Static Race Detection and Mutex Safety and Liveness for Go Programs

Julia Gabet

Imperial College London, United Kingdom
j.gabet18@imperial.ac.uk

Nobuko Yoshida

Imperial College London, United Kingdom
n.yoshida@imperial.ac.uk

Abstract

Go is a popular concurrent programming language thanks to its ability to efficiently combine concurrency and systems programming. In Go programs, a number of concurrency bugs can be caused by a mixture of data races and communication problems. In this paper, we develop a theory based on behavioural types to statically detect data races and deadlocks in Go programs. We first specify lock safety/liveness and data race properties over a Go program model, using the happens-before relation defined in the Go memory model. We represent these properties of programs in a μ-calculus model of types, and validate them using type-level model-checking. We then extend the framework to account for Go's channels, and implement a static verification tool which can detect concurrency errors. This is, to the best of our knowledge, the first static verification framework of this kind for the Go language, uniformly analysing concurrency errors caused by a mix of shared memory accesses and asynchronous message-passing communications.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Software and its engineering → Model checking; Theory of computation → Process calculi

Keywords and phrases Go language, behavioural types, race detection, happens-before relation, safety, liveness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.4

Related Version A full version of the paper is available at [13], `http://arxiv.org/abs/2004.12859`.

Supplementary Material The source code for the tool presented in this paper and instructions to run it are available at [2, 1, 3].

Funding The work is partially supported by VeTSS, EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1 and EP/T014709/1.

Acknowledgements The authors want to thank Nicholas Ng for his initial collaboration on the project.

1 Introduction

Go is a concurrent programming language designed by Google for *programming at scale* [35]. Over the last few years, it has seen rapid growth and adoption: for instance in 2018, major developer surveys [12] show that StackOverflow placed Go in the top 5 most loved and the top 5 most wanted languages; and Github has reported in [14] that Go was the 7th fastest growing language.

One of the core pillars of Go is concurrent programming features, including the locking of shared memory for thread synchronisation, and the use of explicit message passing through channels, inspired by process calculi concurrency models [22, 31]. In practice, shared accesses to memory using locking mechanisms are unavoidable, and could be accidental. It is also of note that both shared memory and message passing operations provide a substantial part of the concurrency features of Go, and are the ones that are more prone to misuse-induced bugs. These unsafe memory accesses may lead to *data races*, where programs silently enter an inconsistent execution state leading to hard-to-debug failures.

```
1  func main() {
2    var x int
3    m := new(sync.RWMutex)
4    go f(m, &x)
5    m.RLock()    // acquire the lock for reading
6    x += 10      // write not protected by the lock
7    m.RUnlock() // release the read-lock
8    m.Lock()     // acquire the lock for writing
9    fmt.Println("x is", x)
10   m.Unlock()   // release the write-lock
11 }
12
13 func f(m *sync.RWMutex, ptr *int) {
14   m.RLock()
15   *ptr += 20   // write not protected by the lock
16   m.RUnlock()
17 }
```

Figure 1 Go program with RWMutex (unsafe)

Figure 1 illustrates a Go program, which makes use of lock `m` to synchronise the `main` and `f` functions updating the content of variable `x`. On line 3, the statement `m := new(sync.RWMutex)` creates a new read-write lock `m`, called `RWMutex` in Go, used to guard memory accesses based on their status as readers or writers. The `RWMutex` object can then be passed around directly as on line 4, circumventing the issue that could arise if we copied the mutex structure instead. It can be locked for writing by calling its `Lock()` method, unlocked from writing handle with its `Unlock()` method, and locked and unlocked for reading with the `RLock()` and `RUnlock()` methods. Readers and writers are mutually exclusive, and writers are mutually exclusive to each other too (hence the name `Mutex`, for *mutual exclusion* lock), but an arbitrary number of readers can hold the lock at the same time. The `go` keyword in front of a function call on line 4 spawns a lightweight thread (called a *goroutine*) to execute the body of function `f`. The two parameters of function `f` – a rwmutex `m`, and an `int` pointer `ptr` – are shared between the caller and callee goroutines, `main` and `f`. Since concurrent access to the shared pointer `ptr` may introduce a data race, the developer tries to ensure serialised, mutually exclusive access to `ptr` in `f` and `x` in `main` by using read-locks. Using read-locks is unsafe in this case, allowing simultaneous write requests to `x` on lines 6 and 15, the program could then output "`x is 20`" with a bad scheduling, dropping the increase of 10 in the same thread as the print statement.

Figure 2 illustrates the same Go program, using the `RWMutex` feature correctly by putting writer sections of the code under writer locks. This alone prevents the data race seen in the first version of the program.

```
func main() {
  var x int
  m := new(sync.RWMutex)
  go f(m, &x)
  m.Lock()     // acquire the lock for writing
  x += 10      // protected by the lock
  m.Unlock()   // release the write-lock
  m.RLock()    // acquire the lock for reading
  fmt.Println("x is", x)
  m.RUnlock() // release the read-lock
}

func f(m *sync.RWMutex, ptr *int) {
  m.Lock()
  *ptr += 20   // protected by the lock
  m.Unlock()
}
```

Figure 2 Go program with RWMutex (safe)

Go provides an optional *runtime* data race detector [48, 15] as a part of the Go compiler toolchain. The race detector is based on LLVM's ThreadSanitizer [40, 45, 41] library, which detects races that manifest during execution. It can be enabled by building a program using the "`-race`" flag. During the program execution, the race detector creates up to four shadow words for every memory object to store historical accesses of the object. It compares every new access with the stored shadow word values to detect possible races. These runtime operations cause high overheads of the runtime detector (5–10 times overhead in memory usage and 2–20 times in execution time on average [15]), hence it is unrealistic to run it with race detection turned on in production code; and because of that, race detection relies on extensive testing or fuzzing techniques [47, 43]. Moreover, as reported in [46], the detector fails to find many non-blocking bugs as it cannot keep a sufficiently long enough history; and its semantics does not capture Go specific non-blocking bugs.

The Go memory model [16] defines the behaviour of memory access in Go as a *happens-before* relation by a *combination* of shared memory and channel communications. It is also reported in [46] that the most difficult bugs to detect are caused when synchronisation mechanisms are used together with message passing operations. For instance, Go can use message passing for sharing memory (channel-as-lock) or passing pointers through channels (pointer-through-channel), which might lead to a serious non-blocking bug, i.e. the program may continue to execute in unwanted and incorrect states or corrupt data in its computations [46], due to subtle interplays with buffered asynchronous communications.

These motivate us to *uniformly* model, statically analyse and detect concurrent non-blocking/blocking shared memory/channel-communications bugs in Go, using a formal model based on a process calculus [22, 31].

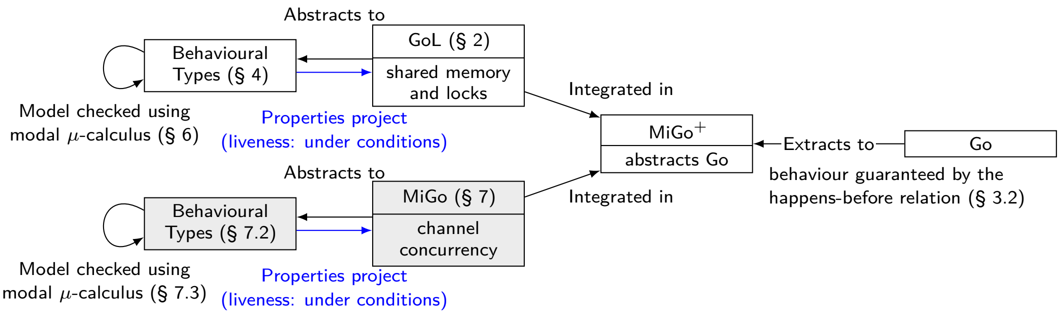

Figure 3 Overview of this paper.

Contributions and Outline. Figure 3 outlines the relationship between the results presented in this paper. This work proposes a *uniform* model which handles first shared memory concurrency (§ 2), and then message-passing concurrency (§ 7) based on *concurrent behavioural types*, and presents the theory, design and implementation of a concurrent bug detector for Go. We formalise a happens-before relation and several key safety and liveness properties in the process calculus following the Go memory model [16] (§ 3). More specifically, in this work, we present the GoldyLocks language (GoL for short), used as a subset of processes of the Go language, and the behavioural types used to model mutual-exclusion locks and shared memory primitives. We then use this calculus and its types to tackle lock liveness and safety, as well as another form of safety: *data race detection.* Our further extension to channels (§ 7) enables us to detect the errors caused by a mixture of shared memory and message passing concurrency. The formulation of a happens-before relation and classification of a data race with respect to the Go memory model along with static analysis of this kind is, to the best of our knowledge, the first of its kind, at least for Go and its mixed memory management features.

Through type soundness and progress theorems of our behavioural typing system (§ 4,§ 5), we are able to represent properties of processes by those of types in the modal μ-calculus (§ 6). In this paper, we explore in particular the formal relationship between type-level properties given by the modal μ-calculus and process properties: we prove which subsets of GoL satisfy the properties of the types characterised by the modal μ-calculus (Theorem 30).

We also present a static analysis tool based on the theory. The tool infers from Go programs [3] the memory accesses, locks and message-passing primitives as behavioural types, and generates a μ-calculus model from these types [2]. We then apply the mCRL2 model checker [8] to detect blocking and non-blocking concurrency errors (§ 8). We conclude the

$$
\begin{array}{rcl}
P, Q, R & \coloneqq & \mu; P \;\mid\; (P \mid Q) \;\mid\; \mathbf{0} \;\mid\; (\nu u)P \\
& \mid & \textbf{if } e \textbf{ then } P \textbf{ else } Q \\
& \mid & X\langle \tilde{e}, \tilde{u} \rangle \;\mid\; \mathsf{new}(x : \sigma); P \\
& \mid & \mathsf{newl}(l); P \;\mid\; \mathsf{newrwl}(l); P \\
& \mid & [x, \sigma :: v] \;\mid\; \lceil l \rceil \;\mid\; \lceil l \rceil^{\star} \\
& \mid & \langle l \rangle_i \;\mid\; \langle l \rangle_i^{\star} \;\mid\; \langle l \rangle_i^{\blacktriangledown}
\end{array}
\qquad
\begin{array}{rcl}
D & \coloneqq & X(\tilde{x}) = P \\
\boldsymbol{P} & \coloneqq & \{D_i\}_{i \in I} \;\mathsf{in}\; P \\
\mu & \coloneqq & \tau \;\mid\; y \leftarrow \mathsf{load}(x) \\
& \mid & \mathsf{store}(x, e) \;\mid\; \ell \\
\ell & \coloneqq & \mathsf{lock}(l) \;\mid\; \mathsf{unlock}(l) \\
& \mid & \mathsf{rlock}(l) \;\mid\; \mathsf{runlock}(l) \\
v & \coloneqq & \mathsf{n} \mid \mathsf{true} \mid \mathsf{false} \mid x \\
e & \coloneqq & v \mid \mathsf{not}(e) \mid \mathsf{succ}(e)
\end{array}
$$

$$
P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathbf{0} \equiv P \qquad (\nu x)\,[x, \sigma :: v] \equiv \mathbf{0}
$$

$$
(\nu l)\lceil l \rceil \equiv \mathbf{0} \qquad (\nu l)\lceil l \rceil^{\star} \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i^{\star} \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i^{\blacktriangledown} \equiv \mathbf{0}
$$

$$
(\nu u)(\nu u')P \equiv (\nu u')(\nu u)P \qquad P \mid (\nu u)Q \equiv (\nu u)\,(P \mid Q) \;\; (u \notin \mathrm{fn}(P))
$$

Figure 4 Syntax of the Process language (top) and Structural Congruence for Stores (bottom).

paper with an overview of related works (§ 9).

Detailed proofs and additional material can be found in the full version of the paper [13]. The tool and benchmark are available from [2, 1, 3].

2 GoL: a Memory-Aware Core Language for Go

This section introduces a core language that models shared memory concurrency, dubbed GoldyLocks (simple subset of Go with shared memory primitives and locks only), shortened as GoL. GoL supports two key features for shared memory concurrency: (1) *shared variables*, created by a shared variable creation primitive, whose values can be read from and written to by multiple threads; and (2) *locks* and *read-write locks* (rwlocks) are modelled by creating a lock store, and recording how it is accessed by (read-)lock and (read-)unlock calls.

2.1 Syntax of GoL

The syntax of the calculus, together with the standard structural congruence $P \equiv P'$ (which includes $\equiv_\alpha$), is given in Figure 4, where e, e' range over *expressions*, x, y over *variables*, l, l' over *locks*, u, u' over *identifiers* (either shared variables or locks) and v over *values* (either local variables, natural numbers or booleans). We write $\tilde{e}$, $\tilde{v}$, $\tilde{x}$ and $\tilde{u}$ for a list of expressions, values, variables and names respectively, and use $\cdot$ as the concatenation operator.

Process syntax $(P, Q, R, ...)$ is given as follows. The *prefix* $\mu; P$ contains either (1) a *silent action* τ; (2) a *store action* of e in $\tilde{x}$, $\mathsf{store}(x, e)$; (3) a *load action* of x, bound to y in the continuation, $y \leftarrow \mathsf{load}(x)$; and (4) actions ($\ell$) for lock/unlock and read-lock/unlock on program locks (denoted by l).

There are three constructs for "new": a *new variable* process $\mathsf{new}(x : \sigma); P$ creates a new shared variable in the heap with payload type σ, binding it to x in the continuation P; a *new lock* process $\mathsf{newl}(l); P$ creates a new program lock and $\mathsf{newrwl}(l); P$ creates a new program read-write lock, binding them to l in the continuation. The syntax includes the *conditional* **if** e **then** P **else** Q, *parallel* process $P \mid Q$, and the *inactive* process $\mathbf{0}$ (often omitted).

A Go program is modelled as a program $\boldsymbol{P}$ in GoL, written $\{D_i\}_{i \in I}$ in P, which consists of a set of mutually recursive process definitions which encode the goroutines and functions used in the program, together with a process P that encodes the program entry point (`main`). The entry point is usually modelled as $X_0\langle\rangle$, a call to a defined process X_0. The entry point is the main process in a collection of mutually recursive process definitions (ranged over by D), parametrised by a list of (expressions and locks) variables.

Process variable X is bound by *definition* D of the form of $X(\tilde{x}) = P$ where $\mathrm{fn}(D) = \emptyset$. This is used by *process call* $X\langle \tilde{e}, \tilde{u} \rangle$ which denotes an instance of the process definition bound to X, with formal parameters instantiated to $\tilde{e}$ and $\tilde{u}$. Note that the entry point could take parameters, if the programmer wants the program to depend on user input data for example, but our examples never make use of that capability.

The part of the syntax denoted by the stores is *runtime* constructs which are generated during the execution (i.e. not written by the programmer and appearing as standalone parallel terms): a shared variable store $[x, \sigma :: v]$ contains message v of type σ; and we represent five internal states of lock stores, situated on the last line of the left column, where the index i is used for rwlocks and the superscripts $\star$ and $\blacktriangledown$ respectively denote locked and waiting locks. *Restriction* $(\nu u)P$ denotes the runtime handle u for a lock or shared variable bound in P, and thus hidden from external processes.

Finally, the notation $\mathrm{fn}(P)$ denotes the sets of free names (locks, shared variables, local variables), ie. ones that have not been bound by a restriction operator (νu), a definition D, a "new" construct, or a load action.

