Ready, set, Go!

Sound and complete data-race detection in the context of message passing

Daniel Schnetzer Fava
danielsf@ifi.uio.no

Martin Steffen
msteffen@ifi.uio.no

Dept. of Informatics, University of Oslo

Abstract. Most papers in the literature address synchronization in the form of lock acquisition and release, where relevant trace events (besides reads to and writes from memory) are acquire, release, fork, and join. We present a data-race detector for a language with channel communication as its sole synchronization primitive, and prove soundness and completeness of the detector.

1 Introduction

One way to deal with complexity is by partitioning a system into cooperating subcomponents. When these subcomponents compete for resources, coordination becomes a prominent goal. One common programming paradigm is to have threads cooperate 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* is also well studied in the absence of shared memory; the goal in these cases is to achieve determinism rather than race-freedom [2, 3, 24]. 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 sound and complete 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 2 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 the event also involves tracking *happens-before* information. The happens-before relation is instrumental to the underlying memory model, which factually is *weak* or relaxed. We should point out, however, that the operational semantics presented here and used for race detection is *not* a weak semantics. 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. Earlier we established that a corresponding weak semantics[1] enjoys the crucial DRF-SC property [5], meaning that for data-race free programs, memory behaves *sequentially consistent*. Therefore, when it comes to race detection, it suffices to concentrate on a sequentially consistent or "strong" memory behavior.

The remaining of the paper is organized as follows. Below we introduce a race-detector in the context of channel communication as sole synchronization mechanism. Section 3 introduces a trace grammar and defines data-races on execution histories. An independence relation on events (technically on event labels) is defined which allows us to reason about equivalent histories. Soundness and completeness of the data-race detector is then proven in Section 4 by relating the execution of a program in the operational semantics to histories according to the trace grammar. Section 5 examines related work and 6 provides a conclusion and touches on future work.

2 Data-race detection

We start in Section 2.1 by presenting the abstract syntax of our calculus. Afterwards, we present a corresponding operational semantics which can be used as a race detector. We present the race detector incrementally. After a short general introduction in Section 2.2, Section 2.3 starts with a simpler detector that is not complete, meaning that some types of races may unnecessarily go unnoticed. We build onto this first iteration of the detector in Section 2.4 in order to derive a sound and complete detector – complete in the given happens-before interpretation of a race. Sections 2.3 and 2.4 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.

2.1 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 [5]. *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 $r_1 + r_2$. Shared variables are denoted by x, z, etc., `load` z represents reading the shared

[1] Note that while the mentioned semantics of [5] 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 [5], 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 sound and complete race detector. Both versions of the semantics are connected by the DRF-SC result, as mentioned.

variable z into the thread, and $z := v$ denotes writing to z. References are dynamically created and are, therefore, part of the *run-time* syntax. Run-time syntax is highlighted in the grammar with an underline as in $\underline{n}$. A new channel is created by $\texttt{make}\,(\texttt{chan}\,T, v)$, 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 over a channel and receiving a value as input from a channel are denoted respectively as $v_1 \leftarrow v_2$ and $\leftarrow v$. After the operation `close`, no further values can be sent on the specified channel. Attempting to send values on a closed channel leads to a panic.

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 impose any actual reduction in expressivity and corresponds to an A-normal form representation [20]. 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" [16, 17] 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.

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

Fig. 1: Abstract syntax

The `let`-construct $\texttt{let}\,r = e\,\texttt{in}\,t$ 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 is seen 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 16 in the appendix for syntactic sugar.

2.2 Read and write conflicts

Races are, roughly speaking, "simultaneous" interactions on a shared memory location. To make such simultaneous accesses problematic, at least one of the accesses has to be a write, i.e., there are no read-read conflicts. A race *manifests* itself in an execution if a step is immediately after another and the two are conflicting. This 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 constitute consecutive conflicting accesses, races in general may involve accesses that are arbitrarily "far apart" temporally. To be able to report more than the most obvious races, a race checker needs to keep extra information. The amount of "non-manifest" data-races it is able to report depends on the amount of information the detector maintains. In general, recording more information leads to a higher degree of "completeness" at the expense of a higher run-time overhead.

As far as conflicts goes, there are read-write conflicts and write-write conflicts. When running, the information the detector keeps naturally pertains to "historic" events; it does not make use of prophetic or speculative information about future events. Taking this temporal aspect into account, one can make a more fine-grained distinction between read-after-write (RaW), write-after-write (WaW), and write-after-read (WaR) conflicts. In the "after-write" situation, one has to remember additional information concerning write-events in order to know whether a subsequent access to the same variable occurs without proper synchronization. (See Section 2.3 for dealing with read-after-write and write-after-write races.) Basically, when accessing a variable, it must be checked that all write events to the same variable *happened-before* the current write access, where current means from the perspective of the thread attempting the access. The question of whether an event occurred in the "definite past" (i.e., whether an event is in happened-before with "now") is *thread-local*.

To capture these notions of ordering in an operational setting, the unadorned operational semantics is equipped with additional information: each thread tracks all write events it is aware of having happened-before (see the component E_{hb} in the run-time configurations of equation (1) below). This is not the only augmentation when comparing the run-time configuration of the undecorated semantics (see equation (7)). When it comes to race detection, it is not enough to store the last value written to each variable; we also need to identify the event that has lead to that last value. In the undecorated semantics, the content v of a variable z is written as a pair $(\!|z{:=}v|\!)$. In the augmented semantics for detecting after-write conflicts, it takes the form $m(\!|z{:=}v|\!)$ where m uniquely identifies the event associated with v having been stored into z (cf. the run-time configurations in equation (7) and equation (1)).

On top of RaW and WaW conflicts covered in Section 2.3, Section 2.4 describes the detection of *write-after-read* conflicts. Consequently, in addition to information about past write events, the race checker needs to remember information about past read events. Abstractly, a read event symbolizes the fact that a load-statement has executed. In the strong semantics, as discussed, a read always observes one definite value which is the result of one particular write event (in contrast to a weak semantics where there may be ambiguities when reading [5]). To check for RaW conflicts, the race checker needs to remember the read events that observe a particular write.

For the presentation of the race checker, it is convenient to group the read-events together with the write event they pertain to. There may be more than one read event associated with a write, as write events can be read multiple times before the variable is overwritten. Therefore, the configuration contains entries of the form $w(\!|z{:=}v|\!)\mathcal{L}^R$ where w is the identifier of the write event and $\mathcal{L}^R$ is a set of identifiers of read events, namely those that accumulated after w. The "records" of the form $w(\!|z{:=}v|\!)\mathcal{L}^R$ can be seen as $n+1$ recorded events, 1 write event together with $n \geq 0$ read-events. In the terminology used in axiomatization of memory models, it can also be seen as a representation of the read-from relation, representing $r_i \rightarrow_{\mathsf{rf}} w$ for all $r_i \in \mathcal{L}^R$ from an element $w(\!|z{:=}v|\!)\mathcal{L}^R$. Note that, when saying that "the remembered read events occurred *after* the write event they read-from," we do not mean that the read-events *happened-after* the write. Instead, they just, incidentally or not, occurred afterwards in an execution.

2.3 Detecting read-after-write (RaW) and write-after-write (WaW) conflicts

For detecting "after-write" conflicts, run-time configurations are given by the following syntax:

$$R ::= p\langle E_{hb}, t\rangle \mid m(\!|z{:=}v|\!) \mid \bullet \mid R \parallel R \mid c[q] \mid \nu n\, R\,. \tag{1}$$

The current state of the memory is represented as $m(\!|z{:=}v|\!)$ which associates the value v with the shared variable z. Given that the identifier m is unique, term $m(\!|z{:=}v|\!)$ can be seen not just as storing the current value for z but also as recording a write *event*, namely, the write event responsible for storing v in z.[2] The 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).

An initial configuration starts with one write-record per variable. The semantics maintains this one-to-one mapping as an invariant, which mean that the collection of write-records behave as a mapping from variable to values.[3] When a variable is written to, the record associated with the variable is updated (alternatively, we can interpret it as an old event being supplanted by a write event with a fresh identity).

Threads t under execution are 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

[2] We will later use the term event when talking about histories or traces. There, events carry slightly different information. For instance, as we are interested in the question whether a history contains evidence of a race, it won't be necessary to mention the actual value of 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 as an element of the configuration. When being precise, we refer to a configuration event as *recorded* event or just *record*. 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 $m(\!|z{:=}v|\!)$.

[3] Behaving like a mapping is in line with the fact that the race checker works under the assumption of a strong memory model.

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 just having occurred in a prior step. A memory event that occurred in a prior step but is not in happens-before relation with the current thread state counts as "concurrent" and is, thereby, potentially in conflict with the thread's next step. More precisely, if $m(\!|z{:=}v|\!)$ is part of the configuration, then a $p\langle E_{hb}, t\rangle$ is allowed to safely write to z if $(m,z) \in E_{hb}$, otherwise a WaW conflict is raised. Similar when reading from a variable.

Knowledge of past events contained in E_{hb} is naturally monotonously increasing: each time a goroutine learns about happens-before information, it adds to its pool of knowledge. In particular, events that are know to have "happened-before" cannot, by learning new information, become "concurrent" and potential candidates for conflicts. The semantics, however, does not just accumulate happens-before information. Instead, for efficiency's sake, it purges outdated information. This "garbage collection" is done in the premise of rule R-WRITE: after the write step, the identity m is no longer part of the configuration. There is no need to remember m as it is enough to remember that m' has happened-before in the post-configuration. Thus, m can be removed from E_{hb}. The removal of happens-before information will become more pronounced in Section 2.4, when E_{hb} contains also information about reads.

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

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

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

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

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

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. Data-races are marked as a transition to E in the derivation rules of Figure 2 and later in Figure 5.

So far, the happens-before information was purely thread-local. Goroutines synchronize via message passing, which means that communication via channels is how happens-before information spreads between the goroutines. The rules for channel communication are given in Figure 3. They will remain unchanged when we cover also RaW conflicts. The exchange of happens-before information via channel communication is also analogous to the treatment of the weak semantics in [5].

Suppose a goroutine p has just updated variable z as captured in the write-record $m(\!|z{:=}v|\!)$. At this point, p is 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 constituting 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) containing the write-record's label and associated variable 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 imply that either p or p' are allowed to update z's value. We can think of the two goroutines as *sharing* the ownership of z. Among p and p', whoever updates z first (re)gains the *exclusive* "ownership" of z.

It may be worth making a parallel with hardware and cache coherence protocols. Given the derivation rules above, we can write a race detector as a state machine. Compared to MESI, the semantics above 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* ownership of 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* ownership of the variable. Other goroutines that are unaware of the most recent write can be said to hold *invalid* data.

Finally, goroutine creation is a synchronizing operation; the child inherits the happens-before set from the parent; see Figure 4.

