel. In other words, the existence of event $(z?)_{p_2}$ in a trace is predicated on the order of execution of chan-nel operations: p_2 only reads from z if the other thread, p_1, receives from c before p_2

does.[20] This requirement places the receive operations between the memory operations. Therefore, a trace in which $(z!)_{p_1}$ and $(z?)_{p_2}$ are side-by-side is not attainable. Yet, as discussed previously, the accesses to z are not ordered by happens-before, and, therefore, are concurrent. Since the accesses are also conflicting, they constitute a data race.

It seems that Mazurkiewicz traces are "more compatible" with confluence checking than data-race checking. In data-race checking, there are non-confluent runs that do not exhibit data races; these runs are non-confluent because they have "races on channels." In our example, the two receives from p_1 and p_2 are in competition for access to the channel. These receive operations are concurrent and non-confluent. Finally, the example also hints at the perhaps more fundamental observation: that races have little to do with *simultaneous* accesses to a shared variable but instead with *unsynchronized* accesses. While simultaneous accesses are clearly unsynchronized, not all unsynchronized accesses may be made simultaneous.[21]

7 Related work

Race detection via the analysis of source code is an undecidable problem. Regardless, race detectors via the static analysis of source code [24, 39, 3] exist and have found application in industry. More recently, Blackshear et al. [3] implement a static analysis tool called RACERD to help the parallelization of previously sequential Java source code. The tool over approximates the behavior of programs and can, thereby, reject programs that turn out to be data-race free. This over approximation was not a hindrance, as even conservative parallelization efforts can lead to gains over purely sequential code.

By and large, however, instead of flagging races in a program as a whole, race detectors have resorted the analysis of particular *runs* of a program. To that end, detectors instrument the program so that races are either flagged during execution, in what is called *on-line* or *on-the-fly* race detection, or on logs captured during execution and analyzed *postmortem*. Even still, dynamic race detection is NP-hard [25] and many techniques have been proposed for detection at scale. Broadly, these techniques involve static analysis used to reduce the number of runtime checks [11][31], and heuristics that trade false-positive [33, 30, 5] or false-negative rates [21] for better space/time utilization. For example, by allowing races to sometimes go undetected, *sampling* race detectors let go of completeness in favor of lower overheads. One common heuristic, called the *cold region hypothesis*, is to sample more frequently from less executed regions of the program. This rule-of-thumb hinges on the assumption that faults are more likely to already have been identified and fixed if they occur in the hot regions of a program [21]. Alternatively, by going after a proxy instead of an actual race, imprecise race detectors let go of soundness. The prominent examples here are Eraser's LockSet [33]

---

[20] In this example, we use the value of the message received on a channel to branch upon. But since a receive from a channel changes a thread's "visibility" of what is in memory, it is possible to craft a similar example in which all message values are `unit` but in which a thread's behavior changes due to a change in the ordering of the receives.

[21] There may not exist a configuration from which two transitions are possible; transitions that involve conflicting memory accesses. Yet, it is possible for two access separated "in time" to be unsynchronized.

and Locksmith [30], which enforce a lock-based synchronization discipline. A violation of the discipline is a *code smell* but not necessarily a race. The amalgamation of different approaches have also been investigated, leading to *hybrid race detectors*. For example, O'Callahan and Choi [26] combined LockSet-based detection with happens-before information reconstructed from vector clocks; Choi et al. [5] extended LockSet to incorporate static analyses.

Another avenue of inquiry has lead to *predictive race detection* [36, 16], which attempts to achieve higher detection capabilities by extrapolating beyond individual runs. Huang et al. [16] incorporate abstracted control flow information and formulate race detection as a constraint solving problem. With the goal of observing more races per run, Smaragdakis et al. [36] introduce a new relation, called *causally-precedes*, which is a generalization of the happens-before relation.