▶ **Example 1** (Processes from Figure 1 and Figure 2). The following process represents the code in Figure 1. We first separate the `main` function in two parts: the part that instantiates the variable and lock, and spawn the side process in parallel to the continuation, that we call X_0; and the rest that processes in parallel to the second goroutine that we put in a separate process P. Process Q is the representation of function `f`, that is run in the second goroutine.

$$\boldsymbol{P}_{\mathsf{race}} := \left\{\begin{array}{ll} X_0 & = \mathsf{new}(x : \mathsf{int}); \mathsf{newrwl}(l); (P\langle x, l\rangle \mid Q\langle x, l\rangle) \\ P(y,z) & = \mathsf{rlock}(z); t_1 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_1 + 10); \mathsf{runlock}(z); \\ & \quad\ \mathsf{rlock}(z); t_2 \leftarrow \mathsf{load}(y); \tau; \mathsf{runlock}(z); \mathbf{0} \\ Q(y,z) & = \mathsf{rlock}(z); t_0 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_0 + 20); \mathsf{runlock}(z); \mathbf{0} \end{array}\right\} \text{ in } X_0\langle\rangle$$

The next process represents the code in Figure 2 in the same fashion as above.

$$\boldsymbol{P}_{\mathsf{safe}} := \left\{\begin{array}{ll} X_0 & = \mathsf{new}(x : \mathsf{int}); \mathsf{newrwl}(l); (P\langle x, l\rangle \mid Q\langle x, l\rangle) \\ P(y,z) & = \mathsf{lock}(z); t_1 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_1 + 10); \mathsf{unlock}(z); \\ & \quad\ \mathsf{rlock}(z); t_2 \leftarrow \mathsf{load}(y); \tau; \mathsf{runlock}(z); \mathbf{0} \\ Q(y,z) & = \mathsf{lock}(z); t_0 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_0 + 20); \mathsf{unlock}(z); \mathbf{0} \end{array}\right\} \text{ in } X_0\langle\rangle$$

2.2 Operational Semantics

The semantics of GoL is given by the labelled transition system (LTS) shown in Figure 5. The LTS system enables us to give a simple and uniform definition of barbs in Definition 5 and a formal correspondence with the modal μ-calculus described in § 6. The LTS rules are written $P \xrightarrow{\alpha} P'$, where α is a label of the form:

$$\boxed{\begin{array}{lll} \alpha := o_{\mathsf{l}} \mid o_{\mathsf{m}}, e & \iota := * \mid 1.\iota \mid 2.\iota & o_{\mathsf{m}} := \mathsf{r}\langle x\rangle \mid \overline{\mathsf{r}\langle x\rangle} \mid (\mathsf{w}\langle x\rangle, \iota) \mid \overline{\mathsf{w}\langle x\rangle} \\ \multicolumn{3}{l}{o_{\mathsf{l}} := \mathsf{l}\langle l\rangle \mid \mathsf{ul}\langle l\rangle \mid \mathsf{rl}\langle l\rangle \mid \mathsf{rul}\langle l\rangle \mid \ulcorner l \urcorner \mid \ulcorner l \urcorner^{\star} \mid \llcorner l \lrcorner \mid \llcorner l \lrcorner^{\blacktriangledown} \mid \llcorner l \lrcorner^{\blacktriangle} \mid \tau_u \mid \tau \qquad o := o_{\mathsf{m}} \mid o_{\mathsf{l}}} \end{array}}$$

They can be either a *data-dependent action* o_{m} along with its data e, used for synchronisation purposes on actions that transmit data, or a *data-independent action* o_{l} alone, used for synchronisation on actions that do not transmit meaningful data, and for the synchronisations τ_u and silent action τ.

The actions in o_{m} define $\mathsf{r}\langle x\rangle$ (read), $(\mathsf{w}\langle x\rangle, \iota)$ (write), $\overline{\mathsf{r}\langle x\rangle}$ and $\overline{\mathsf{w}\langle x\rangle}$ (dual actions) of a shared variable x, where ι denotes an *occurrence* (a position in the parallel composition) that is a string of 1s, 2s and $*$. The actions in o_{l} define (1) $\mathsf{l}\langle l\rangle$ (lock), $\mathsf{ul}\langle l\rangle$ (unlock), $\mathsf{rl}\langle l\rangle$ (read-lock) and $\mathsf{rul}\langle l\rangle$ (read-unlock); (2) lock store actions, $\ulcorner l \urcorner$, $\ulcorner l \urcorner^{\star}$, $\llcorner l \lrcorner$, $\llcorner l \lrcorner^{\blacktriangledown}$ and $\llcorner l \lrcorner^{\blacktriangle}$ (whose purpose is to interact with each action in (1) to produce the lock synchronisation τ_l); as well as (3) synchronisations τ_u and silent actions.

▶ Remark 2. **(1)** The write action $(\mathsf{w}\langle x\rangle, \iota)$ uses occurrence ι to denote the position of the thread which contains that action. By using occurrences, we can differentiate two writes on the same variable happening at the same time, and thereby formally define the notion of data race (see Definition 8); and **(2)** one lock store can produce several different actions which then produce lock synchronisation τ_l with different lock primitives. This allows us to implement the properties with mCRL2 straightforwardly, cf. § 8.

We also define the general label o for actions, which only contains action markers and no data, and will be of use for data-independent marking later on, such as barbs. *Occurences* are ranged over by ι, ι',..., where $*$ denotes the empty occurrence, while $1.\iota$ (resp. $2.\iota$) denotes the left (resp. right) shift of of ι. The left and right shifting operators on action α, $\mathsf{left}\,(\alpha)$ and $\mathsf{right}\,(\alpha)$, are defined as:

$$\mathsf{left}\,((\mathsf{w}\langle x\rangle, \iota), e) = (\mathsf{w}\langle x\rangle, 1.\iota), e \quad \text{and} \quad \mathsf{right}\,((\mathsf{w}\langle x\rangle, \iota), e) = (\mathsf{w}\langle x\rangle, 2.\iota), e$$

with $\mathsf{left}\,(\alpha) = \mathsf{right}\,(\alpha) = \alpha$ if $\alpha \neq (\mathsf{w}\langle x\rangle, \iota), e$. Example 3 will explain the use of these operators with the LTS rules.

Lock and Memory actions

$$\begin{array}{ll}
[\textsc{lck}] & \mathsf{lock}(l); P \xrightarrow{\mathsf{l}\langle l\rangle} P \\
[\textsc{ulck}] & \mathsf{unlock}(l); P \xrightarrow{\mathsf{ul}\langle l\rangle} P \\
[\textsc{rlck}] & \mathsf{rlock}(l); P \xrightarrow{\mathsf{rl}\langle l\rangle} P \\
[\textsc{rulck}] & \mathsf{runlock}(l); P \xrightarrow{\mathsf{rul}\langle l\rangle} P \\
[\textsc{load}] & y \leftarrow \mathsf{load}(x); P \xrightarrow{\mathsf{r}\langle x\rangle, v} P\,\{v/y\} \\
[\textsc{sto}] & \mathsf{store}(x, e); P \xrightarrow{(\mathsf{w}\langle x\rangle, *), e} P \\
\hline
[\textsc{m-lck}] & \lceil l \rceil \xrightarrow{\ulcorner l \urcorner} \lceil l \rceil^\star \\
[\textsc{m-ulck}] & \lceil l \rceil^\star \xrightarrow{\ulcorner l \urcorner^\star} \lceil l \rceil \\
[\textsc{rw-lck}] & \langle l\rangle_0^\blacktriangledown \xrightarrow{\ulcorner l \urcorner} \langle l\rangle_0^\star \\
[\textsc{rw-ulck}] & \langle l\rangle_0^\star \xrightarrow{\ulcorner l \urcorner^\star} \langle l\rangle_0 \\
[\textsc{rw-rlck}] & \langle l\rangle_i \xrightarrow{\llcorner l \lrcorner} \langle l\rangle_{i+1} \\
[\textsc{rw-rulck}] & \langle l\rangle_{i+1} \xrightarrow{\llcorner l \lrcorner^\blacktriangledown} \langle l\rangle_i \\
[\textsc{rw-wait}] & \langle l\rangle_i \xrightarrow{\llcorner l \lrcorner^\blacktriangle} \langle l\rangle_i^\blacktriangledown \\
[\textsc{rw-wulck}] & \langle l\rangle_{i+1}^\blacktriangledown \xrightarrow{\llcorner l \lrcorner^\blacktriangledown} \langle l\rangle_i^\blacktriangledown \\
[\textsc{h-ld}] & [x, \sigma :: v] \xrightarrow{\overline{\mathsf{r}\langle x\rangle}, v} [x, \sigma :: v] \\
[\textsc{h-st}] & [x, \sigma :: v] \xrightarrow{\overline{\mathsf{w}\langle x\rangle}, v'} [x, \sigma :: v']
\end{array}$$

Synchronisation rules

$$[\textsc{c-ld}]\ \frac{P \xrightarrow{\mathsf{r}\langle x\rangle, \tilde{v}} P' \quad Q \xrightarrow{\overline{\mathsf{r}\langle x\rangle}, v} Q'}{P \mid Q \xrightarrow{\tau_x} P' \mid Q}$$

$$[\textsc{c-st}]\ \frac{P \xrightarrow{(\mathsf{w}\langle x\rangle, \iota), e} P' \quad Q \xrightarrow{\overline{\mathsf{w}\langle x\rangle}, v} Q' \quad e \downarrow v}{P \mid Q \xrightarrow{\tau_x} P' \mid Q'}$$

$$[\textsc{c-lck}]\ \frac{P \xrightarrow{\mathsf{l}\langle l\rangle} P' \quad Q \xrightarrow{\ulcorner l \urcorner} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$$

$$[\textsc{c-ulck}]\ \frac{P \xrightarrow{\mathsf{ul}\langle l\rangle} P' \quad Q \xrightarrow{\ulcorner l \urcorner^\star} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$$

$$[\textsc{c-rlck}]\ \frac{P \xrightarrow{\mathsf{rl}\langle l\rangle} P' \quad Q \xrightarrow{\llcorner l \lrcorner} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$$

$$[\textsc{c-rulck}]\ \frac{P \xrightarrow{\mathsf{rul}\langle l\rangle} P' \quad Q \xrightarrow{\llcorner l \lrcorner^\blacktriangledown} Q'}{P \mid Q \xrightarrow{\tau_l} P' \mid Q'}$$

$$[\textsc{c-wait}]\ \frac{P \xrightarrow{\mathsf{l}\langle l\rangle} P' \quad Q \xrightarrow{\llcorner l \lrcorner^\blacktriangle} Q'}{P \mid Q \xrightarrow{\tau} P \mid Q'}$$

$$[\textsc{tau}]\ \tau; P \xrightarrow{\tau} P$$

Runtime structures creation

$$[\textsc{newv}]\ \mathsf{new}(y : \sigma); P \xrightarrow{\tau} (\nu y)\,(P \mid [y, \sigma :: \bot])$$

$$[\textsc{newm}]\ \mathsf{newl}(l); P \xrightarrow{\tau} (\nu l)\,(P \mid \lceil l \rceil) \qquad [\textsc{newrwm}]\ \mathsf{newrwl}(l); P \xrightarrow{\tau} (\nu l)\,(P \mid \langle l\rangle_0)$$

$$[\textsc{par-l}]\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\mathsf{left}(\alpha)} P' \mid Q} \qquad [\textsc{par-r}]\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\mathsf{right}(\alpha)} P \mid Q'} \qquad [\textsc{res1}]\ \frac{P \xrightarrow{\alpha} P' \quad u \notin \mathrm{fn}(\alpha)}{(\nu u)P \xrightarrow{\alpha} (\nu u)P'}$$

$$[\textsc{ift}]\ \frac{e \downarrow \mathsf{true}}{\textbf{if}\ e\,\textbf{then}\ P\,\textbf{else}\ Q \xrightarrow{\tau} P} \qquad [\textsc{iff}]\ \frac{e \downarrow \mathsf{false}}{\textbf{if}\ e\,\textbf{then}\ P\,\textbf{else}\ Q \xrightarrow{\tau} Q} \qquad [\textsc{res2}]\ \frac{P \xrightarrow{\tau_u} P'}{(\nu u)P \xrightarrow{\tau} (\nu u)P'}$$

$$[\textsc{def}]\ \frac{P\,\{\tilde{v},\tilde{u}/\tilde{x}\} \mid Q \xrightarrow{\alpha} R \quad e_i \downarrow v_i \quad X(\tilde{x}) = P \in \{D_i\}_{i \in I}}{X\langle \tilde{e}, \tilde{u}\rangle \mid Q \xrightarrow{\alpha} R} \qquad [\textsc{alpha}]\ \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\beta} P''}{P \xrightarrow{\beta} P''}$$

Figure 5 LTS Reduction Semantics for the Processes.

This LTS defines the semantics of shared variables, locks, and read-write locks which closely follow the specifications in [19]. We first highlight the operational semantics of locks from [17] and rwlocks from [18]. A ***lock*** is a mutual exclusion lock. It must not be copied after its first use: a lock l is created by [NEWM], which is guaranteed fresh by the "(νl)" operation. It is then locked by [C-LCK] and unlocked by [C-ULCK]. A ***read-write lock*** (rwlock) is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a rwlock is an unlocked state. If a goroutine holds a rwlock for reading and another goroutine calls `Lock`, no goroutine should expect to be able to acquire a read-lock until both the initial read-lock and the staged `Lock` call are released. This is to ensure that the lock eventually becomes available to writers; a blocked `Lock` call excludes new readers from acquiring the lock. To model this situation, we annotate a freshly created rwlock by the counter i (instanciated at 0 by [NEWRWM]); this counter is incremented by any fired read-lock (by [C-RLCK]), and blocked from increasing if a `Lock` action gets staged (by [C-WAIT], **note how the `Lock` action is *not* consumed by this rule**); then it is unlocked by read-unlock calls (by [C-RULCK]) until the pending number of read-locks becomes 0, and finally write-locked (by [C-LCK]) and further unlocked by the corresponding unlock (by [C-ULCK]), if a `Lock` was previously staged by [C-WAIT].

A ***shared variable*** is implemented at runtime by a named area in the store, which stores a value of its payload data type, and that can be written to or read by any process within its scope. It is created by [NEWV] with an initial value for declared type σ (0 for int, false for bool, etc.), accessed for reading by [C-LD] and for writing by [C-ST].

The [PAR-∗] rules are explained in Example 3 below.

▶ **Example 3** (Occurrences). Let $P = \mathsf{store}(x, e); P'$, $Q = \mathsf{store}(x, e'); Q'$ and $R = z \leftarrow \mathsf{load}(x); R'$. It follows $P \xrightarrow{(\mathsf{w}\langle x\rangle,*),e} P'$, $Q \xrightarrow{(\mathsf{w}\langle x\rangle,*),e'} Q'$ and $R \xrightarrow{\mathsf{r}\langle x\rangle,v} R'\,\{v/z\}$.
If we compose P and Q, we use [PAR-L] and [PAR-R] to determine the new reductions:

$$P \mid Q \xrightarrow{(\mathsf{w}\langle x\rangle,1.*),e} P' \mid Q \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,*),e) = (\mathsf{w}\langle x\rangle,1.*),e$$
$$P \mid Q \xrightarrow{(\mathsf{w}\langle x\rangle,2.*),e'} P \mid Q' \qquad \mathsf{right}\,((\mathsf{w}\langle x\rangle,*),e') = (\mathsf{w}\langle x\rangle,2.*),e'$$

Composing again, with R:

$$(P \mid Q) \mid R \xrightarrow{(\mathsf{w}\langle x\rangle,1.1.*),e} (P' \mid Q) \mid R \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,1.*),e) = (\mathsf{w}\langle x\rangle,1.1.*),e$$
$$(P \mid Q) \mid R \xrightarrow{(\mathsf{w}\langle x\rangle,1.2.*),e'} (P \mid Q') \mid R \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,2.*),e') = (\mathsf{w}\langle x\rangle,1.2.*),e'$$
$$(P \mid Q) \mid R \xrightarrow{\mathsf{r}\langle x\rangle,v} (P \mid Q) \mid R'\,\{v/z\} \qquad \mathsf{right}\,(\mathsf{r}\langle x\rangle,v) = \mathsf{r}\langle x\rangle,v$$

For process definitions, we implicitly assume the existence of an ambient set of definitions $\{D_i\}_{i\in I}$. Rule [DEF] replaces X by the corresponding process definition (according to the underlying definition environment), instantiating the parameters accordingly. The remaining rules are standard from process calculus literature [36]. We define $\rightarrow$ as $\equiv\xrightarrow{\tau}\equiv \cup \equiv\xrightarrow{\tau_u}\equiv$.

We define a *normal form* for terms, which is used later in § 6:

▶ **Definition 4** (Normal Form). A term P is *in normal form* if $P = (\nu\tilde{u})P'$ and $P' \not\equiv (\nu u)P''$.

We note that, with structural congruence, every well-formed term can be transformed to normal form, and we can then study reduction up to normal form, in order to witness synchronisation actions on channels, memory and mutex.

3 Defining Safety and Liveness: Data Race and Happens-Before

We define the properties of data race freedom and lock safety/liveness through *barbs* (§ 3.1). A ***data race*** happens when two writers (or a reader and a writer) can concurrently access the same shared variable at the same time. ***Unsafe lock access*** happens if (1) unlock happens before lock happens or before waiting read-unlocks release the lock; or (2) read-unlock happens before read-lock happens or after a lock call accesses the process lock. ***Lock liveness*** identifies the ability of (read-)lock requests to always eventually fire. Our first main result is a formalisation of the *happens-before* relation and other properties specified in the Go memory model [16] and a correspondence between a data race characterisation through the happens before relation and another characterisation of a data race through barbs.