2.4 Detecting write-after-read (WaR) conflicts as well

In the previous section we showed the detection of read-after-write and write-after-write races. There, it sufficed for the recorded event $m(\!|z{:=}v|\!)$ to contain one write-label m. The recorded event was updated with a freshly generated label when a new write took place. The detection of write-after-read races, however, requires more book-keeping. We need read- in addition to write-labels. This additional information 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 equipped to remember all potentially troublesome reads to a shared variable.[4] This extra book-keeping involves an update to the run-time configuration's syntax:

$$R ::= p\langle E_{hb}, t\rangle \;\mid\; m(\!|z{:=}v|\!)\mathcal{L}^R \;\mid\; \bullet \;\mid\; R \parallel R \;\mid\; c[q] \;\mid\; \nu n\, R\,. \tag{2}$$

A set of read-labels $\mathcal{L}^R$ is added to a given variable's record. We will still keep track of one write-label per record. What also remains the same is the process of replacing

[4] 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.

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

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

$$\frac{}{p\langle E_{hb}, \underline{c}; t\rangle \ \rightarrow\ p\langle E_{hb}, t\rangle}\ \text{R-SACK}$$

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

$$\frac{}{p\langle E_{hb}, \mathtt{let}\ r = \underline{v}\ \mathtt{in}\ t\rangle \ \rightarrow\ p\langle E_{hb}, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle}\ \text{R-RACK}$$

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

$$\frac{}{\begin{array}{l} c_b[] \parallel p_1\langle c \leftarrow v; t\rangle \parallel p_2\langle \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t_2\rangle \parallel c_f[] \ \rightarrow \\ c_b[] \parallel \quad p_1\langle t\rangle \quad \parallel p_2\langle \mathtt{let}\ r = v\ \mathtt{in}\ t_2\rangle \quad \parallel c_f[] \end{array}}\ \text{R-REND}$$

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

Fig. 3: Strong operational semantics augmented for race detection: channel communication

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

Fig. 4: Strong operational semantics augmented for race detection: thread creation

a record's write-label when a new write takes place. The novelty is that fresh labels are also generated when reading from memory. 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 associate with z in memory, $\mathcal{L}^{R'} = (m', z) \cup \mathcal{L}^R$. The read operation is captured by rule R-READ in Figure 5. The figure shows memory related reduction rules for WaR race detection. Differences when compared to the detector of Section 2 are highlighted in green.

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 also of all accumulated

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

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

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

$$\frac{(m,z) \in E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = (m',z) \cup E_{hb} \qquad \mathcal{L}^{R'} = (m',z) \cup \mathcal{L}^R}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!)\mathcal{L}^R \to p\langle E'_{hb}, \texttt{let}\ r = \ v\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!)\mathcal{L}^{R'}} \text{ R-READ}$$

$$\frac{(m,z) \notin E_{hb}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|z{:=}v|\!)\mathcal{L}^R \to \mathsf{E}} \text{ R-READ-E}_{RaW}$$

Fig. 5: Strong operational semantics augmented for data-race detection

reads to the variable since the last write. 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 $\mathcal{L}^R$ that is not in the thread's happen-before set — see precondition $\mathcal{L}^R \nsubseteq E_{hb}$ of the R-WRITE-E$_{WaR}$ rule.

When compared to the detector of Section 2.3, rule R-WRITE-E$_{WaW}$ is augmented with the precondition $\mathcal{L}^R \subseteq E_{hb}$. Without this precondition, there would be non-determinism when reporting WaW and WaR conflicts. 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, as mentioned before, a race manifests itself in an execution if a step is immediately after another and the two are conflicting. In this example, the read by p is immediately followed by the write of p'. Therefore, it is reasonable to report the WaR race even though the program contains both WaW and WaR races.

Finally, similar to in the previous section, garbage collection of goroutines's E_{hb} occurs in the R-WRITE rule. The write rule also garbage collects $\mathcal{L}^R$. Garbage collection here is more pressing, since WaR data-race detection involves extra book-keeping when compared to the incomplete detector of Section 2.3. We claim the proposed race-detector is sound and complete; the claim is substantiated in Section 4.

2.5 Example

The classic example of message passing, depicted in Figure 6, involves a producer thread that writes to a shared variable and notifies the completion of the write by send-

ing a message onto a channel. A consumer thread receives from the channel and reads from the shared variable.

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

Fig. 6: Message passing example.

The access to the shared variable in this example is obviously 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 x. The memory record involving x is updated with this fresh label, and the pair (m,x) is placed into p_1's happens-before set, thus yielding $E_{hb}{}_1'$. According to rule R-SEND, the send onto c sends not only the message value, 0 in this case, but also $E_{hb}{}_1'$. The action of receiving from c blocks until a message is available. When a message becomes available, the receiving thread receives not only a message but also the happens-before set of the sender at the time the send took place, see rule R-REC. Thus, upon receiving from c, p_2's happens-before set is updated to contain (m,x). In other words, receiving from the channel places the writing to x by p_1 into p_2's definite past. By inspecting p_2's happens-before set when p_2 attempts to load from x, the race-checker makes sure of the fact that p_2 is aware of p_1 prior write to x.

3 Traces

In Section 2.4 we proposed a race detector and claimed its soundness and completeness. In this section, we introduce a trace grammar for a language with shared memory; the language also features buffered channel communication as its sole synchronization primitive. Later in the section, we define an independence relation on events and a partial order on traces. These relations will provide the ground work from substantiating the soundness and completeness claim of Section 2.4.

The execution of memory and synchronization related operations yield *observable events*; a finite sequence of such events is called a *trace*. Figure 7 introduces a trace grammar where p denotes goroutine- or process-identifiers drawn from $p', p_0, p_1, \ldots$, channel identifiers c are drawn from $c', c_0, c_1, \ldots$, and z are variable names drawn from $z', z_0, z_1, \ldots$ We use $\square = \{!, ?\}$ to represent memory operations; $z\square$ represents either a write $z!$ to or a read $z?$ from variable z. Similarly, $* = \{\texttt{sd}^i, \texttt{rv}^i, \texttt{rv}_\perp, \texttt{close}^i\}$ stands for channel operations send, receive, receive from a closed channel, and close. The index $i \in \mathbb{Z}$ as in $\texttt{sd}^i$, $\texttt{rv}^i$, $\texttt{close}^i$, etc, is described later in this section. In `make` c n, the capacity of the channel being created is captured by $n \in \mathbb{N}$. A new goroutine with identifier p is spawn by `go` p.

A channel is synchronous when its capacity is zero and asynchronous otherwise. When a channel c is synchronous, a sender p and a receiver p' rendezvous in the ex-

$$
\begin{array}{lll}
\square ::= \; ! \mid ? & & \text{memory operations} \\
* ::= \; \mathtt{sd}^i \mid \mathtt{rv}^i \mid \mathtt{rv}_\perp \mid \mathtt{close}^i & & \text{channel operations} \\
o ::= \; \mathtt{z}\square \mid \mathtt{go}\ \ p \mid \mathtt{make}\, c\, n \mid *c & & \text{operations} \\
e ::= \; (o)_p \mid (\mathtt{rend}^i\ \ c)^p_p & & \text{events} \\
h ::= \; [\,] \mid h :: e & & \text{traces}
\end{array}
$$

Fig. 7: Trace grammar

change of a message. The rendezvous is captured by the event $(\mathtt{rend}^i\ \ c)^p_{p'}$. All other event are of the form $(o)_p$, thus involving an operation o performed by one goroutine p.

Technically, an event is unique in a trace. To us, uniqueness is derived from the event's position in the trace. Thus, formally, $h[i]$ stands for the label corresponding to the i^{th} event in h. When clear from the context, we take the liberty of using "event" to refer to an index into a trace or to the label associated with the event.

3.1 Preliminary definitions

Figure 8 introduces the predicates $\in_{\mathbf{p}}$, $\in_{\mathbf{c}}$, $\in_{\mathbf{o}}$, and $\in_{\mathbf{e}}$ which capture the notion of a process, a channel, an operation, and an event being part of a trace. It is also useful to inspect whether an event involving a particular process is part of a trace, $\in^{\mathbf{e}}_{\mathbf{p}}$, or whether a event involving a particular channel is part of a trace, $\in^{\mathbf{e}}_{\mathbf{c}}$.

A process p is available in trace h, denoted as $p \in_{\mathbf{p}} h$, if the process has been created in the trace. Note that the main or initial process p_0 is created by the execution environment before a program starts running, therefore it exists in the empty trace, see rule $\in_{\mathbf{p}}$-MAIN. Otherwise, a process p is in h if the operation $\mathtt{go}\ \ p$ has taken place in h. Similarly, a channel c is in h if $(\mathtt{make}\ c\ n)_p$ is in h for some p. The predicate $e \in_{\mathbf{e}} h$ denotes the existence of an event e in a trace h. An operation o is part of a trace, $o \in_{\mathbf{o}} h$, if there exists a process p such that the event $(o)_p$ is in h or, in the case of the rendezvous operation $\mathtt{rend}^i\ \ c$, if $(\mathtt{rend}^i\ \ c)^p_{p'}$ is in h for some p and p'. The predicate $\in^{\mathbf{e}}_{\mathbf{p}}$ (respectively $\in^{\mathbf{e}}_{\mathbf{c}}$) denotes the existence, in a trace, of an event involving a particular process (respectively channel). Figure 9 shows the projection of an event onto the threads involved in the event, $\lfloor e \rfloor_{\mathbf{p}}$, and the projection of an event onto the operation performed as part of the event, $\lfloor e \rfloor_{\mathbf{o}}$.

Figure 10 defines the function $\#\mathtt{SdRv}\ c\ h$ which captures the number of send events minus receive events on a channel c occurring on a trace h. To that purpose, $|h|$ denotes the length of a trace h and $\mathtt{filter}\ f\ h$ stands for the filter function taking as parameters a function f from event e to boolean and a trace h. The $\mathtt{isSd}\, c\, e$ and $\mathtt{isRv}\, c\, e$ are helper functions: $\mathtt{isSd}\, c\, e$ evaluates to true if e is a send on a channel c; false otherwise. Similar for $\mathtt{isRv}\, c\, e$ and receives. These functions come into play in the definition of a well-formed trace. Given the well-form of a trace $\vdash h$ (see Figure 11), predicates FULL_c and EMPTY_c capture the conditions in which a channel is full and empty respectively. When a channel with capacity n is created, n dummy receive operations

$$\mathtt{rv}^{-n}\ \ c,\ \mathtt{rv}^{-n+1}\ \ c,\ \ldots,\ \mathtt{rv}^{-1}\ \ c$$

$$\frac{}{p_0 \in_{\mathbf{p}} [\,]}\ \in_{\mathbf{p}}\text{-MAIN} \qquad \frac{}{p' \in_{\mathbf{p}} h :: (\texttt{go}\ \ p')_p}\ \in_{\mathbf{p}}\text{-GO} \qquad \frac{p \in_{\mathbf{p}} h}{p \in_{\mathbf{p}} h :: e}\ \in_{\mathbf{p}}\text{-::}$$

$$\frac{}{c \in_{\mathbf{c}} h :: (\texttt{make}\ c\ n)_p}\ \in_{\mathbf{c}}\text{-MAKE} \qquad \frac{c \in_{\mathbf{c}} h}{c \in_{\mathbf{c}} h :: e}\ \in_{\mathbf{c}}\text{-::}$$

$$\frac{}{e \in_{\mathbf{e}} h :: e}\ \in_{\mathbf{e}} \qquad \frac{e \in_{\mathbf{e}} h}{e \in_{\mathbf{e}} h :: e'}\ \in_{\mathbf{e}}\text{-::}$$