A number of papers address race detection in the context of channel communication [6, 7, 38]. Some of the papers, however, do not speak of shared memory but, instead, define races as conflicting channel accesses. In that setting, the lack of conflicting accesses to channels imply determinacy. A different angle is taken by Terauchi and Aiken [38], who, among different kinds of channels, define a buffered channel whose buffer is overwritten by every write (i.e. send) but never modified by a read (i.e. receive). This kind of channel, referred to as a *cell*, behaves, in essence, as shared memory. The goal of Terauchi and Aiken [38] is, still, determinacy. Having conflated the concept of shared memory as a channel, determinacy is then achieved by ensuring the absence of conflicting accesses to channels. Our goal, however, is different: we aim to detect data-races but do not want to go as far as ensuring determinacy. Therefore, our approach allows "races" on channel accesses. From a different perspective, however, the work of Terauchi and Aiken [38] can be seen as complementary to ours: We conjecture that their type system can serve as the basis for a static data-race detector.

Among the dynamic data-race detection tools from industry, Banerjee et al. [1] discuss different race detection algorithms including one used by the Intel Thread Checker. The authors describe *adjacent conflicts*, which is similar to our notion of side-by-side or manifest data race. The paper also classifies races similar to our WaR, RaW, and WaW classification.

Go has a race detector integrated to its tool chain [13]. The `-race` command-line flag instructs the Go compiler to instrument memory accesses and synchronization events. The race detector is built on top of Google's sanitizer project [14] and TSan in particular [34, 15]. TSan is part of the LLVM's runtime libraries [35, 20]; it works by instrumenting memory accesses and monitoring locks acquisition and release as well as thread forks and joins. Note, however, that channel communication is the vehicle for achieving synchronization in Go. Even though locks exist, they are part of a package, while channels are built into language. Yet, the race detector for Go sits at a layer underneath. In this paper we study race detection with channel communication taking a central role. Also, different from TSan, we employ propose a technique based on what we call *happens-before sets* as opposed to vector clocks. The consequences of this decision is discussed in detail on Section 5.

It is also relevant to point out that, in the absence of the DRF-SC guarantee, one may resort to finding data races involving weak memory behavior. Since the full C/C++11

memory model can harbor such races, and with the goal of finding data races in production level code, Lidbury and Donaldson [19] extend the ThreadSanitizer (TSan) [34, 15] to support a class of non-sequentially consistent executions.

8 Conclusion

We presented a dynamic data-race detector for a language in the style of Go: featuring channel communication as sole synchronization primitive. The proposed detector records and analyzes information locally and is well-suited for online detection.

Our race detector is built upon a previous result [10], where we formalize a weak memory model inspired by the Go specification [12]. In that setting, we recorded memory read- and write-events that were in happens-before relation with respect to a thread's present operation. This information was stored in a set called E_{hb} or the happens-before set of a thread, and it was used to regulate a thread's visibility of memory events. The core of the paper was a proof of the DRF-SC guarantee, meaning, we proved that the proposed relaxed memory model behaves sequentially consistently in the absence of data races. The proof hinges on the fact that, in the absence of races, all threads agree on the contents of memory; see the *consensus lemma* in [10]. The scaffolding used in the proof of the consensus lemma contains the ingredients used of the race detectors presented in this paper. Based on our experience, we conjecture that one may automatically derive a race detector given a weak memory model and its corresponding proof of the DRF-SC guarantee.

In the DRF-SC the proof of [10], we show that if a program is racy, it behaves sequentially consistent up to the point in which the first data-race is encountered. In other words, this first point of divergence sets in motion all behavior that is not sequentially consistent and which arise from the weakness in the memory model. With this observation, we argue that a race detector can operate under the assumption of sequential consistency. This is a useful simplification, as sequential consistent memory is conceptually much simpler than relaxed memories. If the data-race detector flags the first evidence of a data-race, then program behavior is sequentially consistent up to that point.