3.1 Safety and Liveness Properties through Barbs

We first define barbed process predicates [32] introducing predicates for locks and shared variable accesses. The predicate $P \downarrow_o$ means that P immediately offers a visible action o.

▶ **Definition 5** (Process barbs). The barbs are defined as follows:

Prefix Actions: $\mathsf{store}(x, e) \downarrow_{(\mathsf{w}\langle x\rangle, *)}$; $\quad y \leftarrow \mathsf{load}(x) \downarrow_{\mathsf{r}\langle x\rangle}$; $\quad \mathsf{lock}(l) \downarrow_{\mathsf{l}\langle l\rangle}$;
$\mathsf{unlock}(l) \downarrow_{\mathsf{ul}\langle l\rangle}$; $\quad \mathsf{rlock}(l) \downarrow_{\mathsf{rl}\langle l\rangle}$; $\quad \mathsf{runlock}(l) \downarrow_{\mathsf{rul}\langle l\rangle}$

Programs: if $P \xrightarrow{o,e} P'$ where $o = o_{\mathsf{m}}$ is an action over a shared variable, or $P \xrightarrow{o} P'$ where $o = o_{\mathsf{l}}$ is τ_u or a lock action, then $P \downarrow_o$.

Actions in this case are the same ones as defined before in the operational semantics of GoL, expect for silent action τ. We write $P \Downarrow_o$ if $P \to^* P'$ and $P' \downarrow_o$.

We first define a safety property for locks in Definition 6.

▶ **Definition 6** (Safety). Program $\boldsymbol{P}$ is *safe* if for all P such that $\boldsymbol{P} \to^* (\nu\tilde{u})P$, (a) if $P \downarrow_{\mathsf{ul}\langle l\rangle}$ then $P \downarrow_{\ulcorner l \urcorner^{\star}}$; and (b) if $P \downarrow_{\mathsf{rul}\langle l\rangle}$ then $P \downarrow_{\llcorner l \lrcorner^{\blacktriangledown}}$.

Safety states that in all reachable program states, the unlock action will happen only if the process lock is already locked by the lock action; and the read-unlock will happen only if the process lock is locked by the read-lock action.

Next we define the liveness property: all (read-)lock requests will always eventually fire (i.e. perform a synchronisation).

▶ **Definition 7** (Liveness). Program $\boldsymbol{P}$ is *live* if for all P such that $\boldsymbol{P} \to^* (\nu\tilde{u})P$, if $P \downarrow_{\mathsf{l}\langle l\rangle}$ or $P \downarrow_{\mathsf{rl}\langle l\rangle}$ then $P \Downarrow_{\tau_l}$.

3.2 Happens Before and Data Race

We now define the happens-before relation, closely following [16], and investigate its relationship with data races. The *happens-before* relation between actions o and o', denoted by $P \triangleright o \mapsto o'$, is defined in Figure 6. It is a binary relation which is transitive, non-reflexive and non-symmetric, where $o, o' \in \{(\mathsf{w}\langle x\rangle, \iota), \mathsf{r}\langle x\rangle, \mathsf{l}\langle l\rangle, \mathsf{ul}\langle l\rangle, \mathsf{rl}\langle l\rangle, \mathsf{rul}\langle l\rangle\}$. The operation $\mathsf{left}\,(o)$ denotes that occurrence ι in o changes to $1.\iota$, defined as before by $\mathsf{left}\,((\mathsf{w}\langle x\rangle, \iota)) = (\mathsf{w}\langle x\rangle, 1.\iota)$; otherwise $\mathsf{left}\,(o) = o$. The rules follow the specification in [16].

Rule (con) specifies that within a single goroutine, the happens-before order is the order expressed by the program. Rule (red) gives a form of inheritance: if P reduces to P' and P' has an order between two actions, then P accepts this order as valid as well, as it is a possible future. However, if $P \triangleright o \mapsto o'$, it does not necessarily hold for all of P's reductions.

Rule (par-l) replaces $(\mathsf{w}\langle x\rangle, \iota)$ with $(\mathsf{w}\langle x\rangle, 1.\iota)$ if o or o' is a write action. Rule (par-r) is symmetric. Rules (u-l), (ru-l), (u-rl) and (l-rl) specify the ordering between (read)locks and (read)unlocks, following the reduction semantics.

The following definition states that if a write action happens concurrently with another write action or a read action to the same variable, the program has a data-race.

$$\text{(CON)}\frac{\mu \downarrow_o \quad P \downarrow_{o'}}{\mu; P \rhd o \mapsto o'} \qquad \text{(TRA)}\frac{P \rhd o \mapsto o' \quad P \rhd o' \mapsto o''}{P \rhd o \mapsto o''} \qquad \text{(RED)}\frac{P \to^* P' \quad P' \rhd o \mapsto o'}{P \rhd o \mapsto o'}$$

$$\text{(PAR-L)}\frac{P \rhd o \mapsto o'}{P \mid Q \rhd \mathsf{left}\,(o) \mapsto \mathsf{left}\,(o')} \qquad \text{(PAR-R)}\frac{Q \rhd o \mapsto o'}{P \mid Q \rhd \mathsf{right}\,(o) \mapsto \mathsf{right}\,(o')}$$

$$\text{(U-L)}\frac{P \downarrow_{\mathsf{l}\langle m\rangle} \quad P \downarrow_{\mathsf{ul}\langle m\rangle}}{P \rhd \mathsf{ul}\langle m\rangle \mapsto \mathsf{l}\langle m\rangle} \qquad \text{(RU-L)}\frac{P \downarrow_{\mathsf{l}\langle m\rangle} \quad P \downarrow_{\mathsf{rul}\langle m\rangle}}{P \rhd \mathsf{rul}\langle m\rangle \mapsto \mathsf{l}\langle m\rangle}$$

$$\text{(U-RL)}\frac{P \downarrow_{\mathsf{rl}\langle m\rangle} \quad P \downarrow_{\mathsf{ul}\langle m\rangle}}{P \rhd \mathsf{ul}\langle m\rangle \mapsto \mathsf{rl}\langle m\rangle} \qquad \text{(L-RL)}\frac{P \downarrow_{\mathsf{rl}\langle m\rangle} \quad P \downarrow_{\mathsf{l}\langle m\rangle}}{P \rhd \mathsf{l}\langle m\rangle \mapsto \mathsf{rl}\langle m\rangle}$$

$$\text{(RES)}\frac{P \rhd o \mapsto o' \quad u \notin \mathrm{fn}(o) \cup \mathrm{fn}(o')}{(\nu u)P \rhd o \mapsto o'} \qquad \text{(ALPHA)}\frac{P \rhd o \mapsto o' \quad P \equiv_\alpha Q}{Q \rhd o \mapsto o'}$$

We omit the symmetric rules for most rules ending in a parallel process $P \mid Q$.

Figure 6 Happens-Before Relation

▶ **Definition 8** (Data Race). Program $\boldsymbol{P}$ has a *data race* if there exist two distinct actions $o_1 \neq o_2$, two distinct occurrences $\iota \neq \iota'$, and $\boldsymbol{P} \to^* (\nu\tilde{u})P$, with $o_1 = (\mathsf{w}\langle x\rangle, \iota)$ and $o_2 \in \{(\mathsf{w}\langle x\rangle, \iota'), \mathsf{r}\langle x\rangle\}$, such that $P \Downarrow_{o_1}$, $P \Downarrow_{o_2}$, $\neg(P \rhd o_1 \mapsto o_2)$ and $\neg(P \rhd o_2 \mapsto o_1)$. Program $\boldsymbol{P}$ is *data race free* if it has no data race.

The following theorem states that the data race defined with the happens-before relation coincides with the characterisation given by barbs. The proof is by induction, see [13].

▶ **Theorem 9** (Characterisation of Data Race). $\boldsymbol{P}$ has a data race if and only if there exists P such that $\boldsymbol{P} \to^* (\nu\tilde{u})P$ with $P \downarrow_{o_1}$, $P \downarrow_{o_2}$, $o_1 = (\mathsf{w}\langle x\rangle, \iota)$, $o_2 \in \{(\mathsf{w}\langle x\rangle, \iota'), \mathsf{r}\langle x\rangle\}$ and $\iota \neq \iota'$.

▶ **Example 10** (Processes from Figure 1). We show a possible reduction of $\boldsymbol{P}_{\text{race}}$ in Example 1 that causes the (bad) race.

$$\begin{array}{rcl}
\boldsymbol{P}_{\mathsf{race}} & = & \mathsf{new}(x : \mathsf{int}); \mathsf{newrwl}(l); \left(\begin{array}{l} \mathsf{rlock}(l); t_1 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_1 + 10); \mathsf{runlock}(l); \\ \quad \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{rlock}(l); t_0 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_0 + 20); \mathsf{runlock}(l); \mathbf{0} \end{array}\right) \\
& \to^2 & (\nu x l) \left(\begin{array}{l} \mathsf{rlock}(l); t_1 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_1 + 10); \mathsf{runlock}(l); \\ \quad \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{rlock}(l); t_0 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_0 + 20); \mathsf{runlock}(l); \mathbf{0}[x, \mathsf{int} :: 0] \mid \langle l\rangle_0 \mid \end{array}\right) \\
& \to^6 & (\nu x l) \left(\begin{array}{l} \mathsf{store}(x, 10); \mathsf{runlock}(l); \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{store}(x, 20); \mathsf{runlock}(l); \mathbf{0} \mid [x, \mathsf{int} :: 0] \mid \langle l\rangle_2 \end{array}\right) = (\nu x l)P'
\end{array}$$

Note that the first line is obtained by rewriting using the process definition structure and the [DEF] rule, that tells us the rewritten program and the program with calls share the same reductions. Then we have $P' \downarrow_{(\mathsf{w}\langle x\rangle, 1.1.1.*)}$ and $P' \downarrow_{(\mathsf{w}\langle x\rangle, 1.1.2.*)}$, hence $\boldsymbol{P}_{\mathsf{race}}$ has a data race.

On the other hand, $\boldsymbol{P}_{\mathsf{safe}}$ is data race free, which is ensured by checking every reduction chain of the process for the absence of data race.

4 A Behavioural Typing System for GoL

Our typing system introduces types for locks and shared memory, representing the status of runtime processes accessing to shared variables. It serves as a behavioural abstraction of a valid GoL program, where types take the form of CCS processes with name creation.

$$
\begin{array}{rcl}
T, S & := & \vartheta; T \;\mid\; (T \mid S) \;\mid\; \mathbf{0} \;\mid\; (\nu u)T \\
 & \mid & \oplus\{T_i\}_{i\in I} \;\mid\; \boldsymbol{t}_X\langle\tilde{u}\rangle \;\mid\; \mathsf{new}(x); T \\
 & \mid & \mathsf{newl}(l); T \;\mid\; \mathsf{newrwl}(l); T \\
 & \mid & x^{\blacksquare} \;\mid\; \lceil l \rceil \;\mid\; \lceil l \rceil^{\star} \\
 & \mid & \langle l\rangle_i \;\mid\; \langle l\rangle_i^{\star} \;\mid\; \langle l\rangle_i^{\blacktriangledown}
\end{array}
\qquad
\begin{array}{rcl}
\boldsymbol{T} & := & \{\boldsymbol{t}_i(\tilde{y}_i) = T_i\}_{i\in I}\ \mathsf{in}\ T \\
\vartheta & := & \tau \;\mid\; \mathsf{r}(x) \\
 & \mid & \mathsf{w}(x) \;\mid\; \xi \\
\xi & := & \mathsf{l}(l) \;\mid\; \mathsf{ul}(l) \\
 & \mid & \mathsf{rl}(l) \;\mid\; \mathsf{rul}(l)
\end{array}
$$

$$
T \mid S \equiv S \mid T \qquad T \mid (S \mid S') \equiv (T \mid S) \mid S' \qquad T \mid \mathbf{0} \equiv T \qquad (\nu x)x^{\blacksquare} \equiv \mathbf{0}
$$

$$
(\nu l)\lceil l\rceil \equiv \mathbf{0} \qquad (\nu l)\lceil l\rceil^{\star} \equiv \mathbf{0} \qquad (\nu l)\langle l\rangle_i \equiv \mathbf{0} \qquad (\nu l)\langle l\rangle_i^{\star} \equiv \mathbf{0} \qquad (\nu l)\langle l\rangle_i^{\blacktriangledown} \equiv \mathbf{0}
$$

$$
(\nu u)(\nu u')T \equiv (\nu u')(\nu u)T \qquad T \mid (\nu u)S \equiv (\nu u)\,(T \mid S) \;\; {\scriptstyle (u \notin \mathrm{fn}(T))}
$$

Figure 7 Syntax of the types.

4.1 Behavioural Types with Shared Variables and Mutexes

The syntax of types $(T, S, ...)$ and the structural congruence for the types are given in Figure 7. The type $\vartheta; T$ denotes a store $\mathsf{w}(u)$, load $\mathsf{r}(u)$ of shared variable u, lock $\mathsf{l}(l)$, unlock $\mathsf{ul}(l)$, rlock $\mathsf{rl}(l)$, runlock $\mathsf{rul}(l)$ of a (rw)lock l, followed by the behaviour denoted by type T. It also includes an explicit silent action τ followed by the behaviour T_P.

The type constructs $x^{\blacksquare}$, $\lceil l\rceil$, $\lceil l\rceil^{\star}$, $\langle l\rangle_i$, $\langle l\rangle_i^{\star}$ and $\langle l\rangle_i^{\blacktriangledown}$ denote the type representations of runtime shared variable, unlocked and locked locks, unlocked (or read-locked), locked and lock-waiting rwlocks, respectively. Types for variables and locks include shared variable and (rw)lock creation $\mathsf{new}(x); T$, $\mathsf{newl}(l); T$ and $\mathsf{newrwl}(l); T$ which respectively bind x and l in T. $\mathrm{fn}(T)$ denotes the set of free names of type T.

4.2 Typing System with Shared Variables and Mutexes

Our typing system is defined in Figure 8.

The judgement $(\Gamma \vdash P \blacktriangleright T)$, where Γ is a typing environment that maintains information about locks and shared variables, and types the part of a term explicitly written by the developer. We write $\Gamma \vdash \mathcal{J}$ for $\mathcal{J} \in \Gamma$ and $\Gamma \vdash e{:}\sigma$ to state that the expression e is well-typed according to the types of variables in Γ. We write $u{:}t$ for the typing of a name in generality, which can be (1) $x{:}\mathsf{var}(\sigma)$ to denote a shared variable x with stored value type σ and (2) $l{:}\mathsf{Lock}$ to state that l is a (rw)lock. We omit the rules of expressions e. We write $\mathsf{dom}(\Gamma)$ to denote the set of locks and shared variable bindings in Γ.

The rules are as follows. Rules ⟨LOAD⟩ and ⟨STO⟩ type load and store types for shared variable x where the type of the stored value matches the payload type σ of value x, and the continuation P has type T. Rules ⟨LCK⟩ and ⟨ULCK⟩ (and ⟨RLCK⟩ and ⟨RULCK⟩) type the lock actions in processes by corresponding types. There is no payload type to check, only that the lock name is associated to a lock or read-write lock. Rules ⟨NEWV⟩ and ⟨NEWM⟩ (resp. ⟨NEWRWM⟩) allocate a fresh shared variable name with payload type σ or a lock (resp. rwlock). Other context rules are standard.

The judgement $(\Gamma \vdash_B P \blacktriangleright T)$ types process created during execution of a program and provides the invariants to prove the type safety. B is a set of shared variables and locks with associated runtime buffers to ensure their uniqueness. A shared variable heap is typed with rule ⟨HEAP⟩, and all five states of locks are typed by corresponding lock types. Restriction is typed here, as it takes the relevant type out of the typing context and removes the corresponding name from B.

The judgement $(\Gamma \vdash_B \boldsymbol{P} \blacktriangleright \boldsymbol{T})$ types a program, that consists of a process and a set of runtime stores, accordingly to their respective types.