$$\frac{}{p \in^{\mathbf{e}}_{\mathbf{p}} h :: (o)_p}\ \in^{\mathbf{e}}_{\mathbf{p}}\text{-EV} \qquad \frac{}{p \in^{\mathbf{e}}_{\mathbf{p}} h :: (\texttt{rend}^i\ \ c)^p_{p'}}\ \in^{\mathbf{e}}_{\mathbf{p}}\text{-REND}$$

$$\frac{}{p' \in^{\mathbf{e}}_{\mathbf{p}} h :: (\texttt{rend}^i\ \ c)^p_{p'}}\ \in^{\mathbf{e}}_{\mathbf{p}}\text{-REND'} \qquad \frac{p \in^{\mathbf{e}}_{\mathbf{p}} h}{p \in^{\mathbf{e}}_{\mathbf{p}} h :: e}\ \in^{\mathbf{e}}_{\mathbf{p}}\text{-::}$$

$$\frac{}{c \in^{\mathbf{e}}_{\mathbf{c}} h :: (\texttt{rend}^i\ \ c)^p_{p'}}\ \in^{\mathbf{e}}_{\mathbf{c}}\text{-REND} \qquad \frac{}{c \in^{\mathbf{e}}_{\mathbf{c}} h :: (*c)_p}\ \in^{\mathbf{e}}_{\mathbf{c}}\text{-CHAN} \qquad \frac{c \in^{\mathbf{e}}_{\mathbf{c}} h}{c \in^{\mathbf{e}}_{\mathbf{c}} h :: e}\ \in^{\mathbf{e}}_{\mathbf{c}}\text{-::}$$

$$\frac{(o)_p \in_{\mathbf{e}} h}{o \in_{\mathbf{o}} h}\ \in_{\mathbf{o}} \qquad \frac{(\texttt{rend}^i\ \ c)^p_{p'} \in_{\mathbf{e}} h}{\texttt{rend}^i\ \ c \in_{\mathbf{o}} h}\ \in_{\mathbf{o}}\text{-REND}$$

Fig. 8: Predicates pertaining to membership in a trace.

$$\frac{e = (o)_p}{\lfloor e \rfloor_{\mathbf{p}} = \{p\}}\ \text{PROJ}_{\mathbf{p}} \qquad \frac{e = (\texttt{rend}^i\ \ c)^p_{p'}}{\lfloor e \rfloor_{\mathbf{p}} = \{p,\ p'\}}\ \text{PROJ-R}_{\mathbf{p}}$$

$$\frac{e = (o)_p}{\lfloor e \rfloor_{\mathbf{o}} = o}\ \text{PROJ}_{\mathbf{o}} \qquad \frac{e = (\texttt{rend}^i\ \ c)^p_{p'}}{\lfloor e \rfloor_{\mathbf{o}} = \texttt{rend}^i\ \ c}\ \text{PROJ-R}_{\mathbf{o}}$$

Fig. 9: Projections.

are considered to be part of the trace as specified by rule INIT_c. At channel creation, the n dummy receive operations denote the fact there is room for n messages to be deposited into the channel via send operations.[5] As we will see later in the definition of well-formed traces, a send is allowed to take place if there is a prior corresponding receive operation on the channel. Finally, Figure 10 defines the predicates $\texttt{closed}\ (c,h)$ for identifying whether a channel has been closed in h and $\texttt{sync}\ (c,h)$ for identifying synchronous channels.

[5] The negative receive operations are considered part of a trace, but they do not actually appear in the trace. Therefore, we must not take then into account in the #SdRv c h function.

$$\begin{array}{rcl} \texttt{isSd}\ c\ (\texttt{sd}^i\ c')_p & := & c = c' \\ \texttt{isSd}\ c\ \_ & := & \texttt{false} \\ \#\texttt{Sd}\ c\ h & := & |\texttt{filter}\ (\texttt{isSd}\ c)\ h| \\ \#\texttt{SdRv}\ c\ h & := & \#\texttt{Rv}\ c\ h - \#\texttt{SdRv}\ c\ h \end{array} \qquad \begin{array}{rcl} \texttt{isRv}\ c\ (\texttt{rv}^i\ c')_p & := & c = c' \\ \texttt{isRv}\ c\ \_ & := & \texttt{false} \\ \#\texttt{Rv}\ c\ h & := & |\texttt{filter}\ (\texttt{isRv}\ c)\ h| \end{array}$$

$$\frac{\vdash h \qquad (\texttt{make}\ c\ n)_p \in_{\mathbf{e}} h \qquad \#\texttt{SdRv}\ c\ h = n}{\texttt{full}\ (c,h)}\ \textsc{Full}_c$$

$$\frac{\vdash h \qquad c \in_{\mathbf{c}} h \qquad \#\texttt{SdRv}\ c\ h = 0}{\texttt{empty}\ (c,h)}\ \textsc{Empty}_c$$

$$\frac{(\texttt{make}\ c\ n)_p \in_{\mathbf{e}} h \qquad 1 \leq i \leq n}{\texttt{rv}^{-i}\ \ c \in_{\mathbf{o}} h}\ \textsc{Init}_c$$

$$\frac{(\texttt{close}^i\ \ c)_p \in_{\mathbf{e}} h}{\texttt{closed}\ (c,h)}\ \textsc{Closed}_c \qquad \frac{(\texttt{make}\ c\ 0)_p \in_{\mathbf{e}} h}{\texttt{sync}\ (c,h)}\ \textsc{Sync}_c$$

Fig. 10: Channel related functions and predicates.

3.2 Well-form

The predicates presented so far come together in the definition of a *well-formed trace* given in Figure 11. In general, a trace is well-formed when it is composed of events involving goroutines that have already been created, as captured by the $p \in_{\mathbf{p}} h$ premise. When it comes to spawning a new goroutine, the identifier p' of the child routine must not be in use already, $p' \notin_{\mathbf{p}} h$. Similarly, for events that create a new channel, the new channel's identifier c' must be fresh, $c' \notin_{\mathbf{c}} h$. Close, send, and receive events must involve a channel c that has already been created; this requirement is captured by premise $c \in_{\mathbf{c}} h$ (unless it is subsumed by another premise).

Sends must respect channel capacity when the channel is open. A send can only take place if there is a prior receive asserting that there is space in the channel for a new message. This requirement is captured by $\texttt{rv}^k\ \ c \in_{\mathbf{o}} h$ in $\vdash$SEND. [6] Similarly, a receive can only take place if a corresponding send has occurred; this condition is captured by premise $\texttt{sd}^k\ \ c \in_{\mathbf{o}} h$ in $\vdash$REC. Note that premises $\texttt{rv}^k\ \ c \in_{\mathbf{o}} h$ of $\vdash$SEND and $\texttt{sd}^k\ \ c \in_{\mathbf{o}} h$ of $\vdash$REC imply that the channel is part of the trace, $c \in_{\mathbf{c}} h$, and that it is asynchronous, $\texttt{sync}\ (c,h)$. Premises $\texttt{sd}^{k+|c|}\ \ c \notin_{\mathbf{o}} h$ and $\texttt{rv}^k\ \ c \notin_{\mathbf{o}} h$ make sure that there are no duplicate sends or receives.

When a channel is closed, receiving is non-blocking: if the channel is empty, the receive operation returns the zero element of the type associated with the channel, $\vdash\textsc{Rec}_{\perp}$. There is no event representing a send on a closed channel. This absence mirror

[6] Readers familiar with the Go memory model may notice that premise $\texttt{rv}^k\ \ c \in_{\mathbf{o}} h$ in $\vdash$SEND captures the adage *"the k^{th} receive on a channel with capacity C happens before the $k+C^{th}$ send from that channel completes"* [7].

$$\frac{}{\vdash [\,]}\ \vdash\textsc{Empty} \qquad \frac{\vdash h \qquad p \in_{\mathbf{p}} h}{\vdash h :: (z\Box)_p}\ \vdash\textsc{Mem}$$

$$\frac{\vdash h \qquad p \in_{\mathbf{p}} h \qquad p' \notin_{\mathbf{p}} h}{\vdash h :: (\mathtt{go}\ \ p')_p}\ \vdash\textsc{Go}$$

$$\frac{\vdash h \qquad p \in_{\mathbf{p}} h \qquad c \notin_{\mathbf{c}} h}{\vdash h :: (\mathtt{make}\ c\ n)_p}\ \vdash\textsc{Make}$$

$$\frac{\begin{array}{c}\vdash h \qquad p \in_{\mathbf{p}} h \qquad c \in_{\mathbf{c}} h \qquad (\mathtt{close}^i\ \ c)_{p'} \notin_{\mathbf{e}} h \\ (\mathtt{sd}^i\ \ c)_{p''} \in_{\mathbf{e}} h \to k = \sqcap\{i \mid (\mathtt{sd}^i\ \ c)_{p''} \in h\} \qquad (\mathtt{sd}^i\ \ c)_{p''} \notin_{\mathbf{e}} h \to k = 0\end{array}}{\vdash h :: (\mathtt{close}^k\ \ c)_p}\ \vdash\textsc{Close}$$

$$\frac{\begin{array}{c}\vdash h \qquad p, p' \in_{\mathbf{p}} h \qquad p \neq p' \qquad c \in_{\mathbf{c}} h \qquad \mathtt{sync}\,(c,h) \qquad \mathtt{rend}^k\ \ c \notin_{\mathbf{o}} h \\ (\mathtt{close}^i\ \ c)_{p'} \in h \to 0 \leq k < i\end{array}}{\vdash h :: (\mathtt{rend}^k\ \ c)^p_{p'}}\ \vdash\textsc{Rend}$$

$$\frac{\begin{array}{c}\vdash h \qquad p \in_{\mathbf{p}} h \qquad \mathtt{rv}^k\ \ c \in_{\mathbf{o}} h \qquad \mathtt{sd}^{k+|c|}\ \ c \notin_{\mathbf{o}} h \\ (\mathtt{close}^i\ \ c)_{p'} \in h \to 0 \leq k + |c| < i\end{array}}{\vdash h :: (\mathtt{sd}^{k+|c|}\ \ c)_p}\ \vdash\textsc{Send}$$

$$\frac{\vdash h \qquad p \in_{\mathbf{p}} h \qquad \mathtt{sd}^k\ \ c \in_{\mathbf{o}} h \qquad \mathtt{rv}^k\ \ c \notin_{\mathbf{o}} h}{\vdash h :: (\mathtt{rv}^k\ \ c)_p}\ \vdash\textsc{Rec}$$

$$\frac{\vdash h \qquad p \in_{\mathbf{p}} h \qquad c \in_{\mathbf{c}} h \qquad \mathtt{closed}\,(c,h)}{\vdash h :: (\mathtt{rv}_\perp\ \ c)_p}\ \vdash\textsc{Rec}_\perp$$

Fig. 11: Well-formed trace.

the operational semantics: attempting to send on a closed channel causes a thread to panic, and the panic is modeled by the absence of an applicable reduction rule.