Avenues for future work abound. In contrast to data-race detectors based on vector clocks, our approach using happens-before sets does not provide as terse of a representation for the collection of memory events performed by a thread in between synchronization points. In effect, out approach has a larger foot-print, which ought to be mitigated. On the other hand, our thorough expunging of stale information can serve as inspiration to vector clock based approaches, which allow for the accumulation of stale information—see Section 6. Another extension would be to statically analyze a target program with the goal of removing dynamic checks or ameliorating the detector's memory consumption. Here, we may be able to borrow from the research on static analysis for dynamic race-detection in the context of lock-based synchronization disciplines.

Bibliography

[1] Banerjee, U., Bliss, B., Ma, Z., and Petersen, P. (2006). A theory of data race detection. In *Proceedings of the 4th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), PADTAD 2006, Portland, Maine, USA, July 17, 2006*, pages 69–78.

[2] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83.

[3] Blackshear, S., Gorogiannis, N., O'Hearn, P. W., and Sergey, I. (2018). Racerd: compositional static race detection. *PACMPL*, 2(OOPSLA):144:1–144:28.

[4] Channel types, Go language specification (2016). Channel types, the Go programming language specification. `https://golang.org/ref/spec#Channel_types`.

[5] Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. (2002). Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) Berlin, Germany*, pages 258–269. ACM.

[6] Cypher, R. and Leu, E. (1995). Efficient race detection for message-passing programs with nonblocking sends and receives. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 534–541. IEEE.

[7] Damodaran-Kamal, S. K. and Francioni, J. M. (1993). Nondeterminacy: testing and debugging in message passing parallel programs. *ACM SIGPLAN Notices*, 28(12):118–128.

[8] Dijkstra, E. W. (n.d.). Over de sequentialiteit van procesbeschrijvingen. Circulated privately.

[9] Fava, D. (2020). Grace: a race detector based on happens-before sets. `https://github.com/dfava/grace`.

[10] Fava, D., Steffen, M., and Stolz, V. (2018). Operational semantics of a weak memory model with channel synchronization. *Journal of Logic and Algebraic Methods in Programming*. An extended version of the FM'18 publication with the same title.

[11] Flanagan, C. and Freund, S. N. (2009). FastTrack: Efficient and precise dynamic race detection. In Hind, M. and Diwan, A., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM.

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

[13] golang.race.detector (2013). `https://blog.golang.org/race-detector`.

[14] google.sanitizer (2014). `https://github.com/google/sanitizers`.

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

[16] Huang, J., Meredith, P. O., and Rosu, G. (2014). Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 337–348. ACM.

[17] Katz, S. and Peled, D. (1992). Defining conditional independence using collapses. *Theoretical Computer Science*, 101.

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

[19] Lidbury, C. and Donaldson, A. F. (2017). Dynamic race detection for C++11. In Castagna, G. and Gordon, A. D., editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457. ACM.

[20] llvm.thread.sanitizer (2011). `https://clang.llvm.org/docs/ThreadSanitizer.html`.

[21] Marino, D., Musuvathi, M., and Narayanasamy, S. (2009). Literace: effective sampling for lightweight data-race detection. In *ACM Sigplan notices*, pages 134–143.

[22] Mattern, F. (1988). Virtual time and global states in distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Algorithms*, pages 215–226.

[23] Mazurkiewicz, A. (1987). Trace theory. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, (Advances in Petri Nets 1986) Part II*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer Verlag.

[24] Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM.

[25] Netzer, R. H. B. and Miller, B. P. (1990). On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 2: Software.*, pages 93–97.

[26] O'Callahan, R. and Choi, J.-D. (2003). Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, pages 167–178.

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

[28] Peters, K. and Nestmann, U. (2012). Is it a "good" encoding of mixed choice? In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '12)*, volume 7213 of *Lecture Notes in Computer Science*, pages 210–224. Springer Verlag.

[29] Pozniansky, E. and Schuster, A. (2003). Efficient on-the-fly data race detection in multi-threaded C++ programs. In *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*.

[30] Pratikakis, P., Foster, J. S., and Hicks, M. W. (2006). LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 320–331. ACM.

[31] Rhodes, D., Flanagan, C., and Freund, S. N. (2017). Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 141–156.

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

[33] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

[34] Serebryany, K. and Iskhodzhanov, T. (2009). Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM.