We use the structural congruence on types to define *normal forms* of types in the same way as done for GoL terms in Definition 4, and study further properties on types up to normal form. Examples of typing of processes can be found in Example 11.

$$\boxed{\Gamma \vdash P \blacktriangleright T} \qquad \langle\text{ZERO}\rangle\ \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}} \qquad \langle\text{NEWV}\rangle\ \frac{\Gamma, x : \mathsf{var}(\sigma) \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{new}(x : \sigma); P \blacktriangleright \mathsf{new}(x); T}$$

$$\langle\text{NEWM}\rangle\ \frac{\Gamma, l : \mathsf{Lock} \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{newl}(l); P \blacktriangleright \mathsf{newl}(l); T} \qquad \langle\text{NEWRWM}\rangle\ \frac{\Gamma, l : \mathsf{Lock} \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{newrwl}(l); P \blacktriangleright \mathsf{newrwl}(l); T}$$

$$\langle\text{LCK}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{lock}(l); P \blacktriangleright \mathsf{l}(l); T} \qquad \langle\text{STO}\rangle\ \frac{\Gamma \vdash x : \mathsf{var}(\sigma) \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{store}(x, e); P \blacktriangleright \mathsf{w}(x); T}$$

$$\langle\text{ULCK}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{unlock}(l); P \blacktriangleright \mathsf{ul}(l); T} \qquad \langle\text{LOAD}\rangle\ \frac{\Gamma \vdash x : \mathsf{var}(\sigma) \quad \Gamma, y : \mathsf{var}(\sigma) \vdash P \blacktriangleright T}{\Gamma \vdash y \leftarrow \mathsf{load}(x); P \blacktriangleright \mathsf{r}(x); T}$$

$$\langle\text{RLCK}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{rlock}(l); P \blacktriangleright \mathsf{rl}(l); T} \qquad \langle\text{RULCK}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{runlock}(l); P \blacktriangleright \mathsf{rul}(l); T}$$

$$\langle\text{TAU}\rangle\ \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T} \qquad \langle\text{VAR}\rangle\ \frac{\Gamma \vdash \tilde{e} : \tilde{\sigma} \quad \Gamma \vdash \tilde{u} : \tilde{t}}{\Gamma, X(\tilde{\sigma}, \tilde{t}) \vdash X\langle \tilde{e}, \tilde{u}\rangle \blacktriangleright \boldsymbol{t}_X\langle \tilde{u}\rangle}$$

$$\langle\text{PAR}\rangle\ \frac{\Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \mid Q \blacktriangleright (T \mid S)} \qquad \langle\text{SEL}\rangle\ \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash \textbf{if } e \textbf{ then } P \textbf{ else } Q \blacktriangleright \oplus\{T, S\}}$$

$$\boxed{\Gamma \vdash_B P \blacktriangleright T} \qquad \langle\text{MUT}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \lceil l \rceil \blacktriangleright \lceil l \rceil} \qquad \langle\text{L-M}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \lceil l \rceil^{\star} \blacktriangleright \lceil l \rceil^{\star}} \qquad \langle\text{RMUT}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i \blacktriangleright \langle l \rangle_i}$$

$$\langle\text{L-RW}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^{\star} \blacktriangleright \langle l \rangle_i^{\star}} \qquad \langle\text{W-RW}\rangle\ \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^{\blacktriangledown} \blacktriangleright \langle l \rangle_i^{\blacktriangledown}} \qquad \langle\text{HEAP}\rangle\ \frac{\Gamma \vdash x : \mathsf{var}(\sigma)}{\Gamma \vdash_{\{x\}} [x, \sigma :: v] \blacktriangleright x^{\blacksquare}}$$

$$\langle\text{RES}\rangle\ \frac{\Gamma, u : t \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B \setminus u} (\nu u)P \blacktriangleright (\nu u)T} \qquad \langle\text{PARR}\rangle\ \frac{\Gamma \vdash_{B_1} P \blacktriangleright T \quad \Gamma \vdash_{B_2} Q \blacktriangleright S \quad {\scriptstyle B_1 \cap B_2 = \emptyset}}{\Gamma \vdash_{B_1 \cup B_2} P \mid Q \blacktriangleright (T \mid S)}$$

$$\boxed{\Gamma \vdash_B \boldsymbol{P} \blacktriangleright \boldsymbol{T}}$$

$$\langle\text{DEF}\rangle\ \frac{\Gamma, X_i(\tilde{\sigma}_i, \tilde{t}_i), \tilde{x}_1 : \tilde{\sigma}_i, \tilde{y}_i : \tilde{t}_i \vdash P_i \blacktriangleright T_i \qquad \Gamma, X_1(\tilde{\sigma}_1, \tilde{t}_1), \ldots, X_n(\tilde{\sigma}_n, \tilde{t}_n) \vdash Q \blacktriangleright S}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i \in I} \text{ in } Q \blacktriangleright \{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S}$$

Figure 8 Typing Rules for Shared Variables and Mutexes.

▶ **Example 11.** The unsafe program of Figure 1, modelled by process $\boldsymbol{P}_{\mathsf{race}}$ in Example 1, has the following type:

$$\boldsymbol{T}_{\mathsf{race}} := \left\{ \begin{array}{ll} \boldsymbol{t}_0 & = \mathsf{new}(x); \mathsf{newrwl}(l); (\boldsymbol{t}_P\langle x, l\rangle \mid \boldsymbol{t}_Q\langle x, l\rangle) \\ \boldsymbol{t}_P(y, z) & = \mathsf{rl}(z); \mathsf{r}(y); \mathsf{w}(y); \mathsf{rul}(z); \mathsf{rl}(z); \mathsf{r}(y); \tau; \mathsf{rul}(z); \mathbf{0} \\ \boldsymbol{t}_Q(y, z) & = \mathsf{rl}(z); \mathsf{r}(y); \mathsf{w}(y); \mathsf{rul}(z); \mathbf{0} \end{array} \right\} \text{ in } \boldsymbol{t}_0\langle\rangle$$

The safe version in Figure 2, modelled by process $\boldsymbol{P}_{\mathsf{safe}}$ in Example 1, has type:

$$\boldsymbol{T}_{\mathsf{safe}} := \left\{ \begin{array}{ll} \boldsymbol{t}_0 & = \mathsf{new}(x); \mathsf{newrwl}(l); (\boldsymbol{t}_P\langle x, l\rangle \mid \boldsymbol{t}_Q\langle x, l\rangle) \\ \boldsymbol{t}_P(y, z) & = \mathsf{l}(z); \mathsf{r}(y); \mathsf{w}(y); \mathsf{ul}(z); \mathsf{rl}(z); \mathsf{r}(y); \tau; \mathsf{rul}(z); \mathbf{0} \\ \boldsymbol{t}_Q(y, z) & = \mathsf{l}(z); \mathsf{r}(y); \mathsf{w}(y); \mathsf{ul}(z); \mathbf{0} \end{array} \right\} \text{ in } \boldsymbol{t}_0\langle\rangle$$

4.3 Operational Semantics of the Behavioural Types

This section defines the semantics of our types. The labels, ranged over by o, o', have the form:

$$\boxed{o := \mathsf{r}\langle x\rangle \mid (\mathsf{w}\langle x\rangle, \iota) \mid \mathsf{l}\langle l\rangle \mid \mathsf{ul}\langle l\rangle \mid \mathsf{rl}\langle l\rangle \mid \mathsf{rul}\langle l\rangle \mid x^{\blacksquare} \mid \ulcorner l \urcorner \mid \ulcorner l \urcorner^{\star} \mid \llcorner l \lrcorner \mid \llcorner l \lrcorner^{\blacktriangledown} \mid \llcorner l \lrcorner^{\blacktriangle} \mid \tau \mid \tau_u}$$

The labels denote the actions introduced in this paper: load and store actions, lock, unlock, rlock and runlock actions, shared heap manipulation, and the five kinds of (rw)lock state transitions. The end of the line is for silent transition and synchronisation over a name.

The semantics of our types is given by the labelled transition system (LTS) (modulo α-conversion), extending that of CCS, which is shown in Figure 9.

Lock and Memory actions

$\lfloor\textsc{lck}\rfloor\quad \mathsf{l}(l); T \xrightarrow{\mathsf{l}\langle l\rangle} T$

$\lfloor\textsc{ulck}\rfloor\quad \mathsf{ul}(l); T \xrightarrow{\mathsf{ul}\langle l\rangle} T$

$\lfloor\textsc{rlck}\rfloor\quad \mathsf{rl}(l); T \xrightarrow{\mathsf{rl}\langle l\rangle} T$

$\lfloor\textsc{rulck}\rfloor\quad \mathsf{rul}(l); T \xrightarrow{\mathsf{rul}\langle l\rangle} T$

$\lfloor\textsc{load}\rfloor\quad \mathsf{r}(x); T \xrightarrow{\mathsf{r}\langle x\rangle} T$

$\lfloor\textsc{sto}\rfloor\quad \mathsf{w}(x); T \xrightarrow{(\mathsf{w}\langle x\rangle, *)} T$

$\lfloor\textsc{m-lck}\rfloor\quad \lceil l\rceil \xrightarrow{\ulcorner l\urcorner} \lceil l\rceil^\star$

$\lfloor\textsc{m-ulck}\rfloor\quad \lceil l\rceil^\star \xrightarrow{\ulcorner l\urcorner^\star} \lceil l\rceil$

$\lfloor\textsc{rw-lck}\rfloor\quad \langle l\rangle_0^\blacktriangledown \xrightarrow{\ulcorner l\urcorner} \langle l\rangle_0^\star$

$\lfloor\textsc{rw-ulck}\rfloor\quad \langle l\rangle_0^\star \xrightarrow{\ulcorner l\urcorner^\star} \langle l\rangle_0$

$\lfloor\textsc{rw-rlck}\rfloor\quad \langle l\rangle_i \xrightarrow{\llcorner l\lrcorner} \langle l\rangle_{i+1}$

$\lfloor\textsc{rw-rulck}\rfloor\quad \langle l\rangle_{i+1} \xrightarrow{\llcorner l\lrcorner^\blacktriangledown} \langle l\rangle_i$

$\lfloor\textsc{rw-wait}\rfloor\quad \langle l\rangle_i \xrightarrow{\llcorner l\lrcorner^\blacktriangle} \langle l\rangle_i^\blacktriangledown$

$\lfloor\textsc{rw-wulck}\rfloor\quad \langle l\rangle_{i+1}^\blacktriangledown \xrightarrow{\llcorner l\lrcorner^\blacktriangledown} \langle l\rangle_i^\blacktriangledown$

$\lfloor\textsc{heap}\rfloor\quad x^\blacksquare \xrightarrow{x^\blacksquare} x^\blacksquare$

Synchronisation rules

$$\lfloor\textsc{c-heap}\rfloor\ \frac{T \xrightarrow{o} T' \quad S \xrightarrow{x^\blacksquare} S' \quad o = (\mathsf{w}\langle x\rangle, \iota), \mathsf{r}\langle x\rangle}{T \mid S \xrightarrow{\tau_x} T' \mid S'}$$

$$\lfloor\textsc{c-lck}\rfloor\ \frac{T \xrightarrow{\mathsf{l}\langle l\rangle} T' \quad S \xrightarrow{\ulcorner l\urcorner} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$$

$$\lfloor\textsc{c-ulck}\rfloor\ \frac{T \xrightarrow{\mathsf{ul}\langle l\rangle} T' \quad S \xrightarrow{\ulcorner l\urcorner^\star} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$$

$$\lfloor\textsc{c-rlck}\rfloor\ \frac{T \xrightarrow{\mathsf{rl}\langle l\rangle} T' \quad S \xrightarrow{\llcorner l\lrcorner} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$$

$$\lfloor\textsc{c-rulck}\rfloor\ \frac{T \xrightarrow{\mathsf{rul}\langle l\rangle} T' \quad S \xrightarrow{\llcorner l\lrcorner^\blacktriangledown} S'}{T \mid S \xrightarrow{\tau_l} T' \mid S'}$$

$$\lfloor\textsc{c-wait}\rfloor\ \frac{T \xrightarrow{\mathsf{l}\langle l\rangle} T' \quad S \xrightarrow{\llcorner l\lrcorner^\blacktriangle} S'}{T \mid S \xrightarrow{\tau_l} T \mid S'}$$

$\lfloor\textsc{tau}\rfloor\quad \tau; T \xrightarrow{\tau} T$

Runtime structures creation

$\lfloor\textsc{newv}\rfloor\quad \mathsf{new}(x); T \xrightarrow{\tau} (\nu x)\left(T \mid x^\blacksquare\right)$

$\lfloor\textsc{newm}\rfloor\quad \mathsf{newl}(l); T \xrightarrow{\tau} (\nu l)\,(T \mid \lceil l\rceil)$

$\lfloor\textsc{newrwm}\rfloor\quad \mathsf{newrwl}(l); T \xrightarrow{\tau} (\nu l)\,(T \mid \langle l\rangle_0)$

Context rules

$$\lfloor\textsc{alpha}\rfloor\ \frac{T \equiv T' \quad T' \xrightarrow{o} T''}{T \xrightarrow{o} T''} \qquad \lfloor\textsc{sel}\rfloor\ \frac{j \in I}{\oplus\{T_i\}_{i\in I} \xrightarrow{\tau} T_j}$$

$$\lfloor\textsc{res1}\rfloor\ \frac{T \xrightarrow{o} T' \quad u \notin \mathrm{fn}(o)}{(\nu u)T \xrightarrow{o} (\nu u)T'} \qquad \lfloor\textsc{res2}\rfloor\ \frac{T \xrightarrow{\tau_u} T'}{(\nu u)T \xrightarrow{\tau} (\nu u)T'} \qquad \lfloor\textsc{par-l}\rfloor\ \frac{T \xrightarrow{o} T'}{T \mid S \xrightarrow{\mathsf{left}(o)} T' \mid S}$$

$$\lfloor\textsc{par-r}\rfloor\ \frac{S \xrightarrow{o} S'}{T \mid S \xrightarrow{\mathsf{right}(o)} T \mid S'} \qquad \lfloor\textsc{def}\rfloor\ \frac{T\,\{\tilde{u}/\tilde{x}\} \mid S \xrightarrow{o} T' \quad \boldsymbol{t}_X(\tilde{x}) = T}{\boldsymbol{t}_X\langle\tilde{u}\rangle \mid S \xrightarrow{o} T'}$$

Figure 9 LTS Reduction Semantics for the Types.

Rules $\lfloor\textsc{sto}\rfloor$ and $\lfloor\textsc{load}\rfloor$ allow a type to emit a store and load action on a shared variable x. Rule $\lfloor\textsc{lck}\rfloor$ (resp. $\lfloor\textsc{ulck}\rfloor$) emits a lock (resp. unlock) action on a shared lock l. Rules $\lfloor\textsc{newv}\rfloor$ and $\lfloor\textsc{newm}\rfloor$ (resp. $\lfloor\textsc{newrwm}\rfloor$) create a a new shared heap x or unlocked lock (resp. rwlock) store l. Rule $\lfloor\textsc{heap}\rfloor$ models the ability of a shared heap to be read or updated at any time, and rule $\lfloor\textsc{c-heap}\rfloor$ allows a load or store action to synchronise with its associated heap.

Rule $\lfloor\textsc{m-lck}\rfloor$ makes a lock to be closed, and rule $\lfloor\textsc{m-ulck}\rfloor$ unlocks a claimed lock. Rules $\lfloor\textsc{c-lck}\rfloor$ and $\lfloor\textsc{c-ulck}\rfloor$ make the corresponding actions to synchronise with their associated lock store. Equivalent rules for rwlocks act the same as in the processes. Pay attention to the same quirk as in processes: $\lfloor\textsc{c-wait}\rfloor$ does not consume the lock action in T, as this rules serves to forbid further read-lock calls from being executed if a lock call is staged.

Rule $\lfloor\textsc{sel}\rfloor$ represents the internal choice behaviour of the conditional processes.

In Figure 9, we omit the symmetric rules for parallel composed processes (such as |C-HEAP|). We write $\to$ for $\equiv\xrightarrow{\tau}\equiv \cup \equiv\xrightarrow{\tau_u}\equiv$ and $T \to^* \xrightarrow{o}$ if there exist T' and T'' such that $T \to^* T' \xrightarrow{o} T''$.