3.3 Independence between events and partial order on traces

Traces come from observing the execution of a program and are expressed as strings of events. However, in a concurrent system, 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. As advocated by Mazurkiewicz [14], it is useful to combine sequential observations with a dependency relation for studying "the nonsequential behaviour of systems via their sequential observations." In Figure 12, we define the binary relation $\bowtie$ to represent trace events that are *independent*. This relation allows us to reason about derivable traces: when two adjacent events are independent, we are able to swap their position in a trace.

$$\frac{e_1 \bowtie e_2}{e_2 \bowtie e_1}\ \bowtie\text{-SYM} \qquad \frac{p_1 \neq p_2}{(z?)_{p_1} \bowtie (z?)_{p_2}}\ \bowtie\text{-READ} \qquad \frac{p_1 \notin \lfloor e_2 \rfloor_{\mathbf{p}} \qquad \lfloor e_2 \rfloor_{\mathbf{o}} \neq z\square}{(z\square)_{p_1} \bowtie e_2}\ \bowtie\text{-MEM}$$

$$\frac{p_1, p' \notin \lfloor e_2 \rfloor_{\mathbf{p}} \qquad e_2 \neq (\texttt{go}\ \ p_1)_{p_2}}{(\texttt{go}\ \ p')_{p_1} \bowtie e_2}\ \bowtie\text{-GO}$$

$$\frac{p_1 \notin \lfloor e_2 \rfloor_{\mathbf{p}} \qquad \lfloor e_2 \rfloor_{\mathbf{o}} \notin \{\texttt{go}\ \ p_1,\ \texttt{make}\, c\, m,\ *c,\ \texttt{rend}^i\ \ c\}}{(\texttt{make}\, c\, n)_{p_1} \bowtie e_2}\ \bowtie\text{-MAKE}$$

$$\frac{p_1 \notin \lfloor e_2 \rfloor_{\mathbf{p}} \qquad \lfloor e_2 \rfloor_{\mathbf{o}} \notin \{\texttt{go}\ \ p_1,\ \texttt{make}\, c\, m,\ *c,\ \texttt{rend}^i\ \ c\}}{(*c)_{p_1} \bowtie e_2}\ \bowtie\text{-CHAN}$$

$$\frac{p_1 \neq p_2}{(\texttt{sd}^i\ \ c)_{p_1} \bowtie (\texttt{sd}^j\ \ c)_{p_2}}\ \bowtie\text{-SEND} \qquad \frac{p_1 \neq p_2}{(\texttt{rv}^i\ \ c)_{p_1} \bowtie (\texttt{rv}^j\ \ c)_{p_2}}\ \bowtie\text{-REC}$$

$$\frac{p_1 \neq p_2 \qquad i \neq j \qquad i \neq j + |c|}{(\texttt{sd}^i\ \ c)_{p_1} \bowtie (\texttt{rv}^j\ \ c)_{p_2}}\ \bowtie\text{-SDRV} \qquad \frac{\begin{array}{cc} p_1 \neq p_3 & p_1 \neq p_4 \\ p_2 \neq p_3 & p_2 \neq p_4 \end{array}}{(\texttt{rend}^i\ \ c)^{p_1}_{p_2} \bowtie (\texttt{rend}^j\ \ c')^{p_3}_{p_4}}\ \bowtie\text{-REND}$$

$$\frac{p_1 \notin \lfloor e_2 \rfloor_{\mathbf{p}} \qquad \lfloor e_2 \rfloor_{\mathbf{o}} \notin \{\texttt{go}\ \ p_1,\ \texttt{make}\, c\, m,\ \texttt{close}^i\ \ c\}}{(\texttt{rv}_\perp\ \ c)_{p_1} \bowtie e_2}\ \bowtie\text{-REC}_\perp$$

$$\frac{p_1 \neq p_2}{(\texttt{close}^i\ \ c)_{p_1} \bowtie (\texttt{sd}^j\ \ c)_{p_2}}\ \bowtie\text{-CLOSESD}$$

$$\frac{p_1 \neq p_2}{(\texttt{close}^i\ \ c)_{p_1} \bowtie (\texttt{rv}^j\ \ c)_{p_2}}\ \bowtie\text{-CLOSERV}$$

$$\frac{p_1 \neq p_2 \qquad p_1 \neq p_3}{(\texttt{close}^i\ \ c)_{p_1} \bowtie (\texttt{rend}^j\ \ c)^{p_2}_{p_3}}\ \bowtie\text{-CLOSER}$$

Fig. 12: Independent trace events.

Note that operations from the same goroutine are trivially not independent, as their order is pre-imposed by the program: for events $(o_1)_{p_1}$ and e_2 to be independent, we must have $p_1 \notin \lfloor e_2 \rfloor_{\mathbf{p}}$. Rule $\bowtie$-READ says that reads from different goroutines to the same variable are independent. Memory operations from different goroutines are independent if they address different variables ($\bowtie$-MEM).

The rule $\bowtie$-GO details when the goroutine-creation event $(\texttt{go}\ \ p_3)_{p_1}$ is independent from another operation o from goroutine p_2: the go operation must not be creating p_2, as captured by $p_2 \neq p_3$ in the rule's premise. Similarly, p_2 must not be creating the thread executing the go operation. This last condition, captured by premise $o \neq \texttt{go}\ \ p_1$, is also in

the premise of other rules such as $\bowtie$-MAKE, $\bowtie$-CHAN, and $\bowtie$-REC$_\perp$. Rule $\bowtie$-MAKE specifies that the creation of a channel c is independent from events e_2 that do not involve c. Similar for $\bowtie$-CHAN. In general, sends are independent from other sends and receives from other receives[7] — see rules $\bowtie$-SEND and $\bowtie$-REC.

The dependence between sends and receives follows the Golang memory model [7] to the letter. The model specifies that, given a channel c with capacity $k = |c|$:

A send on c happens-before the corresponding receive from c completes. (3)

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

Therefore, in order for a send event $(\texttt{sd}^i\ c)_{p_1}$ to be independent from a receive event $(\texttt{rv}^j\ c)_{p_2}$, they most not be in a happens-before relation with each other. In the independence rule $\bowtie$-SDRV, premise $i \neq j$ captures condition (3) while $i \neq j + |c|$ captures condition (4). When the channel is synchronous, condition (4) collapses into condition (3) given that $|c| = 0$. Note that rule $\bowtie$-REND does not explicitly require $i \neq j$ since it is implied from the fact that the processes involved in the two rendezvous are required to be different.

Receiving the end-of-transmission marker $\perp$ when receiving from a channel implies that the channel is closed. Therefore, $(\texttt{rv}_\perp\ c)_{p_1}$ and $(\texttt{close}^i\ c)_{p_2}$ are not independent events. Aside from that, a close event on c is independent from sends ($\bowtie$-CLOSESD), receives ($\bowtie$-CLOSERV), and rendezvous ($\bowtie$-CLOSER) on c.

Note that the independence relation is irreflexive. If the relation were reflexive, we would erroneously conclude, for example, that the writes to z in the code snippet $z := 1;\ z := 2$ are independent. The relation $\bowtie$ is also not transitive; for example, $(z_1?)_{p_1} \bowtie (z_2?)_{p_2}$, $(z_2?)_{p_2} \bowtie (z_1!)_{p_3}$, but $(z_1?)_{p_1} \not\bowtie (z_1!)_{p_3}$.

$$\frac{\vdash h :: e_1 :: e_2 \qquad e_1 \bowtie e_2}{h :: e_1 :: e_2 \sqsubseteq h :: e_2 :: e_1}\ \sqsubseteq\text{-CONC} \qquad \frac{\vdash h_1 {}^{++} h \qquad h_1 \sqsubseteq h_2}{h_1 {}^{++} h \sqsubseteq h_2 {}^{++} h}\ \sqsubseteq\text{-PREFIX}$$

$$\frac{h_1 \sqsubseteq h_2 \qquad h_2 \sqsubseteq h_3}{h_1 \sqsubseteq h_3}\ \sqsubseteq\text{-TRANS}$$

Fig. 13: Partial order on traces.

[7] Here, the motivation for attaching an index to channel operations becomes evident. We want, for example, to record the send and receive operations in a trace yet have the recording of such operations be "swappable" in a trace. It is not possible to swap two concurrent sends onto (or receives from) the same channel: different runs are obtained by altering the order in which the channel operations take effect. Yet, once the sends (or receives) have taken effect with respect to the channel's queue, the order in which the sends (or receives) are presented in a trace no longer matters. For example, the i^{th} send operation cannot be swapped with the $(i+1)^{th}$ send operation on the same channel. However, having recorded that $\texttt{sd}\ c$ is the i^{th} send, as in $\texttt{sd}^i\ c$, then we can safely swap the event with $\texttt{sd}^{i+1}\ c$ in a given trace.

In Figure 13 we define the *derivable from* relation $\sqsubseteq$ on traces. Given the set of well-formed finite traces $H = \{h | h = e^* \text{ and } \vdash h\}$, H is a pre-ordered set with $\sqsubseteq$ a pre-order. Rule R-CONC allow us to apply the independence relation in the context of traces. Rule R-PREFIX generalizes $\sqsubseteq$ to prefixes of a potentially longer traces.

3.4 Identifying data-races in traces

A trace h contains a data race if h' contains a *manifest data race* and h' can be derived from h, meaning, $h \sqsubseteq h'$.

Definition 1 (Data race). *A well-formed trace h contains a* manifest data race *if either*

$$(z!)_p\ (z!)_{p'} \qquad \textit{(manifest write-after-write)}$$
$$(z!)_p\ (z?)_{p'} \qquad \textit{(manifest read-after-write)}$$
$$(z?)_p\ (z!)_{p'} \qquad \textit{(manifest write-after-read)}$$

are a sub-sequence of h.

Note that the concept of manifest data race and the derive-from relation $\sqsubseteq$ give us an idealized race detector. A race is detected in an execution resulting in trace h if $h \sqsubseteq h'$ and h' contains a manifest data-race. The idealized detector may not be practical: by consuming traces in their entirety, such a detector requires global information and cannot be easily partitioned. Also, by operating on a trace without necessarily maintaining internal state, a naive detector checking for races in $h :: e$ is likely to duplicate work that went into checking h.

The practicality of this idealized detector is not a problem, however. As we will see in Section 4, the detector operating on traces will serve a specification against which we establish the correctness of the race detector introduced in Section 2.4.

3.5 Example

Let us look at an example that illustrates how the independence and derivable-from relations of Figure 12 and 13 can be used to reason about equivalences of traces. Consider the scenario of Figure 14 in which three threads communicate over a channel of capacity $|c| \geq 2$. From p_1's code we see that whichever goroutine receives from c first

$$p_1\langle x := v_1;\ c \leftarrow m_1;\ y := v_2;\ c \leftarrow m_2\rangle$$
$$p_2\langle \leftarrow c;\ \texttt{load}\, x\rangle$$
$$p_3\langle \leftarrow c;\ \texttt{load}\, y\rangle$$

Fig. 14: Message passing, twice.

can safely read x but not y: reading y would yield a data race. On the other hand, the goroutine that receives from c last can read x and y.

Trace 5 below is associated with a well-synchronized run of the program. We have $(x!)_{p_1} \rightarrow_{\mathsf{hb}} (x?)_{p_2}$ and $(y!)_{p_1} \rightarrow_{\mathsf{hb}} (y?)_{p_3}$ where $\rightarrow_{\mathsf{hb}}$ is the happens-before relation.