[35] Serebryany, K., Potapenko, A., Iskhodzhanov, T., and Vyukov, D. (2011). Dynamic race detection with llvm compiler. In *International Conference on Runtime Verification*, pages 110–114. Springer.

[36] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., and Flanagan, C. (2012). Sound predictive race detection in polynomial time. In *Proceedings of POPL '12*, pages 387–400. ACM.

[37] Steffen, M. and Nestmann, U. (1995). Typing confluence. Interner Bericht IMMD7-xx/95, Informatik VII, Universität Erlangen-Nürnberg.

[38] Terauchi, T. and Aiken, A. (2008). A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):27.

[39] Voung, J. W., Jhala, R., and Lerner, S. (2007). RELAY: Static race detection on millios of lines of code. In *Proceedings oof the 6th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214.

A Strong semantics

For completeness sake and for reference, we include here the operational semantics *without* augmenting it with any information relevant for race checking. It is thereby a conventional operational semantics and corresponds to the strong semantics from [10].

$$e; t \triangleq \texttt{let } r = e \texttt{ in } t \quad \text{when } r \notin fv(t))$$
$$\texttt{stop} \triangleq \Sigma_0$$

Fig. 15: Syntactic sugar

The surface syntax is unchanged from Figure 1. The operational semantics is formulated using run-time configurations as given in equation (7).

$$R ::= \langle t \rangle \mid (\![z{:}{=}v]\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R \,. \tag{7}$$

For race detection, we used the "same" run-time syntax, except that they were augmented with additional information (cf. equation for the intermediate formulation (3) of the race detecting semantics resp. equation (4). Compared to the race detecting semantics, the configurations carry less information. In particular, the recorded events don't carry identifying labels and threads don't keep track of happens-before information as for the race checker.[22]

A.1 Structural congruence

Configurations are interpreted up-to structural congruence, only: Parallel composition is associative and commutative, with the empty configuration as neutral element. The ν-binder is used to manage the scopes for dynamically created names. Besides that, syntax is considered tacitly up-to renaming of bound names, in particular, ν-bound names.

Dynamically created names are channel names. In the augmented semantics, where processes are named and also events carry a label, also names for those entities can be created on-the-fly and they are subject to the congruence rules for ν-bound names.

A.2 Local steps

The rules from Figure 16 concern reduction steps that don't affect the memory or involve channel communication.

---

[22] Note in passing, also in the formalization of the weak semantics in [10], the threads keep track of happens-before information. Here, the additional information is needed to do race detection on the strong semantics, where the semantics itself works without that information, whereas in [10], the additional information is required to describe the (weak) semantics itself.

$$R_1 \parallel R_2 \equiv R_2 \parallel R_1$$
$$(R_1 \parallel R_2) \parallel R_3 \equiv R_1 \parallel (R_2 \parallel R_3)$$
$$\bullet \parallel R \equiv R$$
$$R_1 \parallel \nu n\, R_2 \equiv \nu n\, (R_1 \parallel R_2) \qquad \text{if } n \notin \mathit{fn}(R_1)$$
$$\nu n_1\, \nu n_2\, R \equiv \nu n_2\, \nu n_1\, R$$

Table 1: Structural congruence

$\mathtt{let}\, x = v \,\mathtt{in}\, t \rightsquigarrow t[v/x]$ R-RED

$\mathtt{let}\, x_1 = (\mathtt{let}\, x_2 = e \,\mathtt{in}\, t_1) \,\mathtt{in}\, t_2 \rightsquigarrow \mathtt{let}\, x_2 = e \,\mathtt{in}\, (\mathtt{let}\, x_1 = t_1 \,\mathtt{in}\, t_2)$ R-LET

$\mathtt{if}\, \mathtt{true}\, \mathtt{then}\, t_1 \,\mathtt{else}\, t_2 \rightsquigarrow t_1$ R-COND$_1$ $\qquad$ $\mathtt{if}\, \mathtt{false}\, \mathtt{then}\, t_1 \,\mathtt{else}\, t_2 \rightsquigarrow t_2$ R-COND$_2$

Fig. 16: Local steps

A.3 Memory interactions and channel communication

Reading and writing, the two basic memory interactions, are covered in Figure 17 and channel communication in Figure 18. Compared to the semantics for race detection (cf. Figures 6 and 4), the semantics here is done without extra information and book-keeping of happens-before information. Related to that, the recorded events don't carry any names to identify the event.