▶ **Example 12.** The unsafe version of Figure 1, modelled by process $\boldsymbol{P}_{\mathsf{race}}$ in Example 1 and typed by $\boldsymbol{T}_{\mathsf{race}}$ in Example 11, has the following possible reduction (following the same reduction order as Example 10):

$$\begin{array}{rcl}
\boldsymbol{T}_{\mathsf{race}} & = & \mathsf{new}(x); \mathsf{newrwl}(l); \left(\begin{array}{l} \mathsf{rl}(l); \mathsf{r}(x); \mathsf{w}(x); \mathsf{rul}(l); \mathsf{rl}(l); \mathsf{r}(x); \tau; \mathsf{rul}(l); \mathbf{0} \\ \mid \mathsf{rl}(l); \mathsf{r}(x); \mathsf{w}(x); \mathsf{rul}(l); \mathbf{0} \end{array} \right) \\
\to^2 & & (\nu x l) \left(\begin{array}{l} \mathsf{rl}(l); \mathsf{r}(x); \mathsf{w}(x); \mathsf{rul}(l); \mathsf{rl}(l); \mathsf{r}(x); \tau; \mathsf{rul}(l); \mathbf{0} \\ \mid \mathsf{rl}(l); \mathsf{r}(x); \mathsf{w}(x); \mathsf{rul}(l); \mathbf{0} \mid x^{\blacksquare} \mid \langle l \rangle_0 \end{array} \right) \\
\to^6 & & (\nu x l) \left(\begin{array}{l} \mathsf{w}(x); \mathsf{rul}(l); \mathsf{rl}(l); \mathsf{r}(x); \tau; \mathsf{rul}(l); \mathbf{0} \\ \mid \mathsf{w}(x); \mathsf{rul}(l); \mathbf{0} \mid x^{\blacksquare} \mid \langle l \rangle_2 \end{array} \right) = (\nu x l) T'
\end{array}$$

We note that T' is a type of P' which has a data race in Example 10.

5 Properties of GoL Processes and Types

This section proves two main results, the subject reduction and progress properties with respect to behavioural types. Our goal is to classify subsets of GoL programs for which liveness, data race freedom and safety coincide with liveness, data race freedom and safety of their types. Detailled proofs for this section are available in [13].

5.1 Type soundness of GoL processes

A basic property for types is to be preserved under structural congruence and to be able to reduce the same as the process.

▶ **Proposition 13** (Subject Congruence). If $\Gamma \vdash_B P \blacktriangleright T$ and $P \equiv P'$, then $\exists T' \equiv T$ such that $\Gamma \vdash_B P' \blacktriangleright T'$.

The following type soundness theorem shows that behaviours of processes can be simulated by behaviours of types.

▶ **Theorem 14** (Subject Reduction). If $\Gamma \vdash_B P \blacktriangleright T$ and $P \to P'$, then $\exists T'$ such that $\Gamma \vdash_B P' \blacktriangleright T'$ and $T \to T'$.

The following progress theorem says that the action availability on types infers that on processes.

We first need to define *barbs* to represent capabilities of a type at a given time in reduction, akin to how process barbs are defined in Definition 5.

▶ **Definition 15** (Type Barbs). The barbs on types are defined as follows:

Prefix Actions: $\mathsf{w}(x) \downarrow_{(\mathsf{w}\langle x \rangle, *)}$; $\mathsf{r}(x) \downarrow_{\mathsf{r}\langle x \rangle}$; $\mathsf{l}(l) \downarrow_{\mathsf{l}\langle l \rangle}$; $\mathsf{ul}(l) \downarrow_{\mathsf{ul}\langle l \rangle}$; $\mathsf{rl}(l) \downarrow_{\mathsf{rl}\langle l \rangle}$; $\mathsf{rul}(l) \downarrow_{\mathsf{rul}\langle l \rangle}$

Types: if $T \xrightarrow{o} T'$ where o is a communication action over a shared variable or τ_u or a lock action, then $T \downarrow_o$.

▶ **Theorem 16** (Progress). Suppose $\Gamma \vdash P \blacktriangleright T$. Then if $T \xrightarrow{o} T_0$ for $o \in \{\tau_u, \tau\}$ for some heap or lock u, then there exists P', T' such that $P \to P'$, $T \xrightarrow{o} T'$, and $\Gamma \vdash P' \blacktriangleright T'$.

To prove this theorem, we use a lemma which shows a correspondence of barbs between processes and types (defined similarly with barbs of processes, cf Definition 15). Note that in Theorem 16, T' and T_0 might be different. This is because a selection type (i.e. the internal choice) can reduce non-deterministically but the corresponding conditional process usually is deterministic.

5.2 Safety and Liveness for Types

In this subsection, we define safety and liveness for types, which correspond to Definitions 6, 7 and 8, respectively.

▶ **Definition 17** (Safety). Type $\boldsymbol{T}$ is *safe* if for all T such that $\boldsymbol{T} \to^* (\nu\tilde{u})T$, (a) if $T \downarrow_{\mathsf{ul}\langle l\rangle}$ then $T \downarrow_{\ulcorner l \urcorner^\star}$; and (b) if $T \downarrow_{\mathsf{rul}\langle l\rangle}$ then $T \downarrow_{\llcorner l \lrcorner^\blacktriangledown}$.

▶ **Definition 18** (Liveness). Type $\boldsymbol{T}$ is *live* if for all T such that $\boldsymbol{T} \to^* (\nu\tilde{u})T$, if $T \downarrow_{\mathsf{l}\langle l\rangle}$ or $T \downarrow_{\mathsf{rl}\langle l\rangle}$ then $T \Downarrow_{\tau_l}$.

▶ **Definition 19** (Data Race). $\boldsymbol{T}$ has a data race if and only if there exists T such that $\boldsymbol{T} \to^* (\nu\tilde{u})T$ with $T \downarrow_{o_1}$, $T \downarrow_{o_2}$, $o_1 = (\mathsf{w}\langle x\rangle, \iota)$, $o_2 \in \{(\mathsf{w}\langle x\rangle, \iota'), \mathsf{r}\langle x\rangle\}$ and $\iota \neq \iota'$.

We say that $\boldsymbol{T}$ is data race free if it has no data race.

5.3 Liveness and Safety for Typed GoL

In this section, we state several propositions and theorems adapted from [27] to our new process and types primitives and their LTSs. Our goal is to classify subsets of GoL programs for which liveness, data race freedom and safety coincide with liveness, data race freedom and safety of their types.

First, we prove that safety and data race freedom (which is a form of safety) have no restriction, and that proving that a type is safe always entails the associated program is safe.

▶ **Theorem 20** (Process Safety and Data Race Freedom). *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is safe (resp. data race free). Then* $\boldsymbol{P}$ *is safe (resp. data race free).*

We then prove that liveness of types is equivalent to liveness of programs for a subset of the GoL programs, in three steps: (1) programs that always have a terminating path, (2) finite branching programs, and (3) programs that simulate non-deterministic branching in infinitely recurring conditionals.

We first study the case of programs that always have a path to termination:

▶ **Definition 21** (May Converging Program). Let $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$. We write $\boldsymbol{P} \in \mathsf{May}_\Downarrow$ if for all $\boldsymbol{P} \to^* P'$, $P' \to^* \mathbf{0}$.

An example of May Converging program is the following program, where process P loops and alternates x to values 1 and 0 until the *end* flag is set, and Q loops reading x until it reads a value 0, in which case it sets the *end* flag and returns:

$$\boldsymbol{P}_{\mathsf{mc}} := \left\{ \begin{array}{lcl} X_0 & = & \mathsf{new}(x : \mathsf{int}); \mathsf{new}(end : \mathsf{bool}); \mathsf{newrwl}(l); \\ & & (P\langle x, end, l\rangle \mid Q\langle x, end, l\rangle) \\ P(x, end, l) & = & \mathsf{lock}(l); y \leftarrow \mathsf{load}(x); \mathsf{store}(x, 1-y); z \leftarrow \mathsf{load}(end); \\ & & \mathsf{unlock}(l); \mathbf{if}\ z\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ P\langle x, end, l\rangle \\ Q(x, end, l) & = & \mathsf{lock}(l); y \leftarrow \mathsf{load}(x); \mathsf{unlock}(l); \\ & & \mathbf{if}\ y = 0\ \mathbf{then}\ \mathsf{lock}(l); \mathsf{store}(end, \mathsf{true}); \mathsf{unlock}(l); \mathbf{0} \\ & & \mathbf{else}\ Q\langle x, end, l\rangle \end{array} \right\} \ \mathsf{in}\ X_0\langle\rangle$$

The next proposition states that on these programs, proving liveness of their types is enough to ensure liveness of the associated program.

▶ **Proposition 22.** *Assume* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is live.* (1) *Suppose there exists* P' *such that* $\boldsymbol{P} \rightarrow^* P' \not\rightarrow$*. Then* $P' \equiv \mathbf{0}$*; and* (2) *If* $\boldsymbol{P} \in \mathsf{May}_{\Downarrow}$*, then* $\boldsymbol{P}$ *is live.*

We now need to define a subset of May Converging programs, that is the set of always terminating programs. This is needed because our implementation, that we describe in § 8, only allows to check and ensure liveness for terminating programs, ie. the result of our tool for liveness is assured to coincide with actual program liveness only on terminating programs.

Note that the tool is able to model check non-terminating programs (under the assumption they don't spawn an unbounded amount of new threads), but may in rare instances lead to a false positive, due to the approximations the model checker has to make in this case.

▶ **Definition 23** (Terminating Program)**.** We write $\boldsymbol{P} \in \mathsf{Terminate}$ if there exists some non-negative number n such that, for all P such that $\boldsymbol{P} \rightarrow^n P$, $P \not\rightarrow$.

The following proposition states that this subset of programs is included in the set of May Converging programs. We note that this inclusion is strict: a program that may loop forever on a select construct, with a timeout branch that terminates the program, is May Converging but not terminating in the sense of the above definition, as we may always find a reduction path that continues longer than any finite bound.

▶ **Proposition 24.** $\boldsymbol{P} \in \mathsf{Terminate}$ *implies* $\boldsymbol{P} \in \mathsf{May}_{\Downarrow}$*.*

Proof. By definition of the May Converging set of programs, all programs that always converge are May Converging. ◀

▶ **Example 25.** Note that the running examples we defined in Figure 1 and 2 are both terminating, and so are their modelling processes given in Example 1.

The next set of programs we highlight is finite branching programs. We first define a series of items, including deterministic marking of conditionals and the set of infinitely branching programs, in order to grab everything not infinitely branching (*ie.* outside of the defined set).

Marked Programs. Given a program $\boldsymbol{P}$ we define its *marking*, written $\mathsf{mark}(\boldsymbol{P})$, as the program obtained by deterministically labelling every occurrence of a conditional of the form $\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q$ in $\boldsymbol{P}$, as $\mathbf{if}^n\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q$, such that n is distinct natural number for all conditionals in $\boldsymbol{P}$.

Marked Reduction Semantics. We modify the marked reduction semantics, written $P \xrightarrow{l} P'$, stating that program P reduces to P' in a single step, performing action l. The grammar of action labels is defined as: $l \coloneqq \alpha \mid n{\cdot}\mathsf{L} \mid n{\cdot}\mathsf{R}$ where α denotes a non-conditional action, taking into account all existing actions and all rules expect [IFT] and [IFF], $n{\cdot}\mathsf{L}$ denotes a conditional branch marked with the natural number n in which the then branch is chosen, and $n{\cdot}\mathsf{R}$ denotes a conditional branch in which the else branch is chosen. Because of the changes in notations, conditional branches are not considered a standard reduction step in $\rightarrow$ any more. The marked reduction semantics replace rules [IFT] and [IFF].

Trace. We define an execution trace of a program P as the potentially infinite sequence of action labels $\vec{l}$ such that $P \xrightarrow{l_1} P_1 \xrightarrow{l_2} \ldots$, with $\vec{l} = \{l_1, l_2 \ldots\}$. We write $\mathbb{T}_P$ for the set of all possible traces of a process P.

Reduction Contexts are given by: $\mathbb{C}_r \ \coloneqq\ [] \mid (P \mid \mathbb{C}_r) \mid (\mathbb{C}_r \mid P) \mid (\nu u)\mathbb{C}_r$.

Infinite Conditional. We say that $\boldsymbol{P}$ has infinite conditionals, written as $\boldsymbol{P} \in \mathsf{Inf}$, iff $\mathsf{mark}(\boldsymbol{P}) \to^* \mathbb{C}_r[\mathbf{if}^n\ e\,\mathbf{then}\ P\,\mathbf{else}\ Q] = R$, for some n, and R has an infinite trace where $n{\cdot}\mathsf{L}$ or $n{\cdot}\mathsf{R}$ appears infinitely often. We say that such an n is an *infinite conditional mark* and write $\mathsf{InfCond}(\boldsymbol{P})$ for the set of all such marks.

We state in the next proposition that finite branching programs can be ensured live by checking for liveness of their types.

▶ **Proposition 26** (Liveness for Finite Branching). *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is live and* $\boldsymbol{P} \notin \mathsf{Inf}$. *Then* $\boldsymbol{P}$ *is live.*

An example of finite branching program is the Dining Philosophers problem:

$$\boldsymbol{P}_{\mathsf{dinephil}} \coloneqq \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(f_1 : \mathsf{int}); \mathsf{new}(f_2 : \mathsf{int}); \mathsf{new}(f_3 : \mathsf{int}); \\ & \quad \mathsf{newl}(l_1); \mathsf{newl}(l_2); \mathsf{newl}(l_3); \\ & \quad \left(\begin{array}{l} P\langle f_1, f_2, l_1, l_2, 1\rangle \mid P\langle f_2, f_3, l_2, l_3, 2\rangle \\ \mid P\langle f_1, f_3, l_1, l_3, 3\rangle \end{array} \right) \\ P(f_l, f_r, l_l, l_r, id) & = \mathsf{lock}(l_l); y \leftarrow \mathsf{load}(f_l); \tau; \mathsf{store}(f_l, id); \\ & \quad \mathsf{lock}(l_r); z \leftarrow \mathsf{load}(f_r); \tau; \mathsf{store}(f_r, id + 2); \\ & \quad \mathsf{unlock}(l_r); \mathsf{unlock}(l_l); P\langle f_l, f_r, l_l, l_r, id\rangle \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

Here, P defines the behaviour of a philosopher, trying to get a hold of both forks assigned to him, and them release them. Other implementations of this problem's algorithm (including ones using channel communications) can be found in the full version [13].

Next we define in the infinite branching programs a subset containing only programs that simulate non-deterministic branching.

Conditional Mapping. The mapping $(\boldsymbol{P})^*$ replaces all occurrences of marked conditionals $\mathbf{if}^n\ e\,\mathbf{then}\ P\,\mathbf{else}\ Q$, such that $n \in \mathsf{InfCond}(\boldsymbol{P})$, with $\mathbf{if}\ \ *\ \mathbf{then}\ P\,\mathbf{else}\ Q$. Its reduction semantics follow the nondeterministic semantics of selection in types, reducing with a τ label. This mapping is applicable to processes P.

Alternating Conditionals. We say that $\boldsymbol{P}$ has *alternating conditional branches*, written $\boldsymbol{P} \in \mathsf{AC}$, iff $\boldsymbol{P} \in \mathsf{Inf}$ and if $\boldsymbol{P} \to^* (\nu\tilde{u})P$ then $P^* \Downarrow_o$ implies $P \Downarrow_o$.

The concurrent version of the Prime Sieve [27, 33] is an example of program that has alternating conditionals. Our implementation of it in Go can be found in the full version [13], and is not detailed here as it uses channels, which we will introduce in an extension to this work in § 7. An other simple example of alternating conditionals is as follows:

$$\boldsymbol{P}_{\mathsf{ac}} \coloneqq \left\{ \begin{array}{l} X_0 \quad\ \ = \mathsf{new}(x : \mathsf{bool}); \mathsf{new}(y : \mathsf{int}); P\langle x, y\rangle \\ P(b, i) = z \leftarrow \mathsf{load}(b); \mathbf{if}\ \ z\,\mathbf{then}\ t \leftarrow \mathsf{load}(i); \\ \qquad\qquad \mathsf{store}(i, t+1); \mathsf{store}(b, \mathsf{not}(z)); P\langle b, i\rangle\ \mathbf{else}\ \mathsf{store}(b, \mathsf{not}(z)); P\langle b, i\rangle \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

We finally state that programs in the alternating conditionals set can be ensured live by ensuring that their types are live.

▶ **Theorem 27** (Liveness). *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ *and* $\boldsymbol{T}$ *is live and* $\boldsymbol{P} \in \mathsf{AC}$. *Then* $\boldsymbol{P}$ *is live.*

To summarise this section, we identified three classes of GoL programs for which we can prove liveness by proving type liveness: (1) programs that always have access to a terminating path (Definition 21 and Proposition 22), including the strict subset of programs that always terminate within a finite number of reduction steps, similar to our running examples; (2) programs that do not exhibit an infinite branch containing an infinitely occurring conditional (Proposition 26), such as the Dining Philosophers problem (used in our benchmarks, see § 8 for more details); and (3) programs with infinite branches that contain infinitely occurring conditionals, with the condition that these infinitely occurring conditionals simulate a non-deterministic choice (Theorem 27), like our Prime Sieve implementation [27, 33] and the example presented above.

6 Verifying Program Properties: the Modal μ-Calculus

In this section, we introduce the modal μ-calculus and express various properties over the types. We then explain how the type-level properties are transposed to process-level properties, as proved in § 5.3.