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (y!)_{p_1}\ (\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_2}\ (\mathtt{rv}^1\ c)_{p_3}\ (x?)_{p_2}\ (y?)_{p_3} \tag{5}$$

Figure 12 and 13 define when events of a trace can be swapped with the resulting trace being derivable from the original one. For example, according to $\bowtie$-MEM, the last two events in Trace 5 are independent. We can use that fact along with $\sqsubseteq$-CONC in order to derive an equivalent trace in which $(x?)_{p_2}$ and $(y?)_{p_3}$ are swapped.

Consider now Trace 6 below; it looks like Trace 5 except that the order of the receives is reversed.

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (y!)_{p_1}\ (\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}\ (\mathtt{rv}^1\ c)_{p_2}\ (x?)_{p_2}\ (y?)_{p_3} \tag{6}$$

Despite their similarities, Trace 5 and 6 are not derivable-from one another. Changing the order of the receive operations between threads p_2 and p_3 fundamentally alters the execution of the program: it changes which goroutine receives what message, thus altering the program's synchronization. As a mater of fact, while Trace 5 is well synchronized, Trace 6 contains a read-after-write (RaW) conflict between $(y!)_{p_1}$ and $(y?)_{p_3}$. As a sanity check, note that it is not possible to derive Trace 6 by applying the rules of Figure 12 and 13 starting from Trace 5.

In order to make the race condition in Trace 6 more evident, we can derive an equivalent trace that contains a *manifest data race*. The following trace with a manifest data race is obtained by swapping events of Trace 6 according to the rules of Figure 12 and 13:

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (y!)_{p_1}\ (\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}\ (\mathtt{rv}^1\ c)_{p_2}\ \boxed{(x?)_{p_2}\ (y?)_{p_3}} \qquad (\bowtie\text{-MEM})$$

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (y!)_{p_1}\ (\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}\ \boxed{(\mathtt{rv}^1\ c)_{p_2}\ (y?)_{p_3}}\ (x?)_{p_2} \qquad (\bowtie\text{-CHAN})$$

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (y!)_{p_1}\ \boxed{(\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}}\ (y?)_{p_3}\ (\mathtt{rv}^1\ c)_{p_2}\ (x?)_{p_2} \qquad (\bowtie\text{-SDRV})$$

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ \boxed{(y!)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}}\ (\mathtt{sd}^1\ c)_{p_1}\ (y?)_{p_3}\ (\mathtt{rv}^1\ c)_{p_2}\ (x?)_{p_2} \qquad (\bowtie\text{-CHAN}, \bowtie\text{-SYM})$$

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}\ (y!)_{p_1}\ \boxed{(\mathtt{sd}^1\ c)_{p_1}\ (y?)_{p_3}}\ (\mathtt{rv}^1\ c)_{p_2}\ (x?)_{p_2} \qquad (\bowtie\text{-CHAN})$$

$$(x!)_{p_1}\ (\mathtt{sd}^0\ c)_{p_1}\ (\mathtt{rv}^0\ c)_{p_3}\ (y!)_{p_1}\ (y?)_{p_3}\ (\mathtt{sd}^1\ c)_{p_1}\ (\mathtt{rv}^1\ c)_{p_2}\ (x?)_{p_2}$$

4 Formal properties of the data-race detector

We presented in Section 2 the operational semantics of a race detector for a language inspired by Go. We then presented a trace grammar and defined the concept of manifest data-race in Section 3. Using the independence relation on trace events $\bowtie$ and the derive-from relation on traces $\sqsubseteq$, we showed how to obtain a trace containing a manifest data-race from a trace obtained from an ill-synchronized run. Here we relate the operational semantics to the trace grammar so that, in Sections 4.2 and 4.3, we can establish

soundness and completeness of the race detector. When reasoning about traces via the independence relation on events and the derive-from relation on traces, one needs to argue for the correctness of the relations with respect to a labeled transition system. This discussion is relegated to Appendix C.

4.1 Traces from the race-detector

A trace can be constructed straightforwardly from the reduction rules of the operational semantics. Rule R-GO yields the trace event $(\texttt{go}\ p')_p$ where p is the parent and p' is the child-process. R-WRITE and R-READ yields trace events $(z?)_p$ and $(z!)_p$ respectively. Channel creation R-MAKE yields $(\texttt{make}\, c\, v)_p$ with v the channel buffer size. R-SEND and send acknowledgments R-SACK yield $(\underline{\texttt{sd}\, c})_p$ and $(\texttt{sd}\ c)_p$ respectively. Similar for R-REC and $(\underline{\texttt{rv}\, c})_p$ as well as R-RACK and $(\texttt{rv}\ c)_p$. R-REC$_\perp$ yields $(\texttt{rv}_\perp\ c)_p$, R-REND does $(\texttt{rend}\ c)^p_{p'}$, and R-CLOSE $(\texttt{close}\ c)_p$. Except for R-SEL-DEF, which generates no events, the rules for select also produce send, receive, and synchronous send/receive trace events. Local reduction rules generate no trace events. The following three rules yield events besides placing the program configuration in an irreducible error state: R-WRITE-E$_{WaR}$ and R-WRITE-E$_{WaW}$ yield trace event $(z!)_p$, R-READ-E$_{RaW}$ yields $(z?)_p$.

The trace h obtained from the direct translation of the operational semantics' reduction rules is post processed by the algorithm of Figure 15. This post processing outputs a trace acc conforming to the trace grammar of Section 3. During the processing, the al-

```
process h = processRec (λx. 0) (λx. 0) (λx. 0) acc h

processRec S R I acc  [ ] = acc
          | S R I acc  (h::(sd c)_p)
               = process (inc S c) R (set I p (S c)) acc  h
          | S R I acc  (h::(rv c)_p)
               = process S (inc R c) (set I p (R c)) acc  h
          | S R I acc  (h::(sd  c)_p)
               = process S R I (acc::(sd^(I c)  c)_p)  h
          | S R I acc  (h::(rv  c)_p)
               = process S R I (acc::(rv^(I c)  c)_p)  h
          | S R I acc  (h::(rend  c)^p_p')
               = process (inc S c) R I (acc::(rend^(S c)  c)^p_p')  h
          | S R I acc  (h::(close  c)_p)
               = process S R I (acc::(close^(S c)  c)_p)  h
          | S R I acc  (h::e)  =  process S R I (acc::e)  h

inc X c = λu. if u=c then (X c)+1 else (X c)

set I p v = λu. if u=p then v else (I p)
```

Fig. 15: From operational semantics' trace to trace grammar of Section 3.

gorithm filters out the initiation of send and receive events but keeps their acknowledgments. The algorithm also attaches an index to the send and receive acknowledgments, to rendezvous events, and to close events.

The algorithm of Figure 15 works as follows. The S and R are functions from channel identifier to a natural number. The S keeps track of the number of send events on each channel, while the R keeps track of the number of receives. When a send event $(\underline{\texttt{sd}\ c})_p$ is encountered in trace h, the send-counter for channel c is incremented via $(\text{inc}\ \text{S}\ c)$. Similar for receive and $(\text{inc}\ \text{R}\ c)$. The I is a function from process identifier to natural number; it records the value for the send (or receive) counter on c at the time a goroutine initiates a send (or receive) event on c. Send events $(\underline{\texttt{sd}\ c})_p$ in h are filtered out, meaning that they do not get carried over to the resulting acc trace. Instead, the acknowledgment of the send is recorded in acc, along with the send-counter at the time the send event was initiated: $\text{acc}::(\texttt{sd}^{(\text{I}\ c)}\ c)_p)$. The same ideas apply to receive events. Rendezvous and close events are similarly labeled with an event index. Other events in h are transferred unaltered to the output trace acc.

From here on, when we refer to h, we are talking about the output of process having been given as input a trace obtained from the data-race detector. Therefore, h is a trace in the grammar of Section 3.

4.2 Soundness of the race detector

In this section we show that if a run $S_0 \xrightarrow{h} S$ is flagged by the data-race detector, then either:

- h has a manifest data-race, or
- there exists a trace h_{dr} containing a manifest data-race and $h \sqsubseteq h_{dr}$.

We will focus on WaW races, noting that proofs of the soundness of WaR and RaW races follow similarly. Here we prove correctness with respect to a slightly simpler version of the race detector, namely, one without the garbage collection term $\backslash(\_, z)$ in the R-WRITE rule. This modification makes happens-before set E_{hb} of each thread monotonically increasing, which, in turn, makes the proof slightly simpler.

The following notation is used in the proofs: Let $\texttt{idx}\ (e,h) = i$ be the index of event e in trace h. If $h[i] = (o)_p$, then let $E_{hb}@h[i]$ be the happens-before set associated with thread p immediately after the execution of operation o. If $h[i] = (\texttt{rend}^k\ c)^p_{p'}$ then let $E_{hb}@h[i]$ be the union of the happens-before sets associated with thread p and p'. We use the notation $E_{hb}@e$ when it is clear from the context that $e = h[i]$ for a given h and i.

The race detector flags a WaW race via the reduction rule R-WRITE-E$_{WaW}$, which generates an event $(z!)_p$. We thus consider runs of the form $S_0 \xrightarrow{h::(z!)_p} S$.

Lemma 2. *(Existence of a prior write event.) Let $S_0 \xrightarrow{h::(z!)_p} S$ be a run flagged as containing a WaW race. The prefix $S_0 \xrightarrow{h}$ must contain a write to z producing a label m such that (m,z) is not in the happens-before set of p.*

Proof. If $m(\!|z{:=}v|\!)\mathcal{L}^R$ is the record associated with z in a configuration, in order for R-WRITE-E_{WaW} to be enabled given a write to z, the pair (m,z) must not be in the happens-before set of the writing thread.

By well-formedness of the initial configuration, we have that for every shared variable z, there exists a unique record $m_{0,z}(\!|z{:=}v|\!)\emptyset$ in S_0 such that $(m_{0,z},z) \in E_{hb}$ of the main thread p_0. Given that every thread is a descendant of p_0 and that parents pass their happens-before set to their children, we have that $(m_{0,z},z)$ is in the happens-before set of every thread.[8] Therefore, in order for R-WRITE-E_{WaW} to be enabled given a write to z, the label m in the record $m(\!|z{:=}v|\!)\mathcal{L}^R$ must not be the initial label $m_{0,z}$.

Since R-WRITE is the only rule that alters the label associated with a record $m(\!|z{:=}v|\!)\mathcal{L}^R$, an event corresponding to a write to z reduced via R-WRITE must exist in h in order for the race detector to flag a WaW race in $S_0 \xrightarrow{h::(z!)_p} S$. □

Corollary 3. *(Prior write by different thread.) Note that in Lemma 2, if p' is the thread performing the write to z producing label m with (m,z) not in the happens-before of p, then it must be that $p' \neq p$.*

Proof. The write occurring in $S_0 \xrightarrow{h}$ places (m,z) in the E_{hb} of the writing thread p'. The write triggering the WaW race requires (m,z) not to be in the E_{hb} of p. By monotonicity of the happens-before set we have that $p \neq p'$. □

Lemma 4. *(Most recent write). Let $S_0 \xrightarrow{h::(z!)_p} S$ be a run flagged as containing a WaW race. Let $\bar{w}$ be a write to z in h such that, for all other writes w to z in h, $\texttt{idx}\,(w,h) <\texttt{idx}\,(\bar{w},h)$. In other words, $\bar{w}$ is the most recently occurring write-event involving z in h. The label $\bar{m}$ associated with $\bar{w}$ is not in the happens-before of p.*

Proof. Assume the label $\bar{m}$ associated with $\bar{w}$ is in the happens-before set of p. By Lemma 2 we have that there exist a write w' by thread $p' \neq p$ producing label m' such that (m',z) is not in the happens before of p.