$\langle z := v'; t \rangle \parallel (\!|z{:=}v|\!) \rightarrow \langle t \rangle \parallel (\!|z{:=}v'|\!)$ R-WRITE

$\langle \mathtt{let}\, r = \mathtt{load}\, z \,\mathtt{in}\, t \rangle \parallel (\!|z{:=}v|\!) \rightarrow \langle \mathtt{let}\, r = v \,\mathtt{in}\, t \rangle \parallel (\!|z{:=}v|\!)$ R-READ

Fig. 17: Read and write steps

For the channel communication in Figure 18, no more happens-before information is communicated. Especially the forward channel carries only the communicated value v (cf. rule R-SEND and R-REC). To realize the boundedness of the channels, the semantics still maintains the two parts of a channel: the forward channel for communication, and the backward channel for "flow-control." The backward channel c_b does not carry any information, just the number of entries representing still empty slots in the forward channel. We use the unit value () for that, and initially, the backward channel is filled with a number of ()'s corresponding to the capacity of the channel (see rule R-MAKE).

For channel communication, the semantics distinguished between synchronous communication, i.e., a "rendezvous" over a channel of capacity 0, and asynchronous com-

munication, with a channel of non-zero, but finite capacity. For the asynchronous case, both sending and receiving

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

$$\frac{\neg \textit{closed}(c_f[q_2])}{c_b[q_1 :: ()] \parallel \langle c \leftarrow v; t \rangle \parallel c_f[q_2] \;\; \rightarrow \;\; c_b[q_1] \parallel \langle t \rangle \parallel c_f[v :: q_2]} \text{ R-SEND}$$

$$\frac{v \neq \bot}{\begin{array}{l} c_b[q_1] \parallel \langle \texttt{let } r = \leftarrow c \texttt{ in } t \rangle \parallel c_f[q_2 :: v] \;\; \rightarrow \\ c_b[E_{hb} :: q_1] \parallel \;\; \langle \texttt{let } r = v \texttt{ in } t \rangle \;\; \parallel c_f[q_2] \end{array}} \text{ R-REC}$$

$$\frac{}{\langle \texttt{let } r = \leftarrow c \texttt{ in } t \rangle \parallel c_f[\bot] \;\; \rightarrow \;\; \langle \texttt{let } r = \bot \texttt{ in } t \rangle \parallel c_f[\bot]} \text{ R-REC}_\bot$$

$$\frac{}{\begin{array}{l} c_b[] \parallel \langle c \leftarrow v; t \rangle \parallel \langle \texttt{let } r = \leftarrow c \texttt{ in } t_2 \rangle \parallel c_f[] \;\; \rightarrow \\ c_b[] \parallel \;\;\;\; \langle t \rangle \;\;\;\; \parallel \langle \texttt{let } r = v \texttt{ in } t_2 \rangle \;\;\; \parallel c_f[] \end{array}} \text{ R-REND}$$

$$\frac{\neg \textit{closed}(c_f[q])}{\langle \texttt{close } (c); t \rangle \parallel c_f[q] \;\; \rightarrow \;\; \langle t \rangle \parallel c_f[\bot :: q]} \text{ R-CLOSE}$$

Fig. 18: Channel communication

B Rules for the select statement

Rules dealing with the select statement semantics are given on Figure 19. The R-SEL-SEND and R-SEL-REC rules apply to asynchronous channels and are analogous to R-SEND and R-REC. The R-SEL-SYNC rules apply to open synchronous channels (i.e. the forward and backward queues are empty). The R-SEL-REC$\bot$ is analogous to R-REC$\bot$. Finally, the default rule (R-SEL-DEF) applies when no other select rule applies.