6.1 The Modal μ-Calculus

We first define a *pointed LTS* for the types, to denote the capabilities available at this point in the simulation.

▶ **Definition 28** (Pointed LTS of types). We define the pointed LTS of a program's types as:

A set of states $\mathcal{S}$, labelled by the (restriction-less) types accessible by reducing from the entrypoint $\boldsymbol{t}_0$ with $\xrightarrow{\tau}$ and $\xrightarrow{\tau_u}$; this entrypoint is defined as the type of the entrypoint X_0 of the program: $\mathcal{S} \coloneqq \{T : \boldsymbol{t}_0 \to^* (\nu\tilde{u})T \text{ and } T \not\equiv (\nu\tilde{u'})T'\}$.

A set of labelled transitions $\mathcal{A}$, in $\mathcal{S}\times\mathcal{S}\times\{\tau,\tau_u\}$: $\mathcal{A} \coloneqq \{(T,T',o) : T,T' \in \mathcal{S} \text{ and } T \xrightarrow{o} T'\}$.

A set of barbs attached to each state, describing the actions its labelled type can take according to the set of barbs of this type. These take the form of the barbs as they were defined above: $\forall T \in \mathcal{S}, \mathcal{F}(T) \coloneqq \{\downarrow_o :\ T\downarrow_o\}$.

The modal μ-calculus is a calculus that allows to express temporal properties on such pointed LTS, like the fact that there exists an accessible state where some property is true, or the fact that some property is true in *all reachable states*. The syntax of these formulae is given below, where α is a set of barbs over the types available to the LTS of types, or transition actions τ or τ_u available as transitions to the LTS of types, as defined above:

$$\begin{array}{lcl} \phi & \coloneqq & \top \mid \bot \mid \neg\phi \mid \phi\wedge\phi \mid \phi\vee\phi \mid \phi\Rightarrow\phi \mid [\alpha]\,\phi \mid \langle\alpha\rangle\,\phi \mid \nu\mathsf{Z}.\phi \mid \mu\mathsf{Z}.\phi \mid \mathsf{Z} \\ \alpha & \coloneqq & \alpha+\alpha \mid\ \downarrow_o \mid\ \downarrow_{\tilde{o}} \mid \tau \mid \tau_u \mid \mathbb{S} \qquad \mathbb{S} \coloneqq \{\tau_u : u \in \mathrm{fn}(T)\} \cup \{\tau\} \end{array}$$

The formulae contain the true and false constants, negation, implication, conjunction and disjunction (both of which can be generalised over a set of actions, where this set can be restrained by some condition).

The *diamond modality*, $\langle\alpha\rangle\,\phi$, is true when at least one of the actions in α is available from the current state and, if it is a barb then ϕ must be true in the current state, and if it is a transition action then ϕ must be true in the resulting state. If no action in α is available, then this formula is false. For example, $\left\langle\downarrow_{(\mathsf{w}\langle z\rangle,*)}\right\rangle\top$ holds on every state where a store action on z is available as the main action, but not when the only store action available is labelled otherwise, e.g. $1.*$.

The *box modality*, $[\alpha]\,\phi$, is valid when, for every state reachable by following an action in α from the current state, ϕ is true. This set of states can be empty, in case no action in α is available, in which case this formula is vacuously true. For example, $[\tau]\,\bot$ is true only when no τ transition is available to the current state of the pointed LTS of the type.

The *lowest fixed point* $\mu\mathsf{Z}.\phi$ and *greatest fixed point* $\nu\mathsf{Z}.\phi$ are the standard recursive constructs, where the least fixed point is the intersection of prefixed points, and the greatest fixed point is the union of postfixed points. That implies the following properties, given for understanding:

1. $\mu \mathsf{Z}.\mathsf{Z} = \bot$: the lowest fixed point defaults to false;
2. $\nu \mathsf{Z}.\mathsf{Z} = \top$: the greatest fixed point defaults to true;
3. if $\phi[\mathsf{Z} := \psi] \Rightarrow \psi$ then $\mu \mathsf{Z}.\phi \Rightarrow \psi$: the lowest fixed point can be expanded on the left of a logical implication;
4. if $\psi \Rightarrow \phi[\mathsf{Z} := \psi]$ then $\psi \Rightarrow \nu \mathsf{Z}.\phi$: the greatest fixed point can be expanded on the right of a logical implication.

To express that some modal μ-calculus formula ϕ is true on a state labelled with type T in the LTS $\mathcal{T}$, we say that T satisfies ϕ in the LTS $\mathcal{T}$, written $T \models_{\mathcal{T}} \phi$.

Two key properties that can be expressed are: ϕ is *always true*, which means that every state T in $\mathcal{T}$ satisfies that formula; and ϕ is *eventually true* which means that there exists a reachable state that satisfies this formula. These are expressed with the fixed-point modalities explained above:

$$\boxed{\text{Always } \phi: \quad \Psi(\phi) = \nu \mathsf{Z}.\phi \wedge [-]\mathsf{Z} \qquad \text{Eventually } \phi: \quad \Phi(\phi) = \mu \mathsf{Z}.\phi \vee \langle - \rangle \mathsf{Z}}$$

6.2 Properties of the Behavioural Types

Figure 10 defines the *local* properties we check on the states of the behavioural types LTS, which means they are defined for *one state only*. The global properties can be checked on the entrypoint of the LTS by checking for $\Psi(\phi)$, ie. "always ϕ".

1. Mutex safety (a):
   $\psi_{s_a} = \bigwedge_l \left\langle \downarrow_{\mathsf{ul}\langle l \rangle} \right\rangle \top \Rightarrow \langle \downarrow_{\ulcorner l \urcorner^\star} \rangle \top$
2. Mutex safety (b):
   $\psi_{s_b} = \bigwedge_l \left\langle \downarrow_{\mathsf{rul}\langle l \rangle} \right\rangle \top \Rightarrow \left\langle \downarrow_{\llcorner l \lrcorner^\blacktriangledown} \right\rangle \top$
3. Mutex liveness:
   $\psi_l = \bigwedge_l \left\langle \downarrow_{\mathsf{l}\langle l \rangle} + \downarrow_{\mathsf{rl}\langle l \rangle} \right\rangle \top \Rightarrow \Phi\left(\langle \tau_l \rangle \top\right)$
4. Data race freedom:
   $\psi_d = \bigwedge_{x,\iota} \left\langle \downarrow_{(\mathsf{w}\langle x \rangle, \iota)} \right\rangle \top \Rightarrow \left[\sum_{\iota' \neq \iota} \downarrow_{(\mathsf{w}\langle x \rangle, \iota')} + \downarrow_{\mathsf{r}\langle x \rangle} \right] \bot$

Figure 10 Modal μ-calculus properties of types

Property ψ_{s_a} checks for the first half of lock safety, that is a lock can only be unlocked if it is currently in locked state, and property ψ_{s_b} checks the second half of lock safety, that is a read/write-lock can only be read-unlocked one level if it is in a read-locked state currently.

Property ψ_l states lock liveness, that is if a lock or read-lock action is staged, the same lock will eventually synchronise (and as such, when applied on a global level $\Psi(\psi_l)$, the lock or read-lock in question will eventually fire, since it becomes false if at any point there is a lock or read-lock staged but no future synchronisation on the lock). Remember that in our model, liveness of the types only entails liveness of the program if the program is in one of the subsets defined previously, in particular if the program terminates or only has alternating conditionals.

Finally, property ψ_d checks local data race freedom, that is if a write action is available on some variable x, then no other read or write action is available on the same variable in the current state. $\Psi(\psi_d)$ checks for data race freedom on the whole of accessible states, so checking that on the entrypoint $\boldsymbol{t}_0$ of a type LTS $\mathcal{T}$ ensures the type of the associated program is data race free, and thus that said program is data race free.

▶ **Example 29.** We can check that the type T' from Example 12 does not verify ψ_d:

$$\psi_d = \left(\left\langle \downarrow_{(\mathsf{w}\langle z \rangle, 1.1.1.*)} \right\rangle \top \Rightarrow \left[\downarrow_{(\mathsf{w}\langle z \rangle, 1.1.2.*)} + \downarrow_{\mathsf{r}\langle z \rangle} \right] \bot \right) \wedge \left(\left\langle \downarrow_{(\mathsf{w}\langle z \rangle, 1.1.2.*)} \right\rangle \top \Rightarrow \left[\downarrow_{(\mathsf{w}\langle z \rangle, 1.1.1.*)} + \downarrow_{\mathsf{r}\langle z \rangle} \right] \bot \right)$$

which is false for T', hence $T' \not\models_{\mathcal{T}_{\mathsf{race}}} \psi_d$: locally, T' has a datarace. Then $\boldsymbol{t}_0 \not\models_{\mathcal{T}_{\mathsf{race}}} \Psi(\psi_d)$, meaning $\boldsymbol{T}_{\mathsf{race}}$ has a data race, since its associated entrypoint in its LTS $\mathcal{T}_{\mathsf{race}}$ does not satisfy data race freedom property $\Psi(\psi_d)$.

On the other hand, the type $\boldsymbol{T}_{\mathsf{safe}}$ from Example 12, modelling the safe version of our running example, verifies the data race freedom property, as well as safety and liveness:

$$\boldsymbol{T}_{\mathsf{safe}} \models_{\mathcal{T}_{\mathsf{safe}}} \Psi(\psi_d) \wedge \Psi(\psi_l) \wedge \Psi(\psi_{s_a} \wedge \psi_{s_b})$$

The types corresponding to the other examples in § 5.3 ($\boldsymbol{P}_{\mathsf{mc}}, \boldsymbol{P}_{\mathsf{dinephil}}$ and $\boldsymbol{P}_{\mathsf{ac}}$) are also safe, live and data race free.

The following theorem states that type-level model-checking can justify process properties under the conditions given in § 5.3. We define the pointed LTS of processes $\mathcal{T}_{\boldsymbol{P}}$ and the satisfaction property $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \phi$ in the same way as they are defined for types in this section.

▶ **Theorem 30** (Model Checking of GoL processes)**.** *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$.

1. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \Psi(\phi)$ *for* $\phi \in \{\psi_{s_a}, \psi_{s_b}, \psi_d\}$*, then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \Psi(\phi)$.
2. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \Psi(\psi_l)$ *and either (a)* $\boldsymbol{P} \in \mathsf{May}\Downarrow$ *or (b)* $\boldsymbol{P} \notin \mathsf{Inf}$ *or (c)* $\boldsymbol{P} \in \mathsf{AC}$*, then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \Psi(\psi_l)$.

Proof. By Theorems 20 and 27, and Propositions 22 and 26. ◀

7 Extending the framework for Go with channels

```
1  func main() {
2    var x int
3    ch := make(chan int, 1)        ⇒ 2
4    go f(ch, &x)
5    ch <- Lock // send to ch
6    x += 10    // protected by ch  ⇒ race
7    <-ch       // receive from ch
8    ch <- Lock
9    fmt.Println("x is", x)
10   <-ch
11 }
12
13 func f(ch chan int, ptr *int) {
14   ch <- Lock
15   *ptr += 20 // protected by ch  ⇒ race
16   <-ch
17 }
```

Figure 11 Go programs: safe (size 1) ⇒ race (size 2)

One of the core features of the Go language is the use of channels for communication in concurrent programming. In Go programs, a number of concurrency bugs can be caused by a *mixture* of data races and communication problems. In this section, we develop a theory which can uniformly analyse concurrency errors caused by a mix of shared memory accesses and asynchronous message-passing communications, integrating coherently our framework in [27, 28]. We include channel communications as a synchronisation primitive in our model for data race checking, following the official Go specification.

Figure 11 illustrates a Go program, which makes use of a channel `ch` to synchronise the `main` and `f` functions updating the content of the shared variable `x`. On line 3, the statement `ch := make(chan int,` *num*`)` creates a new shared channel `ch` with a buffer size of *num* for passing `int` values. Channels can be sent to or received from using the `<-` operator, where `ch <-` *value* and `<-ch` depict sending *value* to the channel and receiving from the channel respectively. At runtime, sending to a full channel (i.e. number of items in channel $\geq num$), or receiving from an empty channel (i.e. number of items in channel $= 0$) blocks. The `go` keyword in front of a function call on line 4 spawns a lightweight thread (called a *goroutine*) to execute the body of function `f`. The two parameters of function `f` – a channel `ch`, and an `int` pointer `ptr` – are shared between the caller and callee goroutines, `main` and `f`. Since concurrent access to the shared pointer `ptr` may introduce a data race, a pair of channel send and receive are used to ensure serialised, mutually exclusive access to `ptr` in `f` and `x` in `main`. If the buffer size of the shared channel is set to 2 by mistake (as denoted by ⇒ in line 3), allowing simultaneous write requests to `x` on lines 6 and 15, the program could output "`x is 20`" with a bad scheduling, dropping the increase of 10 in the same thread as the print statement. We use this program as our running example in this section.

Channel actions

$$\textsc{[snd]}\quad c!\langle e\rangle; P \xrightarrow{\overline{c},e} P$$
$$\textsc{[rvc]}\quad c?(y); Q \xrightarrow{c,v} Q\{^{v}/_{y}\}$$
$$\textsc{[end]}\quad \mathsf{close}\, c; P \xrightarrow{\mathsf{end}[c]} P$$
$$\textsc{[buf]}\quad c\langle\sigma,n\rangle{::}\tilde{v} \xrightarrow{\mathsf{end}[c]} c^{\star}\langle\sigma,n\rangle{::}\tilde{v}$$
$$\textsc{[push]}\quad \frac{|\tilde{v}| < n}{c\langle\sigma,n\rangle{::}\tilde{v} \xrightarrow{{}^{\bullet}c,v} c\langle\sigma,n\rangle{::}v\cdot\tilde{v}}$$
$$\textsc{[pop]}\quad c\langle\sigma,n\rangle{::}\tilde{v}\cdot v \xrightarrow{c^{\bullet},v} c\langle\sigma,n\rangle{::}\tilde{v}$$
$$\textsc{[cpop]}\quad c^{\star}\langle\sigma,n\rangle{::}\tilde{v}\cdot v \xrightarrow{c^{\star},v} c^{\star}\langle\sigma,n\rangle{::}\tilde{v}$$
$$\textsc{[cld]}\quad c^{\star}\langle\sigma,n\rangle{::}\emptyset \xrightarrow{c^{\star},\perp^{\sigma}} c^{\star}\langle\sigma,n\rangle{::}\emptyset$$

Synchronisation rules

$$\textsc{[close]}\quad \frac{P \xrightarrow{\mathsf{end}[c]} P' \qquad Q \xrightarrow{\overline{\mathsf{end}[c]}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$
$$\textsc{[scom]}\quad \frac{P \xrightarrow{\overline{c},e} P' \qquad Q \xrightarrow{c,v} Q' \qquad e\downarrow v}{(P \mid Q) \mid c\langle\sigma,0\rangle{::}\emptyset \xrightarrow{\tau_c} \big(P' \mid Q'\big) \mid c\langle\sigma,0\rangle{::}\emptyset}$$
$$\textsc{[out]}\quad \frac{P \xrightarrow{\overline{c},e} P' \qquad Q \xrightarrow{{}^{\bullet}c,v} Q' \qquad e\downarrow v}{P \mid Q \xrightarrow{\tau_c} P' \mid Q'}$$
$$\textsc{[in]}\quad \frac{P \xrightarrow{c,v} P' \qquad Q \xrightarrow{c^{\bullet},v} Q' \text{ or } Q \xrightarrow{c^{\star},v} Q'}{P \mid Q \xrightarrow{\tau_c} P' \mid Q'}$$
$$\textsc{[bra]}\quad \frac{\pi_j; P_j \mid P \xrightarrow{\alpha} P' \qquad \alpha\in\{\tau,\tau_c\}}{\mathsf{select}\{\pi_i; P_i\}_{i\in I} \mid P \xrightarrow{\alpha} P'}$$

Runtime creation

$$\textsc{[newc]}\quad \mathsf{newchan}(y{:}\sigma,n); P \xrightarrow{\tau} (\nu c)\,(P\,\{^{c}/_{y}\} \mid c\langle\sigma,n\rangle{::}\emptyset) \qquad (c\notin \mathrm{fn}(P))$$

Figure 12 Remaining LTS Semantics of Processes.