By definition of $\bar{w}$, we have that $\texttt{idx}\,(w',h) <\texttt{idx}\,(\bar{w},h)$. Since the race detector did not flag a WaW race when executing $\bar{w}$, it must be that (m',z) was in the happens-before set of $\bar{p}$ at the time $\bar{w}$ took place.

Give that happens-before sets are inherited (during thread creation) or communicated (via message passing) in their entirety, in order for the happens-before set of p to contain $(\bar{m},z)$ it must be that it also contains (m',z), which is a contradiction. □

Lemma 5. *(No reads in between). Let $S_0 \xrightarrow{h::(z!)_p} S$ be a run flagged as containing a WaW race. Let $\bar{w}$ be a write to z in h such that, for all other writes w to z in h, $\texttt{idx}\,(w,h) <\texttt{idx}\,(\bar{w},h)$. There cannot be an event $\hat{e} = (z?)_{\hat{p}}$ such that $\texttt{idx}\,(\bar{w},h) <\texttt{idx}\,(\hat{e},h)$.*

Proof. If there were a read from z in h and $(\bar{m},z)$ were not in the happens-before of $\hat{p}$, then R-READ-E_{RaW} would have fired and execution would have halted at $\hat{e}$. If there were a read from z in h and $(\bar{m},z)$ were in the happens-before of $\hat{p}$, then R-WRITE-E_{WaW} would not be enabled at e and a WaR race would have been flagged instead. □

[8] Note that we are making use of the monotonicity of the happens-before set. In the version of the race-detector with garbage collection, because happens-before sets are not monotonically increasing, we would have to make a more elaborate argument.

Because of Lemmas 4 and 5, we are able to focus on showing a manifest data-race between two crucial memory events, irrespective of other memory operations. To summarize, at this point we have established that if $S_0 \xrightarrow{h::(z!)_p} S$ is a run flagged as containing a WaW race, there exists a write event $\bar{w}$ to variable z in h such that:

- $\bar{w}$ splits h into $h = h_0\ \bar{w}\ \hat{h}$,
- there are no reads or writes to z in $\hat{h}$,
- $\bar{w}$ and $(z!)_p$ constitute a WaW race.

Next, we show that we can derive a manifest data-race from the subsequence $\bar{w}\ \hat{h}\ w$ where $\bar{w} = (z!)_{\bar{p}}$ and $w = (z!)_p$. Informally, we will use the derivable-from relation $\sqsubseteq$ to "move" events from $\hat{h}$ to the left of $\bar{w}$ or the right of w, thus *hollowing* $\hat{h}$ so that $\bar{w}\ w$ will be a subsequence of the resulting trace. Since $\bar{w}\ w = (z!)_{\bar{p}}(z!)_p$ with $\bar{p} \neq p$, this subsequence constitutes a manifest write-after-write data race.

Lemmas 6 and 7 are stepping stones for the more fundamental Lemma 8. When hollowing $\hat{h}$ in $\bar{w}\ \hat{h}\ w$, some events may need to be moved to the left of $\bar{w}$ and some to the right of w. Lemma 8 handles the case in which $\bar{w} \ldots e_r\ e_l \ldots w$ with e_r needing to be moved to the right of w and e_l needing to be moved to the left of $\bar{w}$. In this case, we show that e_r and e_l can be swapped, in other words, that $\bar{w} \ldots e_l\ e_r \ldots w$ can be derived. The ability to swap e_r and e_l hinges on their independence, which is the topic of Lemma 8.

Lemma 6. *(Different threads). Let h be a well-formed trace with $e_1\ e_2$ as sub-trace. If $E_{hb}@e_1 \not\subseteq E_{hb}@e_2$, then $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$.*

Proof. Here we prove the contrapositive: Given that the happens-before set is monotonically increasing, in each of the following four cases, if $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} \neq \emptyset$, then $E_{hb}@e_1 \subseteq E_{hb}@e_2$.

Case: $e_1 = (o_1)_{p_1}$ and $e_2 = (o_2)_{p_2}$ with $p_1 = p_2$

Case: $e_1 = (o_1)_{p_1}$ and $e_2 = (\texttt{rend}^i\ \ c)^{p_2}_{p_2'}$ with $p_1 = p_2$ or $p_1 = p_2'$

Case: $e_1 = (\texttt{rend}^i\ \ c)^{p_1}_{p_1'}$ and $e_2 = (o_2)_{p_2}$ with $p_1 = p_2$ or $p_1' = p_2$

Case: $e_1 = (\texttt{rend}^i\ \ c_1)^{p_1}_{p_1'}$ and $e_2 = (\texttt{rend}^j\ \ c_2)^{p_2}_{p_2'}$ with $p_1 = p_2$, $p_1 = p_2'$, $p_1' = p_2$, or $p_1' = p_2'$

□

Lemma 7. *(Channel handle propagation). Given a trace h obtained from the operational semantics, it is not possible to derive an h' from h where $(\texttt{make}\ c\ n)_p\ \ (*c)_{p'}$ is a subsequence with $p \neq p'$.*

Proof. Channel creation takes place via R-MAKE, which puts a νc-binder around the thread p that created the channel and the channel forward and backward queues. In order for thread p' to get a handle to c, p' must somehow be inside the binder. In all the ways p' exist inside the binder, at least one event e sits between the making and the using of the channel with $(\texttt{make}\ c\ n)_p \not\bowtie e$ and $e \not\bowtie (*c)_{p'}$.

Case: p creates c then spawns p'
If p creates c and then spawns p' via `go`, then p' is inside the νc-binder and can have a handle on c. An example program would be:

```
let c = make(chan int, n) in go { \* handle on c *\}
```

and reduces as follows:

$$
\begin{array}{lr}
p\langle \texttt{let}\ r = \texttt{make}\ (\texttt{chan}\ T, v)\ \texttt{in}\ \texttt{go}\ t';\ t\rangle & \text{R-MAKE} \\
\quad \rightarrow\ \nu c\ (p\langle \texttt{let}\ r = c\ \texttt{in}\ \texttt{go}\ t';\ t\rangle \parallel c_f[] \parallel c_b[q]) & \text{R-LET} \\
\quad \rightarrow\ \nu c\ (p\langle \texttt{go}\ t'[c/r];\ t[c/r]\rangle \parallel c_f[] \parallel c_b[q]) & \text{R-GO} \\
\quad \rightarrow\ \nu c\ (\nu p'\ (p\langle t[c/r]\rangle \parallel p'\langle t'[c/r]\rangle \parallel c_f[] \parallel c_b[q])) &
\end{array}
$$

The corresponding trace h would have the event $(\texttt{go}\ \ p')_p$ in between the $(\texttt{make}\ c\ n)_p$ and any event $(*c)_{p'}$ on c from a thread $p' \neq p$. Since $(\texttt{make}\ c\ n)_p \not\bowtie (\texttt{go}\ \ p')_p$ and $(\texttt{go}\ \ p')_p \not\bowtie (o)_{p'}$, the subsequence $(\texttt{make}\ c\ n)_p\ \ (*c)_{p'}$ cannot exist in a trace derived from h.

Case: p writes c onto a shared variable read by p'
In this case, we would have the following subsequence where $(z!)_p$ writes c onto the shared variable z and $(z?)_{p'}$ reads c from z.

$$(\texttt{make}\ c\ n)_p\ \ (z!)_p\ \ (z?\ )_{p'}\ (*c)_{p'}$$

In terms of the configuration, the structural congruence rule $P_1 \parallel \nu n\ P_2 \equiv \nu n\ (P_1 \parallel P_2)$ if $n \notin fn(P_1)$ expands the scope of the ν-binder from around p, c_f, and c_b to also surround p' before p' reads c from z. Note that $(z!)_p \not\bowtie (z?)_{p'}$, $(\texttt{make}\ c\ n)_p \not\bowtie (z!)_p$ because of program order, and $(z?)_{p'} \not\bowtie (*c)_{p'}$ also because of program order.

Without synchronization, it is possible for p' to read the old value of z as opposed to the value of z which contains c. Without synchronization, this scenario contains a data race. If synchronization is added, for example, by p writing to z and then sending on a channel and by p' reading the message before accessing z, then the race is eliminated. The impossibility of deriving $(\texttt{make}\ c\ n)_p\ \ (*c)_{p'}$ now comes not only from the dependence between the write-to and the read-from z but also from the channel communication between the two threads.

Case: p sends c to p' on a channel c'
The `rend` event below captures a synchronous message from p to p' in which the handle to c is communicated over a channel $c' \neq c$. A similar argument can be made about communication over asynchronous channels.

$$(\texttt{make}\ c\ n)_p\ \ (\texttt{rend}\ \ c')^p_{p'}\ \ (*c)_{p'}$$

Since $(\texttt{make}\ c\ n)_p \not\bowtie (\texttt{rend}\ \ c')^p_{p'}$ and $(\texttt{rend}\ \ c')^p_{p'} \not\bowtie (*c)_{p'}$, the subsequence $(\texttt{make}\ c\ n)_p\ \ (*c)_{p'}$ is not derivable.[9] □

[9] Similar to the case in which c is written to z, the νc-binder is made to wrap around p' by the use of the congruence rule $P_1 \parallel \nu n\ P_2 \equiv \nu n\ (P_1 \parallel P_2)$ if $n \notin fn(P_1)$.

Lemma 8. *(Independence). Consider the postfix $\bar{w}\ \hat{h}\ w$ of a run flagged as containing a write-after-write race on z where $w = (z!)_p$, $\bar{w} = (z!)_{\bar{p}}$ with no memory events involving z in $\hat{h}$. Pick two adjacent event $e_1\ e_2$ from $\hat{h}$; in other words, $\hat{h} = \hat{h}_l\ e_1\ e_2\ \hat{h}_r$. If $E_{hb}@\bar{w} \subseteq e_1$ and $E_{hb}@\bar{w} \not\subseteq e_2$ then $e_1 \bowtie e_2$.*

Proof. Since $E_{hb}@\bar{w} \subseteq e_1$ and $E_{hb}@\bar{w} \not\subseteq e_2$ we have that $E_{hb}@e_1 \not\subseteq E_{hb}@e_2$. Thus, by Lemma 6 we have that $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$.

From here, the proof proceeds via case analysis on e_1 and e_2.

Case: $e_1 = (\texttt{make}\ c\ n)_{p_1}$
First consider the case in which the operation performed at e_1 does not involve c. In these cases, we have that $e_1 \bowtie e_2$ via $\bowtie$-MAKE, $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$, and well-form. Via Lemma 7, we have that if e_2 is an operation involving a channel c', we have that $c' \neq c$.