$$\frac{g_i = c \leftarrow v \qquad \neg closed(c_f[q_f]) \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{l} c_b[q_b :: (E''_{hb})] \parallel p\langle E_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel c_f[q_f] \quad \rightarrow \\ \qquad c_b[q_b] \parallel \qquad p\langle E'_{hb}, t_i[()/r_i] \rangle \qquad \parallel c_f[(v, E_{hb})) :: q_f] \end{array}} \text{ R-SEL-SEND}$$

$$\frac{g_i = \leftarrow c \qquad q_f = q'_f :: (v, E''_{hb}) \qquad v \neq \bot \qquad q'_b = (E_{hb}) :: q_b \qquad E'_{hb} = E_{hb} + E''_{hb}}{\begin{array}{c} c_b[q_b] \parallel p\langle E_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel c_f[q_f] \quad \rightarrow \\ c_b[q'_b] \parallel \quad p\langle E'_{hb}, \mathtt{let}\ r_i = v\ \mathtt{in}\ t_i \rangle \quad \parallel c_f[q'_f] \end{array}} \text{ R-SEL-REC}$$

$$\frac{g_i = c \leftarrow v \qquad E_{hb} = E'_{hb} + E''_{hb} \qquad c_b[] \qquad c_f[]}{\begin{array}{c} p_1\langle E'_{hb}, \sum_i r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel p_2\langle E''_{hb}, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t_2 \rangle \quad \rightarrow \\ p_1\langle E_{hb}, t_i[()/r_i] \rangle \parallel p_2\langle E_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t_2 \rangle \end{array}} \text{ R-SEL-SYNC}_1$$

$$\frac{g_i = \leftarrow c \qquad E_{hb} = E'_{hb} + E''_{hb} \qquad c_b[] \qquad c_f[]}{\begin{array}{c} p_1\langle E'_{hb}, c \leftarrow v; t_1 \rangle \parallel p_2\langle E''_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \quad \rightarrow \\ p_1\langle E_{hb}, t_1 \rangle \parallel p_2\langle E_{hb}, \mathtt{let}\ r_i = v\ \mathtt{in}\ t_i \rangle \end{array}} \text{ R-SEL-SYNC}_2$$

$$\frac{g_i = c \leftarrow v \qquad g_j = \leftarrow c \qquad E_{hb} = E'_{hb} + E''_{hb} \qquad c_b[] \qquad c_f[]}{\begin{array}{c} p_1\langle E'_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel p_2\langle E''_{hb}, \sum_j \mathtt{let}\ r_j = g_j\ \mathtt{in}\ t_j \rangle \quad \rightarrow \\ p_1\langle E_{hb}, t_i[()/r_i] \rangle \parallel p_2\langle E_{hb}, \mathtt{let}\ r_j = v\ \mathtt{in}\ t_j \rangle \end{array}} \text{ R-SEL-SYNC}_3$$

$$\frac{g_i = \leftarrow c \qquad c_f[(\bot, E''_{hb})] \qquad E'_{hb} = E_{hb} + E''_{hb}}{p\langle E_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \quad \rightarrow \quad p\langle E'_{hb}, \mathtt{let}\ r_i = \bot\ \mathtt{in}\ t_i \rangle} \text{ R-SEL-REC}_\bot$$

$$\frac{g_i = \mathtt{default} \qquad \neg \exists j.\ i \neq j.\ p\langle E_{hb}, \sum_j \mathtt{let}\ r_j = g_j\ \mathtt{in}\ t_j \rangle \parallel P \rightarrow p\langle E'_{hb}, t' \rangle \parallel P'}{p\langle E_{hb}, \sum_i \mathtt{let}\ r_i = g_i\ \mathtt{in}\ t_i \rangle \parallel P \quad \rightarrow \quad p\langle E_{hb}, t_i[()/r_i] \rangle \parallel P} \text{ R-SEL-DEF}$$

Fig. 19: Operational semantics: Select statement