7.1 Channels in Processes

We add to the processes the following constructs to account for *channel actions* (defined as $\pi \coloneqq c!\langle e\rangle \mid c?(x) \mid \tau$) and runtime buffer:

$$P \coloneqq \ldots \mid \pi; P \mid \mathsf{close}\, c; P \mid \mathsf{select}\{\pi_i; P_i\}_{i\in I} \mid \mathsf{newchan}(c{:}\sigma,n); P \mid c\langle\sigma,n\rangle{::}\tilde{v} \mid c^{\star}\langle\sigma,n\rangle{::}\tilde{v}$$

Channels are ranged over by a, b, c, which are from now also included under the generic names u, and sets of channels are ranged over by $\tilde{c}$. The new syntax contains the ability to send and receive messages through channels, in capabilities under prefix π, and the ability to close a channel. There is also a select construct that allows selection between several processes guarded by channel send or receive actions, or a silent action. Lastly, we can create a new channel, and there are two runtime constructs denoting respectively open and closed channel c with payload type σ, allowed buffer size n and current buffered messages $\tilde{v}$.

We add the structural congruence rules for queues, $(\nu c)c\langle\sigma,n\rangle{::}\tilde{v} \equiv \mathbf{0}$ and $(\nu c)c^{\star}\langle\sigma,n\rangle{::}\tilde{v} \equiv \mathbf{0}$, and to the LTS the new corresponding reduction rules, along with their labels, shown in Figure 12. The rules include creating a new channel with [newc]; sending to and receiving from a buffered channel with [out] and [in]; closing a channel with [close]; synchronous communications for channels with buffer size 0 using rule [scom]; and reducing a select construct with [bra].

▶ **Example 31** (Processes from Figure 11). The following process represents the safe version of the code in Figure 11. As in Example 1, we separate the `main` function in two parts, the part that instantiates the variable and channel, and spawn the side process in parallel to the continuation; and two called processes P and Q.

$$\boldsymbol{P}_{\mathsf{c-race}} \coloneqq \left\{\begin{array}{lcl} X_0 & = & \mathsf{new}(x:\mathsf{int}); \mathsf{newchan}(c{:}\mathsf{int}, 2); (P\langle x,c\rangle \mid Q\langle x,c\rangle) \\ P(y,z) & = & z!\langle\mathsf{Lock}\rangle; t_1 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_1+10); z?(u_1); \\ & & z!\langle\mathsf{Lock}\rangle; t_2 \leftarrow \mathsf{load}(y); \tau; z?(u_2); \mathbf{0} \\ Q(y,z) & = & z!\langle\mathsf{Lock}\rangle; t_0 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_0+20); z?(u_0); \mathbf{0} \end{array}\right\} \text{ in } X_0\langle\rangle$$

The unsafe version $\boldsymbol{P}_{\mathsf{c-safe}}$ is the same, replacing the 2 for a 1 in the channel instanciation. This example reduces, like the one with a rwlock, allowing to see the possible data race:

$$\boldsymbol{P}_{\mathsf{c-race}} \to^{6} (\nu x l) \left(\begin{array}{l} \mathsf{store}(x, 10); c?(u_1); \\ \quad c!\langle\mathsf{Lock}\rangle; t_2 \leftarrow \mathsf{load}(x); \tau; c?(u_2); \mathbf{0} \\ \mid \mathsf{store}(x, 20); c?(u_0); \mathbf{0} \mid [x, \mathsf{int} :: 0] \mid c\langle\mathsf{int}, 2\rangle{::}\mathsf{Lock}\cdot\mathsf{Lock} \end{array}\right) = (\nu x l)P'$$

7.2 Liveness and Safety for Channels

To define the liveness and safety properties for channels, we first extend the barbs as follows:

▶ **Definition 32** (Process barbs)**.** The barbs are expanded as follows:

prefix actions: $c?(x) \downarrow_c$; $c!\langle e\rangle \downarrow_{\overline{c}}$.

select: we add the rule: $\dfrac{\forall i \in \{1, ..., n\} : \pi_i; P_i \xrightarrow{o_i} P_i \wedge o_i \neq \tau}{\mathsf{select}\{\pi_i; P_i\}_{i\in\{1,...,n\}} \downarrow_{\{o_1,...,o_n\}}}$

The rest is unchanged, but takes into account end actions, as well as buffer actions.

Next is extending the safety and liveness properties to channels, by adding the following definitions: (1) ***Channel Safety***: A channel can be closed only once, and when closed should not be used to send a message. A closed channel can be used to receive an unbounded number of times though, and will wield a default value of the channel's type when the queue is empty; and (2) ***Channel Liveness***: no channel action blocks indefinitely, ie. all channel actions lead to synchronisation on the channel eventually (or on a channel of the list of guarding actions for a select construct that has no silent action guard).

▶ **Definition 33** (Channel Safety)**.** Program $\boldsymbol{P}$ is *channel safe* if for all P such that $\boldsymbol{P} \to^* (\nu\tilde{u})P$, if $P \downarrow_{c^\star}$ then $\neg(P \Downarrow_{\mathsf{end}[c]})$ and $\neg(P \Downarrow_{\overline{c}})$.

▶ **Definition 34** (Channel Liveness)**.** Program $\boldsymbol{P}$ satisfies *channel liveness* if for all P such that $\boldsymbol{P} \to^* (\nu\tilde{u})P$, (a) if $P \downarrow_c$ or $P \downarrow_{\overline{c}}$ then $P \Downarrow_{\tau_c}$; and (b) if $P \downarrow_{\tilde{o}}$ then $P \Downarrow_{\tau_{c_i}}$ for some $c_i \in \mathrm{fn}(\tilde{o})$.

$$\text{(BUF)}\ \frac{P \downarrow_c \quad P \downarrow_{\overline{c}} \quad |\tilde{v}| = n}{P \mid c\langle\sigma, n\rangle{::}\tilde{v} \rhd c \mapsto \overline{c}} \qquad \text{(BUF-RCV)}\ \frac{P \downarrow_{\overline{c}} \quad \exists j \in I : \pi_j \downarrow_c \quad |\tilde{v}| = n}{(P \mid \mathsf{select}\{\pi_i; Q_i\}_{i\in I}) \mid c\langle\sigma, n\rangle{::}\tilde{v} \rhd c \mapsto \overline{c}}$$

$$\text{(CL-RCV)}\ \frac{P \downarrow_c}{P \mid c^\star\langle\sigma, n\rangle{::}\tilde{v} \rhd c^\star \mapsto c} \qquad \text{(BUF-SND)}\ \frac{P \downarrow_c \quad \exists j \in I : \pi_j \downarrow_{\overline{c}} \quad |\tilde{v}| = n}{(P \mid \mathsf{select}\{\pi_i; Q_i\}_{i\in I}) \mid c\langle\sigma, n\rangle{::}\tilde{v} \rhd c \mapsto \overline{c}}$$

$$\text{(SCOM)}\ \frac{P \downarrow_c \quad P \downarrow_{\overline{c}}}{P \mid c\langle\sigma, n\rangle{::}\emptyset \rhd \overline{c} \mapsto c} \qquad \text{(SCOM-SND)}\ \frac{P \downarrow_c \quad \exists j \in I : \pi_j \downarrow_{\overline{c}}}{(P \mid \mathsf{select}\{\pi_i; Q_i\}_{i\in I}) \mid c\langle\sigma, n\rangle{::}\emptyset \rhd \overline{c} \mapsto c}$$

$$\text{(END)}\ \frac{P \downarrow_{\mathsf{end}[c]}}{P \rhd \mathsf{end}[c] \mapsto c^\star} \qquad \text{(SCOM-RCV)}\ \frac{P \downarrow_{\overline{c}} \quad \exists j \in I : \pi_j \downarrow_c}{(P \mid \mathsf{select}\{\pi_i; Q_i\}_{i\in I}) \mid c\langle\sigma, n\rangle{::}\emptyset \rhd \overline{c} \mapsto c}$$

We omit the symmetric rules for most rules ending in a parallel process $P \mid Q$.

Figure 13 Rest of Go's Happens-Before Relation

The channel synchronisations for the happens-before relation are listed in Figure 13. They consist of channel communication according to the official Go memory model: a send happens-before the corresponding receive, and if the channel buffer size is n, then the k-th receive happens-before the $k + n$-th send. We add on top of that that closing a channel happens-before any default value is received from it, and when a channel is closed, default values are emmited by the closed buffer before the corresponding receive reads it.

We extend our behavioural types with the following constructs, mirroring process constructs, and using the syntax and semantics from [27, 28]:

$$S, T \coloneqq \ldots \mid \kappa; T \mid \mathsf{end}[c]; T \mid \&\{\kappa_i; T_i\}_{i\in I} \mid (\nu\, c^n)T \mid \lfloor c \rceil^n_k \mid c^\star \qquad \kappa \coloneqq \overline{c} \mid c \mid \tau$$

We show the typing rules for added channel constructs, which contain the new type primitives, in Figure 14. We also add the structure rules $(\nu c)\lfloor c \rceil^n_k \equiv \mathbf{0}$ and $(\nu c)c^\star \equiv \mathbf{0}$; and the LTS

$\boxed{\Gamma \vdash P \blacktriangleright T}$

$$\langle\textsc{newc}\rangle\ \frac{\Gamma, y{:}\mathsf{ch}(\sigma,n) \vdash P \blacktriangleright T \quad c \notin \mathsf{dom}(\Gamma) \cup \mathrm{fn}(T)}{\Gamma \vdash \mathsf{newchan}(y{:}\sigma, n); P \blacktriangleright (\nu\, c^n) T\,\{^c\!/_y\}}$$

$$\langle\textsc{snd}\rangle\ \frac{\Gamma \vdash c{:}\mathsf{ch}(\sigma,n) \quad \Gamma \vdash e{:}\sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash c!\langle e\rangle; P \blacktriangleright \overline{c}; T} \qquad \langle\textsc{rvc}\rangle\ \frac{\Gamma \vdash c{:}\mathsf{ch}(\sigma,n) \quad \Gamma, x{:}\sigma \vdash P \blacktriangleright T}{\Gamma \vdash c?(x); P \blacktriangleright c; T}$$

$$\langle\textsc{bra}\rangle\ \frac{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}{\Gamma \vdash \mathsf{select}\{\pi_i; P_i\}_{i\in I} \blacktriangleright \&\{\kappa_i; T_i\}_{i\in I}} \qquad \langle\textsc{end}\rangle\ \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{close}\, c; P \blacktriangleright \mathsf{end}[c]; T}$$

$\boxed{\Gamma \vdash_B P \blacktriangleright T}$

$$\langle\textsc{buf}\rangle\ \frac{\Gamma \vdash c{:}\mathsf{ch}(\sigma,n) \quad |\tilde{v}| = k}{\Gamma \vdash_{\{c\}} c\langle\sigma,n\rangle{::}\tilde{v} \blacktriangleright \lfloor c\rceil^n_k} \qquad \langle\textsc{c-buf}\rangle\ \frac{\Gamma \vdash c{:}\mathsf{ch}(\sigma,n)}{\Gamma \vdash_{\{c\}} c^\star\langle\sigma\rangle{::}\tilde{v} \blacktriangleright c^\star}$$

Figure 14 Typing Rules for Channels.

$\boxed{\text{Channel actions}}$

$\lfloor\textsc{snd}\rfloor\ \overline{c}; T \xrightarrow{\overline{c}} T$

$\lfloor\textsc{rvc}\rfloor\ c; T \xrightarrow{c} T$

$\lfloor\textsc{end}\rfloor\ \mathsf{end}[c]; T \xrightarrow{\mathsf{end}[c]} T$

$\lfloor\textsc{cld}\rfloor\ c^\star \xrightarrow{c^\star} c^\star$

$\lfloor\textsc{buf}\rfloor\ \lfloor c\rceil^n_k \xrightarrow{\mathsf{end}[c]} c^\star$

$$\lfloor\textsc{pop}\rfloor\ \frac{k \geq 1}{\lfloor c\rceil^n_k \xrightarrow{c^\bullet} \lfloor c\rceil^n_{k-1}} \qquad \lfloor\textsc{push}\rfloor\ \frac{k < n}{\lfloor c\rceil^n_k \xrightarrow{{}^\bullet c} \lfloor c\rceil^n_{k+1}}$$

$\boxed{\text{Runtime creation}}$

$\lfloor\textsc{newc}\rfloor\ (\nu\, c^n) T \xrightarrow{\tau} (\nu c)\,(T \mid \lfloor c\rceil^n_0)$

$\boxed{\text{Synchronisation rules}}$

$$\lfloor\textsc{close}\rfloor\ \frac{T \xrightarrow{\mathsf{end}[c]} T' \quad S \xrightarrow{\mathsf{end}[c]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$$

$$\lfloor\textsc{scom}\rfloor\ \frac{T \xrightarrow{\overline{c}} T' \quad S \xrightarrow{c} S'}{(T \mid S) \mid \lfloor c\rceil^0_0 \xrightarrow{\tau_c} (T' \mid S') \mid \lfloor c\rceil^0_0}$$

$$\lfloor\textsc{out}\rfloor\ \frac{T \xrightarrow{\overline{c}} T' \quad S \xrightarrow{{}^\bullet c} S'}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$$

$$\lfloor\textsc{in}\rfloor\ \frac{T \xrightarrow{c} T' \quad S \xrightarrow{o} S' \quad o \in \{c^\bullet, c^\star\}}{T \mid S \xrightarrow{\tau_c} T' \mid S'}$$

$$\lfloor\textsc{bra}\rfloor\ \frac{\kappa_j; T_j \mid T \xrightarrow{o} T' \quad o \in \{\tau, \tau_c\}}{\&\{\kappa_i; T_i\}_{i\in I} \mid T \xrightarrow{o} T'}$$

Figure 15 Remaining LTS Semantics of Types.

semantics for the communication primitives (Figure 15). They correspond to the ones found for the processes.

All results in § 5 hold as-is with the new definitions. We only add the new barbs, like for processes (identical definition), and the following type properties:

▶ **Definition 35** (Channel Safety). Type $\boldsymbol{T}$ is *channel safe* if for all T such that $\boldsymbol{T} \rightarrow^* (\nu\tilde{u})T$, if $T \downarrow_{c^\star}$ then $\neg(T \Downarrow_{\mathsf{end}[c]})$ and $\neg(T \Downarrow_{\overline{c}})$.

▶ **Definition 36** (Channel Liveness). Type $\boldsymbol{T}$ is *channel live* if for all T such that $\boldsymbol{T} \rightarrow^* (\nu\tilde{u})T$, (a) if $T \downarrow_c$ or $T \downarrow_{\overline{c}}$ then $T \Downarrow_{\tau_c}$; and (b) if $T \downarrow_{\tilde{o}}$ then $T \Downarrow_{\tau_{c_i}}$ for some $c_i \in \mathrm{fn}(\tilde{o})$.

They correspond to the ones added for processes, and are integrated in other theorems of § 5.

7.3 Modal μ-Calculus Properties for Channels

1. Channel safety:
   $\psi_s = \bigwedge_c \langle\downarrow_{c^\star}\rangle \top \Rightarrow \Psi\left(\left[\downarrow_{\overline{c}} + \downarrow_{\mathsf{end}[c]}\right] \bot\right)$
2. Channel liveness (a):
   $\psi_{l_a} = \bigwedge_c \langle\downarrow_c + \downarrow_{\overline{c}}\rangle \top \Rightarrow \Phi\left(\langle\tau_c\rangle \top\right)$
3. Channel liveness (b):
   $\psi_{l_b} = \bigwedge_{\tilde{c}} \langle\downarrow_{\tilde{o}}\rangle \top \Rightarrow \Phi\left(\left\langle\sum_{c_i\in\tilde{c}} \tau_{c_i}\right\rangle \top\right)$

Figure 16 Modal μ-calculus properties for channels

With extending to the channel primitives, all definitions in § 6 still hold with added properties in the modal μ-calculus for channel liveness and safety. These are defined in Figure 16.

The model-checking result is also extended as the following theorem to capture the situation where shared memory and message passing co-exist.

▶ **Theorem 37** (Model Checking of GoL processes). *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$.

1. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \Psi(\phi)$ *for* $\phi \in \{\psi_{s_a}, \psi_{s_b}, \psi_s, \psi_d\}$, *then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \Psi(\phi)$.
2. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \Psi(\phi)$ *for* $\phi \in \{\psi_l, \psi_{l_a}, \psi_{l_b}\}$ *and either (a)* $\boldsymbol{P} \in \mathsf{May}\Downarrow$ *or (b)* $\boldsymbol{P} \notin \mathsf{Inf}$ *or (c)* $\boldsymbol{P} \in \mathsf{AC}$, *then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \Psi(\phi)$.

This extension to our framework allows us not only to integrate the previous framework by [27, 28], but also show to some extent the modularity of our memory-based approach. With channels, this extension of GoL is implementing a significant range of the concurrency features of Go, allowing for a range of programs to be model-checked for data races, liveness issues and other safety issues in the use of locks and channels.