Case: $e_1 = (\texttt{sd}^i\ \ c)_{p_1}$
We split on the $o = \lfloor e_2 \rfloor_{\mathbf{o}}$ operation, starting with operations involving c. The interesting case is $o = \texttt{rv}^j\ \ c$, where we have three options: (1) If $i \neq j$ and $i \neq j + |c|$, then $e_1 \bowtie e_2$ via $\bowtie$-SDRV. (2) If $i = j$, according to the operational semantics, the happens-before set of the process p_2 executing e_2 is augmented with the happens-before set of p_1 when p_1 performed e_1. Therefore, $e_1 = (\texttt{sd}^i\ \ c)_{p_1}$ followed by $e_2 = (\texttt{rv}^j\ \ c)_{p_2}$ with $i = j$ violates the assumption $E_{hb}@e_1 \not\subseteq E_{hb}@e_2$. (3) The case in which $i = j + |c|$ is not well-formed according to $\vdash$SEND: in order for there to be a send with index $j + |c|$ in a trace h, there must already be a receive with index j in h. Other cases are covered as follows: If $o = \texttt{sd}^j\ \ c$ then $e_1 \bowtie e_2$ by $\bowtie$-SEND along with $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$. Similar for $o = \texttt{close}^j\ \ c$ and $\bowtie$-CLOSESD as well as $o = \texttt{rv}_\perp\ \ c$ and $\bowtie$-REC$_\perp$. Note that $o = \texttt{make}\ c\ n$ or $o = \texttt{rend}^i\ \ c$ would not yield a well-formed trace. Finally, operations involving $c' \neq c$ as well as non-channel operations are covered by $\bowtie$-CHAN along with $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$ and the well-form condition.

Case: $e_1 = (\texttt{rv}^i\ \ c)_{p_1}$
Similar to $e_1 = (\texttt{sd}^i\ \ c)_{p_1}$.

Case: $e_1 = (\texttt{rend}^i\ \ c)^{p_1}_{p_1'}$
If $e_2 = (\texttt{rend}^j\ \ c')^{p_2}_{p_2'}$ then, given that $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$, we have $e_1 \bowtie e_2$ via $\bowtie$-REND regardless of whether $c = c'$. In the case of $e_2 = (o)_{p_2}$, we split on the operation o. Let us first consider operations involving c. If $o = \texttt{close}^j\ \ c$ then $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$ imply $e_1 \bowtie e_2$ via $\bowtie$-CLOSER. Similar for $o = \texttt{rv}_\perp\ \ c$ and $\bowtie$-REC$_\perp$. Note that the cases in which $o \in \{\texttt{make}\ c\ n,\ \ \texttt{sd}^j\ \ c,\ \ \texttt{rv}^j\ \ c\}$ are not well formed. Now consider operations involving $c' \neq c$. If $o = *c'$ then $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$ imply $e_1 \bowtie e_2$ via $\bowtie$-CHAN. Similar for $o = \texttt{make}\ c'\ n$ and $\bowtie$-MAKE. Finally, if o is a memory operation then $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$ imply $e_1 \bowtie e_2$ via $\bowtie$-MEM. Similarly for $o = \texttt{go}\ \ p'$ and $\bowtie$-GO.

Case: $e_1 = (\texttt{rv}_\perp\ \ c)_{p_1}$
We have $e_1 \bowtie e_2$ via $\bowtie$-REC$_\perp$, $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$, and well-form. We note that well-form prevents the $\texttt{rv}_\perp\ \ c$ in e_1 to be followed by $\texttt{close}^i\ \ c$.

Case: $e_1 = (\texttt{go}\ \ p')_{p_1}$
We have $e_1 \bowtie e_2$ via $\bowtie$-GO, $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$, and the fact that, in a well-formed trace, the operation performed at e_2 cannot be $\texttt{go}\ \ p_1$.

Case: $e_1 = (\texttt{close}^i\ c)_{p_1}$
We split on the operation $o = \lfloor e_2 \rfloor_{\mathbf{o}}$ starting with operations involving c. The interesting case to consider is $o = \texttt{rv}_\perp\ c$. According to the operational semantics, the happens-before set of p_1 when closing c is transferred to the thread receiving the end-of-transmission marker from c. Therefore, $e_1 = (\texttt{close}^i\ c)_{p_1}$ followed by $e_2 = (\texttt{rv}_\perp\ c)_{p_2}$ violates the assumption $E_{hb}@e_1 \not\subseteq E_{hb}@e_2$. Other cases are covered as follows: If $o = \texttt{sd}^j\ c$ or $o = \texttt{rv}^j\ c$, then $e_1 \bowtie e_2$ by $\bowtie$-CLOSESD or $\bowtie$-CLOSERV along with $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$. Similar for $o = \texttt{rend}^j\ c$ and $\bowtie$-CLOSER. Note that $o = \texttt{close}^j\ c$ and $o = \texttt{make}\ c\ n$ would imply that the trace is not well-formed. For the remaining cases, $e_1 \bowtie e_2$ via $\bowtie$-CHAN along with $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$ and well-form.

Case: $e_1 = (x\Box)_{p_1}$
If $o = (x\Box)_{p_2}$ then both e_1 and e_2 must be read operations, otherwise they would constitute a manifest data-race. In that case, events are independent via $\bowtie$-READ along with $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$. For all other cases, $e_1 \bowtie e_2$ via $\bowtie$-MEM and $\lfloor e_1 \rfloor_{\mathbf{p}} \cap \lfloor e_2 \rfloor_{\mathbf{p}} = \emptyset$.

□

When are now ready to show that it is possible to hollow $\hat{h}$ in $\bar{w}\ \hat{h}\ w$ in order to derive a trace containing $\bar{w}\ w$ as the manifest data-race.

Lemma 9. *(Hollowing). Consider the postfix $\bar{w}\ \hat{h}\ w$ of a run flagged as containing a write-after-write race on z $\bar{w} = (z!)_{\bar{p}}$, where $w = (z!)_p$, and there are no memory events involving z in $\hat{h}$. Pick an event e from $\hat{h}$; in other words, $\hat{h} = \hat{h}_l\ e\ \hat{h}_r$. The event e can be moved either to the left of $\bar{w}$ or to the right of w. More precisely:*

1. *If $E_{hb}@\bar{w} \not\subseteq E_{hb}@e$, then $\bar{w}\ \hat{h}_l\ e \sqsubseteq \ldots e\ \bar{w} \ldots$*
2. *If $E_{hb}@\bar{w} \subseteq E_{hb}@e$, then $e\ \hat{h}_r\ w \sqsubseteq \ldots w\ e \ldots$*

Proof. The proof in by induction on the size of $\hat{h}_l$ and $\hat{h}_r$.

Case: $E_{hb}@\bar{w} \not\subseteq E_{hb}@e$
Here we need to prove that e an be moved to the left of $\bar{w}$ in $\bar{w}\ \hat{h}_l\ e$. In the base case $|\hat{h}_l| = 0$ and we need to show that $\bar{w} \bowtie e$. Since $E_{hb}@\bar{w} \not\subseteq E_{hb}@e$, then $e = (o)_{p'}$ with $p' \neq p$ or $e = (\texttt{rend}^i\ c)^{p'}_{p''}$ with $p' \neq p$ and $p'' \neq p$. Given the assumption that e is not a memory event involving z, we have that $\bar{w} \bowtie e$, thus $\bar{w}\ e \sqsubseteq e\ \bar{w}$ via $\sqsubseteq$-CONC.

For $\hat{h}_l$ such that $|\hat{h}_l| = n+1$, let e' be the last element of $\hat{h}_l$.

If $E_{hb}@\bar{w} \not\subseteq E_{hb}@e'$ then, since the length of the sub-trace between $\bar{w}$ and e' is n, we can apply the induction hypothesis to move e' to the left of $\bar{w}$. Once e' has been moved, the length of the sub-trace between $\bar{w}$ and e is less than $n+1$, which means we can again apply the induction hypothesis:

$$
\begin{array}{lll}
 & \bar{w} \ldots e'\ e & \text{(induction hypothesis and } E_{hb}@\bar{w} \not\subseteq E_{hb}@e') \\
\sqsubseteq & \ldots e'\ \bar{w} \ldots e & \text{(induction hypothesis and } E_{hb}@\bar{w} \not\subseteq E_{hb}@e) \\
\sqsubseteq & \ldots e'\ \ldots e\ \bar{w} \ldots &
\end{array}
$$

If $E_{hb}@\bar{w} \subseteq E_{hb}@e'$, then:

$$
\begin{array}{lll}
 & \bar{w} \ldots e'\ e & (e' \bowtie e \text{ via Lemma 8}) \\
\sqsubseteq & \bar{w} \ldots e\ e' & \text{(induction hypothesis and } E_{hb}@\bar{w} \not\subseteq E_{hb}@e) \\
\sqsubseteq & \ldots e\ \bar{w} \ldots e' &
\end{array}
$$

Case: $E_{hb}@\bar{w} \subseteq E_{hb}@e$
In the base case $|\hat{h}_r| = 0$ and we need to show that $w \bowtie e$. Consider the case of $e = (o)_{p'}$. If $p' = p$, then $E_{hb}@\bar{w} \subseteq E_{hb}@e \subseteq E_{hb}@w$, which contradicts Lemma 2. Similar for the case of $e = (\texttt{rend}^i\ c)^{p'}_{p''}$ and $p' = p$ or $p'' = p$. Therefore, e is an event not involving p. Since, by assumption, e is also an event not involving z, then $e \bowtie w$.

For $\hat{h}_r$ such that $|\hat{h}_r| = n+1$, let e' be the first element of $\hat{h}_r$.

If $E_{hb}@\bar{w} \subseteq E_{hb}@e'$, then

$$
\begin{array}{ll}
 & e\ e' \ldots w \qquad \text{(induction hypothesis and } E_{hb}@\bar{w} \subseteq E_{hb}@e') \\
\sqsubseteq & e \ldots w\ e' \ldots \qquad \text{(induction hypothesis and } E_{hb}@\bar{w} \subseteq E_{hb}@e) \\
\sqsubseteq & \ldots w\ e\ e' \ldots
\end{array}
$$

If $E_{hb}@\bar{w} \not\subseteq E_{hb}@e'$, then

$$
\begin{array}{ll}
 & e\ e' \ldots w \qquad (e' \bowtie e \text{ via Lemma 8}) \\
\sqsubseteq & e'\ e \ldots w \qquad \text{(induction hypothesis and } E_{hb}@\bar{w} \subseteq E_{hb}@e) \\
\sqsubseteq & e'\ \ldots w\ e \ldots
\end{array}
$$

□

Theorem 10. *(Soundness of the data-race detector with respect to WaW races). If* $S_0 \xrightarrow{h} S$ *is a run flagged by the data-race detector as containing a WaW race, then* $h \sqsubseteq h_{dr}$ *with* h_{dr} *containing a manifest WaW data-race.*

Proof. From Lemma 4 we have that h can be broken down into $h_0\ \bar{w}\ \hat{h}\ w$ with $\bar{w} = (z!)_{\bar{p}}$ and $w = (z!)_p$. From Lemma 9 we have that for every $e \in_{\mathbf{e}} \hat{h}$, e can be moved to either the left of $\bar{w}$ or the right of w, meaning:

- $\bar{w}\ \ldots e \ldots w \sqsubseteq e \ldots \bar{w}\ \ldots w$ or
- $\bar{w}\ \ldots e \ldots w \sqsubseteq \bar{w}\ \ldots w \ldots e$

We proceed by reducing the length of $\hat{h}$ until the resulting trace contains $\bar{w}\ w$ as a subtrace. □

4.3 Completeness of the race detector

We give a detailed sketch of the completeness proof.

Theorem 11. *(Completeness of the data-race detector with respect to WaW races). Let* $S \xrightarrow{h} S'$ *be a run such that* $h \sqsubseteq h_{dr}$ *and* h_{dr} *contains a manifest WaW race; in other words,* $(z!)_{\bar{p}}\ (z!)_p$ *is a subsequence of* h_{dr} *for* $\bar{p} \neq p$. *Then* $S \xrightarrow{h} S'$ *is flagged as containing a data-race by the race detector; in other words, event* $(z!)_p$ *is emitted by a reduction via the rule* R-WRITE-E_{WaW} *rule.*

Let $\bar{w} = (z!)_{\bar{p}}$ and $w = (z!)_p$ be events in h with $h \sqsubseteq h_{dr}$ and $\bar{w}\ w$ a subsequence of h_{dr}. Since $\bar{w}\ w$ is a subsequence of h_{dr} and $\bar{w}$ and w are in conflict, meaning $\bar{w} \not\bowtie w$, then $\bar{w}$ must occur before w in h as well. Then $h_0\ \bar{w}\ \hat{h}\ w$ is a subsequence of h. We can prove

that the run $S \xrightarrow{h_0\ \bar{w}\ \hat{h}\ w} S''$ is flagged by the race detector as containing a WaW race. We can do so with a proof by contradiction: assume that $S \xrightarrow{h_0\ \bar{w}\ \hat{h}\ w} S''$ is not flagged as containing a WaW race, meaning, the reduction that emits w is via R-WRITE as opposed to R-WRITE-E$_{WaW}$. This assumption leads us to a contradiction, namely, that a manifest data-race is not derivable from $h_0\ \bar{w}\ \hat{h}\ w$. The argument proceeds as follows.

Since execution did not stop at $\bar{w}$, $\bar{w}$ must have been emitted by the R-WRITE rule. Say $\bar{m}$ is the fresh label generated by the R-WRITE reduction. Then $(\bar{m}, z)$ is placed in the happens-before set of $\bar{p}$. For w to be reduced via R-WRITE as opposed to R-WRITE-E$_{WaW}$, the pair $(\bar{m}, z)$ must be in the happens-before set of p in the configuration preceding w, meaning, in $S \xrightarrow{h_0\ \bar{w}\ \hat{h}}$.[10]

The proof proceeds by looking at the ways in which $(\bar{m}, z)$ becomes part of the happens-before set of p. All such ways involve one or more reductions which emit an event e such that $\bar{w} \not\bowtie e$ and $e \not\bowtie w$. In other words, the manifest WaW race $\bar{w}\ w$ is not derivable from $\bar{w}\ \hat{h}\ w$:

$$\bar{w} \ldots e \ldots w \not\sqsubseteq \ldots \bar{w}\ w \ldots$$

The impossibility of deriving a manifest data-race contradicts our initial assumption and implies that $(\bar{m}, z)$ must not exist in the happens-before set of p. This implication means that w must have been emitted by a reduction via the R-WRITE-E$_{WaW}$ rule. Therefore, the run $S \xrightarrow{h_0\ \bar{w}\ \hat{h}\ w}$ is flagged by the race detector as containing a WaW race.

5 Related work

In this paper, soundness of a race detector refers to the absence of false positives, meaning that a detector only flags racy executions. Soundness is also referred to in the race-detection literature as *precision*. Precise race detectors typically use vector clocks to represent the happen-before relation over operations in a trace [13]. Vector clocks (VC) are, however, expensive. Common operations on VC consume $O(n)$ time and, a VC's representation requires $O(n)$ storage where n is the number of threads spawn during the execution of a program [6]. A line of research in race-detection has been to reduce the dependence on VCs by, for example, employing static analysis to remove runtime checks [6][19].

Another way of achieving better space or time utilization is by letting go of either completeness or precision. By allowing races to sometimes go undetected, sampling race detectors trade completeness for lower a overhead. 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 [12].

[10] Technically $S \xrightarrow{h}$ is a set of configurations, all of which are equivalent from the point of view of the proof.

By going after a proxy instead of an actual race, imprecise race detectors let go of soundness. For example, Eraser's LockSet [21] and Locksmith [18] enforce a lock-based synchronization discipline. A violation of the discipline is a *code smell* but not necessarily a race. Extensions of LockSet that incorporate static analyses have also been made [1]. There also exist hybrid approaches that combine lockset-based detection with happens-before information reconstructed from vector clocks [15].

Golang has a race detector integrated to its tool chain [8]. 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 [9] and the thread sanitizer, TSan, in particular [22, 10]. Thread sanitizer is part of the LLVM's runtime libraries [23, 11]. It works by instrumenting memory accesses as well as 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.

A number of papers address race detection in the context of channel communication [2, 3, 24]. 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 [24], 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 [24] is again 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 [24] can be seen as complementary to our approach. They proposed a type system for checking confluence. We conjecture that their type system can serve as the basis for a static race-detector.

6 Conclusion

We presented a dynamic 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. The operational semantics of the detector is given along with a proof of soundness and completeness. The proofs relate reductions in the semantics to events in a trace grammar. A trace h contains a data race if h' contains a *manifest data race* (see Definition 1) and h' can be derived from h. Thus, the concept of manifest data race and the derive-from relation $\sqsubseteq$ give us an idealized race detector that serves as specification against which we establish the correctness of proposed race detector's operational semantics. Soundness is proven by showing that, if the race detector flags an execution as racy, there exists a manifest data-race in the trace underlying the given program's execution. Similarly, for com-

pleteness, if a trace containing a manifest data-race can be derived from an execution, then the execution is flagged as racy by the detector.

In a previous result [5], we proposed a weak memory model for a Go like language and proved what is called the sequential-consistent data-race freedom guarantee, or SC-DRF for short. The guarantee says that if a program is data-race free, then it behaves sequentially consistent when executing under the proposed weak memory. In the proof, 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. For example, one notable extension would be to statically analyze a target program with the goal of removing dynamic checks. 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] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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's thereby a conventional operational semantics and corresponds to the strong semantics from [5].

$$\begin{aligned} e;\, t &::= \texttt{let}\ r = e\ \texttt{in}\ t \quad \text{when } r \notin \mathit{fn}(t) \\ \texttt{stop} &::= \textstyle\sum_0 \end{aligned}$$

Fig. 16: 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 (1) of the race detecting semantics resp. equation (2). 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.[11]

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 17 concern reduction steps that don't affect the memory or involve channel communication.

[11] Note in passing, also in the formalization of the weak semantics in [5], 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 [5], the additional information is required to describe the (weak) semantics itself.

$$
\begin{array}{rl}
R_1 \parallel R_2 \equiv R_2 \parallel R_1 & \\
(R_1 \parallel R_2) \parallel R_3 \equiv R_1 \parallel (R_2 \parallel R_3) & \\
\bullet \parallel R \equiv R & \\
R_1 \parallel \nu n\, R_2 \equiv \nu n\, (R_1 \parallel R_2) & \text{if } n \notin fn(R_1) \\
\nu n_1\, \nu n_2\, R \equiv \nu n_2\, \nu n_1\, R &
\end{array}
$$

Table 1: Structural congruence

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

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

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

Fig. 17: Local steps

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

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

B Rules for the select statement

Rules dealing with the select statement semantics are given on Figure 18. 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$\perp$ is analogous to R-REC$\perp$. Finally, the default rule (R-SEL-DEF) applies when no other select rule applies.

C Correctness of the independence and derivable-from relations

The correctness of the independence and derivable-from relations of Figure 12 and 13 can be proven with respect to the race detector semantics of Section 2 by interpreting the operational semantics as inducing a labeled transition system (LTS).

Let $(\mathcal{S}, \mathcal{L}, \rightarrow)$ be a labeled transition system over a set of states $\mathcal{S}$, set of labels $\mathcal{L}$ and transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. We write $S \xrightarrow{l} S'$ when $(S, l, S') \in \rightarrow$. If h is the sequence $l_1 :: \ldots :: l_n$, then $S \xrightarrow{h} S'$ if there are $S_1, ..., S_{n-1}$ such that $S \xrightarrow{l_1} S_1 \ldots S_{n-1} \xrightarrow{l_n} S$. We define $S \xrightarrow{h} = \{S' | S \xrightarrow{h} S'\}$. Given $\xrightarrow{a}$ and $\xrightarrow{b}$, we define their composition straightforwardly as $\xrightarrow{a}\xrightarrow{b} = \{(S, S') | \exists \hat{S}.\ S \xrightarrow{a} \hat{S} \xrightarrow{b} S'\}$.

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

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

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

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

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

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

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

Fig. 18: Operational semantics: Select statement

Definition 12 (Swap). *Given a state $S_0 \in \mathcal{S}$ and a sequence of labels $h = l_1, l_2, ..., l_n$, we say that $\xrightarrow{a}$ and $\xrightarrow{b}$ are* swappable *in $S_0 \xrightarrow{h}$ if, for all $S' \in S_0 \xrightarrow{h}$, $S' \xrightarrow{a}\xrightarrow{b} \subseteq S' \xrightarrow{b}\xrightarrow{a}$.*

We call *uniform swap* a swap that holds for all traces h.

Definition 13 (Uniform swap). *Given $\xrightarrow{a}\xrightarrow{b}$, we say that $\xrightarrow{a}$ and $\xrightarrow{b}$ are* uniformly swappable *if $\xrightarrow{a}\xrightarrow{b} \subseteq \xrightarrow{b}\xrightarrow{a}$.*

Let $S_0 \xrightarrow{h::e_1::e_2} S$ be a run starting from the initial state S_0, emitting the trace $h :: e_1 :: e_2$, and terminating on state S. The lemmas below capture the notion of soundness and completeness of the derivable-from relation.

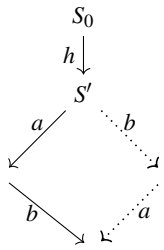

Fig. 19: Swappable in $S_0 \xrightarrow{h}$.

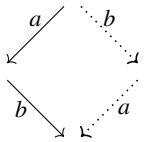

Fig. 20: Uniformly swappable.

Lemma 14 (Soundness). *If the relation $h :: e_1 :: e_2 \sqsubseteq h :: e_2 :: e_1$ is derivable according to the trace rules, then it must be that $S_0 \xrightarrow{h::e_2::e_1} S$.*

Lemma 15 (Completeness). *If $\xrightarrow{e_1}$ and $\xrightarrow{e_2}$ are swappable in $S_0 \xrightarrow{h}$, then $h :: e_1 :: e_2 \sqsubseteq h :: e_2 :: e_1$.*

Note that $\bowtie$ is related *uniform swap*, while $\sqsubseteq$ is related to the concept of *swap*.