7.4 Types and process (program) liveness

There are several categories of processes for which the equivalence between types and process (program) liveness is not ensured: (3) programs that have an infinite conditional that is not an alternating conditional, *if they do not always have a termination path available.* They can be checked by the model checker if they are not in (3), however the result may not coincide with the process liveness; (2) programs that neither have an infinite conditional, nor always have a potential path for termination (e.g. a program that recurses indefinitely without ever having an ending branch available through a `select` construct, without the need of a conditional in the recursing selection); and (3) programs that are not finite control – i.e. programs that spawn an unbounded amount of new processes – because the model-checker will not be able to generate a linear representation of them (see § 8).

Note that for (1) and (2), the tool returns "live" if the types are live, though it may be the case that the programs are not live.

8 Implementation and Evaluation

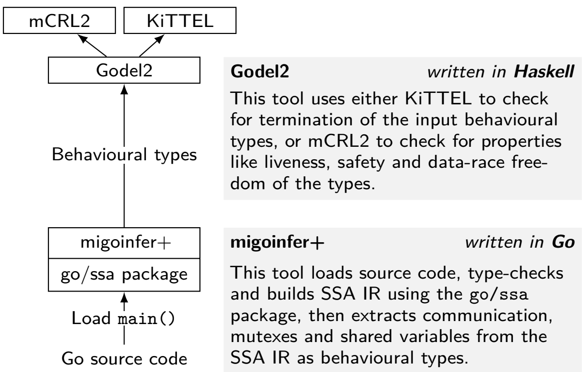

Figure 17 Workflow of the verification toolchain.

The tool chain. Our implementation tool (shown in Figure 17) consists of a type inference tool and a type verifier. The type inference tool (migoinfer+) [3] extracts behavioural types, including eight new primitives related to shared memory: creating a new lock (called mutex in the tool, in reference to the name of the mutual exclusion lock implementation in Go) or shared address, exclusive write-locking or unlocking of a lock or a read-write lock, read-locking/unlocking a read-write lock, and reading or writing a shared variable. This new inference tool supports both channel-based communication primitives from [28] and shared memory primitives.

migoinfer+ currently supports a subset of the Go language syntax, extracting only variables and mutexes created explicitly inside the body of a function, and does not support embedding or mutexes in `struct`. These usage patterns of mutexes can be transformed to the flat representation we support, allowing us to analyse the examples in our benchmark [1]. Note that it is advised to avoid the non-declared sharing of variables, channels and mutexes to a nameless child goroutine, as it may not extract the parameter passing properly, and this is a good practice in Go to specify shared parameters. Programs that spawn an unbounded number of goroutines such as our prime-sieve example can be extracted by migoinfer+ if they respect the above limitations. Lastly, the use of some (non-default) packages, such as the `net` package, is known to break migoinfer+ under certain conditions, making it not extract the types correctly.

The type verifier (Godel2) [2] analyses the new extracted primitives, implements the theory presented in this paper, and uses the mCRL2 [44, 20] model checker as a backend to check safety and data race properties. Regarding the liveness properties, as discussed after Theorem 16 and in [27, 28], liveness of types does not imply liveness of processes, due to conditionals behaving differently in the types and the processes. In Theorem 30, we identified the three classes of Go programs where both liveness properties coincide. One such class is a set of terminating processes, as defined in Definition 23, which is a strict subset of may converging processes (Proposition 24). To make sure liveness coincides on types and processes, we combine the termination checker KITTeL [11] to our tool (see also [28, § 5]). This tool can check processes that are not terminating under certain conditions, namely they should not spawn an unbounded number of threads. However, such programs may, in rare cases, lead to false positives or negatives regarding liveness (and possibly safety), because of the approximations the model checker has to make when running against models with cycles.

Evaluations. We evaluate our tool for reference on an 8-core Intel i7-7700K machine with 16 GB memory, in a 64-bit Linux environment running go 1.12.2. Table 1 shows the results for a range of programs that mix shared memory with either channels or mutexes as locking mechanism. The sources for those examples can be found in the benchmark repository [1]. Programs `no-race` and `simple-race` are programs made to test the behaviour of mutexes and check that liveness errors are properly reported. The channel version of our running example, from Figure 11 is named `channel-as-lock`, and `channel-as-lock-bad` is a variation of the `-fixed` version but with channel sends and receive switched, hence the program deadlocks on the first attempt to lock of each thread as there is nothing to receive.

Table 1 Go Programs Verified by the Toolchain.

Programs	LoC	Sum	Safe	Live	DRF	time (ms)
`no-race`	15	9	✓	✓	✓	691.45
`no-race-mutex`	24	33	✓	✓	✓	785.57
`no-race-mut-bad`	23	20	✓	×	✓	721.77
`simple-race`	13	8	✓	✓	×	701.93
`simple-race-fix`	19	17	✓	✓	✓	731.73
`deposit-race`[1]	18	14	✓	✓	×	697.90
`deposit-fix`[1]	24	27	✓	✓	✓	727.43
`ch-as-lock-race`[2]	19	20	✓	✓	×	753.99
`ch-as-lock-fix`[2]	19	20	✓	✓	✓	745.64
`ch-as-lock-bad`	19	20	✓	×	✓	749.97
`prod-cons-race`	38	156	✓	✓	×	1,903.52
`prod-cons-fix`	40	188	✓	✓	✓	1,971.26
`dine5-unsafe`	35	106	×	✓	✓	6,996.27
`dine5-deadlock`	35	106	✓	×	✓	12,278.33
`dine5-fix`	35	106	✓	✓	✓	8,998.04
`dine5-chan-race`	59	2672	✓	✓	×	∼ 185mn
`dine5-chan-fix`	59	2688	✓	✓	✓	∼ 645mn

[1] [10], [2] Figure 11, LoC: Lines of Code, DRF: Data Race Free, Sum: Summands, ✓: Formula is true, ×: Formula is false

The `deposit` implementation is taken from [10] (the example to present data races and locking mechanisms), and `prod-cons` is a shared memory implementation of the classic producer-consumer algorithm, where two producers race against each other and one consumer takes whichever product is available first. In this example, all three threads share a single memory heap, supposed to be protected by a mutex. Finally, `dine5` is an implementation of the Dining Philosophers problem as explained in § 5.3, and `dine5-chan` is a channel variant adapted slightly to allow for a potential shared-memory data race.

We note that the Prime Sieve algorithm [27, 33] is not analysed by our tool, as it continually spawns new threads, making the state space too big for the mCRL2 model-checker.

Future work for applying this approach to real-world Go programs are: working around the explosion seen with `select`+channels in `dine5-chan`, for which using a different model for `select` constructs and channel actions than the one in our implementation might be sufficient; working on the implementation for a wider range of extractions for channels, shared memory and mutexes embedded in `structs`, or to implement a parser that flattens those structs upstream of migoinfer+; and working on analysis of programs that dynamically spawn new goroutines – this would require non-trivial approximations to be leveraged. Note that it should represent only a small fraction of programs, as most daily-use protocols should be implementable without the need for such unbounded growth in memory usage.

All examples in Table 1 are analysed by our tool, and the time given as an indication scales exponentially with the number of summands (and possibly action labels) and their ordering, in the linear process specification that represents the types in the model checker. Those directly depend from the source code of the analysed program.

9 Conclusion and Related Work

The Go language provides a unique programming environment where both explicit communication and shared memory concurrency primitives co-exist. This work introduces GoL as an abstraction layer for Go code, as well as behavioural types to propose a static verification framework for detecting concurrency bugs in Go. These include deadlocks and safety for both mutual exclusion locks and channel communication, as well as data race detection for shared memory primitives.

Shared memory locks and channels cover by themselves a substantial amount of Go's concurrency features. The former is is a low-level, standard library provision and the latter is a high-level, built-in language feature. Go only features these two basic building blocks because one can use them to implement most higher levels of concurrency abstraction, for example actors models.

The works [27, 28] built behavioural types for verification of concurrency bugs for channel-based message passing. We integrate with their asynchronous calculus (a.k.a. AMiGo) for our channel-related extension in § 7. These works, however, were lacking more shared memory concurrency with locks and shared pointers, and did not tackle data races for shared pointers, which we do. It does not study happens before relations either (for channels). It furthermore was lacking complete proofs on their equivalence theorems for liveness, which is also addressed in this paper. We also proved GoL satisfies the properties of the types characterised by the modal μ-calculus (Theorems 30,37). The paper [28] has informally described them, but these have never been formalised nor proved.

The work [42] defines forkable behaviours (ie. regular expressions with a fork construct) to capture goroutine spawning in synchronous Go programs. They develop a tool based on this model to analyse directly Go programs. Their approach is sound, but suffers from several limitations, which were overcome by [27, 28]; their tool does not treat shared memory concurrency primitives and locks.

The work [25] observed that asynchronous distributed systems can be verified by only modelling *synchronisations* in the core protocol, and introduces a language IceT similar to GoL for specifying synchronisation in message-passing programs. Their focus was to verify functional correctness of the input protocol, and requires input programs to be synchronisable (i.e. no deadlocks nor spurious sends in the input programs). Their approach allows for checking correctness of an implementation, given a reasonable amount of annotations. It is orthogonal to our work in which we only need to check for runtime sanity. Both approaches independently benefit the user, and should be run individually on testing code in order to check both for concurrency behavioural bugs and for implementation bugs.

Recent works [46, 9] provide empirical studies of Go programs, which show that almost half of concurrency bugs in Go are non-blocking bugs, mostly shared memory problems, and the remaining blocking bugs are mostly related to channel and lock misuse. That gives an incentive to make tools and implementations built on the concurrent behavioural theory, for easy detection of such bugs. Our work is part of that effort.

A large body of race detection tools targeting other languages such as Java are available. ThreadSanitizer (TSan) [40, 45, 41] which is included in LLVM/Clang is one of the most widely deployed dynamic race detectors. The runtime race detector of Go [15] uses TSan's runtime library.

The work [30] proposes a subset of the Go language akin to GoL, along with a modular approach to statically analyse processes. Their approach combines lattice-valued regular expressions and a shuffle operator allowing for separate analysis of single threads, and they prove their theory to be sound. They have a prototype implementation in OCaml to check deadlocks in synchronous message-passing programs. The work [6] uses a protocol description language, Scribble [39], which is a practical incarnation of multiparty session types [23] to generate Go APIs, ensuring deadlock freedom and liveness of communications by construction. Neither [30] nor [6] treat either communication error or data race detection, both handled in this paper, nor do they treat shared variables, which our approach extends upon.

The main difference in code writing between Go and GoL is the handling of continuations for select and if-then-else constructs, where Go allows for standard continuation while GoL restrains the user to use tail calls. This is handled by our extraction tool, as it extracts the Go code to GoL by building an SSA representation before extracting relevant primitives from it, see Figure 17 in § 8.

The idea to use the LTS of behavioural types for programming analysis dates back to [34] for Concurrent ML, and since then, it has been applied to many works [5]. Some tackle mutual exclusion locks, but systematically lack support for read-write mutual exclusion locks, including works [24, 4, 21]. The work [26] aims to guarantee liveness with termination of a typed π-calculus. We study wider classes in the theory, aiming termination to use the existing tool (KITTeL) in order to integrate with our tool-chain to scale – thus the main aim and the target (real Go programs in our case) differ from [26].

Type-level model-checking for message-passing programming was first addressed in [7]. Recent applications using mCRL2 include verifications of multiparty session typed π-calculus [37] and the Dotty programming language (the future Scala 3) [38].

Our future works include studying the soundness and completeness of the happens-before relation provided by the Go memory model, ie. studying if the definition of data race given by it covers all data races that can happen in Go, and whether it does not provide false positives; speeding-up the analysis using more mCRL2 options and the extension to an incremental analysis based on happens-before relations, as taken in other languages, e.g. [29, 49]; as well as possibly counter-example extraction for code failing verification, to provide direct access to the detected bugs to developers. There is also the possibility to work on handling dynamic process creation, widening the analysis scope of our current tool and model.

References

1 Godel 2 Benchmarks. `https://github.com/JujuYuki/godel2-benchmark`.

2 Godel 2. `https://github.com/JujuYuki/godel2`.

3 migoinfer+. `https://github.com/JujuYuki/gospal`.

4 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. URL: `https://doi.org/10.1145/1119479.1119480`, `doi:10.1145/1119479.1119480`.

5 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3), 2017. `doi:10.1561/2500000031`.

6 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–30. ACM, 2019. `doi:10.1145/3290342`.

7 Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *POPL'02*, pages 45–57, 2002.

8 Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. *An Overview of the mCRL2 Toolset and Its Recent Advances*, pages 199–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. `doi:10.1007/978-3-642-36742-7\_15`.

9 Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in Go projects. In *SANER*. IEEE, 2019.

10 Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

11 Stephan Falke and Marc Brockschmidt. KITTeL/KoAT. `https://github.com/s-falke/kittel-koat`, 2018.

12 Steve Francia. Nine years of Go. `https://blog.golang.org/9years`, 2018.

13 Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs (extended version). *CoRR*, abs/2004.12859, 2020. URL: `http://arxiv.org/abs/2004.12859`, `arXiv:2004.12859`.

14 GitHub. The fastest growing languages, 2018. `http://octoverse.github.com/`.

15 Go. Data Race Detector. `https://golang.org/doc/articles/race_detector.html`, 2013.

16 Go. The Go Memory Model. `https://golang.org/ref/mem`, 2014.

17 Golang. mutex.go, 2019. `https://golang.org/src/sync/mutex.go`.

18 Golang. rwmutex.go, 2019. `https://golang.org/src/sync/rwmutex.go`.

19 Golang. The Go Programming Language, 2019. `https://golang.org`.

20 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

21 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. URL: `https://doi.org/10.1145/3371074`, `doi:10.1145/3371074`.

22 C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

23 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

24 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 128–141, New York, NY, USA, 2001. ACM. URL: `http://doi.acm.org/10.1145/360204.360215`, `doi:10.1145/360204.360215`.

25 Klaus v. Gleissenthall and Rami Gökhan Kıcı and Alexander Bakst and Deian Stefan and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, January 2019. URL: `http://doi.acm.org/10.1145/3290372`, `doi:10.1145/3290372`.

26 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), May 2008. URL: https://doi.org/10.1145/1745312.1745313, doi:10.1145/1745312.1745313.

27 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 748–761. ACM, 2017. URL: http://doi.acm.org/10.1145/3009837.3009847, doi:10.1145/3009837.3009847.

28 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1137–1148, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3180155.3180157, doi:10.1145/3180155.3180157.

29 Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 359–373, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3192366.3192390, doi:10.1145/3192366.3192390.

30 Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In Andreas Podelski, editor, *Static Analysis*, pages 284–305, Cham, 2018. Springer International Publishing.

31 Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

32 Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *LNCS*, 1992.

33 Nicholas Ng and Nobuko Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CC 2016*, pages 174–184. ACM, 2016.

34 Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL*, 1994. doi:10.1145/174675.174538.

35 Rob Pike. Go at Google. In *SPLASH*, pages 5–6, New York, NY, USA, 2012. ACM.

36 Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

37 Alceste Scalas and Nobuko Yoshida. Less Is More: Multiparty Session Types Revisited. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–29. ACM, 2019.

38 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Programming Language Design and Implementation*, 2019.

39 Scribble. Scribble Project, 2008. www.scribble.org.

40 Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1791194.1791203, doi:10.1145/1791194.1791203.

41 Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-29860-8_9, doi:10.1007/978-3-642-29860-8_9.

42 Kai Stadmüller, Martin Sulzmann, and Peter Thiemann. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*, volume 10017 of *LNCS*, 2016.

43 Syzkaller. Randomized testing for Go. https://github.com/google/syzkaller, 2015.

44 Technische Universiteit Eindhoven. mCRL2. https://www.mcrl2.org/web/user_manual/index.html, 2018.

45 The Clang Team. ThreadSanitizer. http://clang.llvm.org/docs/ThreadSanitizer.html, 2015.

46 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, New York, NY, USA, 2019. ACM. URL: `http://doi.acm.org/10.1145/3297858.3304069`, `doi:10.1145/3297858.3304069`.

47 Dmitry Vyukov. Randomized testing for Go. `https://github.com/dvyukov/go-fuzz`, 2015.

48 Dmitry Vyukov and Andrew Gerrand. Introducing the Go Race Detector. `https://blog.golang.org/race-detector`, 2013.

49 Sheng Zhan and Jeff Huang. Echo: Instantaneous in situ race detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 775–786, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2950290.2950332`, `doi:10.1145/2950290.2950332`.