Ghost Signals

Verifying Termination of Busy Waiting

Tobias Reinhard[1][0000−0003−1048−8735](✉) and Bart Jacobs[1][0000−0002−3605−249X]

imec-DistriNet Research Group, KU Leuven, Leuven, Belgium
{tobias.reinhard,bart.jacobs}@kuleuven.be

Abstract. Programs for multiprocessor machines commonly perform busy waiting for synchronization. We propose the first separation logic for modularly verifying termination of such programs under fair scheduling. Our logic requires the proof author to associate a *ghost signal* with each busy-waiting loop and allows such loops to iterate while their corresponding signal s is not set. The proof author further has to define a well-founded order on signals and to prove that if the looping thread holds an obligation to set a signal s', then s' is ordered above s. By using conventional shared state invariants to associate the state of ghost signals with the state of data structures, programs busy-waiting for arbitrary conditions over arbitrary data structures can be verified.

1 Introduction

Programs for multiprocessor machines commonly perform busy waiting for synchronization [22,23]. In this paper, we propose a separation logic [24,31] to modularly verify termination of such programs under fair scheduling. Specifically, we consider programs where some threads busy-wait for a certain condition C over a shared data structure to hold, e.g., a memory flag being set by other threads. By modularly, we mean that we reason about each thread and each function in isolation. That is, we do not reason about thread scheduling or interleavings. We only consider these issues when proving the soundness of our logic. Assuming fair scheduling is necessary since busy-waiting for a condition C only terminates if the thread responsible for establishing the condition is sufficiently often scheduled to establish C.

Busy waiting is an example of *blocking* behaviour, where a thread's progress *requires interference* from other threads. This is not to be confused with *non-blocking* concurrency, where a thread's progress does not rely on—and may in fact be *impeded* by—interference from other threads. Existing proposed approaches for verifying termination of concurrent programs consider only programs that only involve non-blocking concurrent objects [32], or *primitive blocking constructs* of the programming language, such as acquiring built-in mutexes, receiving from built-in channels, joining threads, or waiting for built-in monitor condition variables [2, 5, 19], or both [11]. Existing techniques that do support busy waiting are not Hoare logics; instead, they verify termination-preserving

contextual refinements between more concrete and more abstract implementations of busy-waiting concurrent objects [15, 21]. In contrast, we here propose the first conventional program logic for modular verification of termination of programs involving busy waiting, using Hoare triples as module specifications.

In order to prove that a busy-waiting loop terminates, we have to prove that it performs only finitely many iterations. To do this we introduce a special form of *ghost resources* [13] which we call *ghost signals*. As ghost resources they only exist on the verification level and hence do not affect the program's runtime behaviour. Signals are initially unset and come with an obligation to set them. Setting a signal does not by definition correspond to any runtime condition. So, in order to use a signal s effectively, anyone using our approach has to prove an invariant stating that s is set if and only if the condition of interest holds. Further, the proof author must prove that every thread discharges all its obligations by performing the corresponding actions, e.g., by setting a signal and establishing the corresponding condition by setting the memory flag.

In our verification approach we tie every busy-waiting loop to a finite set of ghost signals S that correspond to the set of conditions the loop is waiting for. Every iteration that does not terminate the loop must be justified by the proof author proving that some signal $s \in S$ has indeed not been set, yet. This way, we reduce proving termination to proving that no signal is waited for infinitely often.

Our approach ensures that no thread directly or indirectly waits for itself by requiring the proof author (i) to choose a well-founded and partially ordered set of levels $\mathcal{L}evs$ and (ii) to assign a level to every signal and by (iii) only allowing a thread to wait for a signal if the signal's level is lower than the level of each held obligation. This guarantees that every signal is waited for only finitely often and hence that every busy-waiting loop terminates. We use this to prove that every program that is verified using our approach indeed terminates.

We start by gradually introducing the intuition behind our verification approach and the concepts we use. In § 2.1 and § 2.2 we present the main aspects of using signals to verify termination. We start by treating them as physical thread-safe resources and only consider busy waiting for a signal to be set. Then, we drop thread-safety and explain how to prove data-race- and deadlock-freedom. In § 2.3 and § 2.4 we generalize our approach to busy waiting for arbitrary conditions over arbitrary data structures and then lift signals to the verification level by introducing ghost signals.

In § 3 we sketch the verification of a realistic producer-consumer example involving a bounded FIFO to demonstrate our approach's usability and address fine-grained concurrency in § 4. Further, we describe the available tool support in § 5 and discuss integrating higher-order features in § 6. We conclude by comparing our approach to related work and reflecting on it in § 7 and § 8.

We formally define our logic and prove its soundness in the extended version of this paper [28]. To keep the presentation in this paper simple, we assume busy-waiting loops to have a certain syntactical form. In our technical report [29] we present a generalised version of our logic and its soundness proof. Further, we

verify the realistic example presented in § 3 in full detail in the extended version of this paper and in the technical report, using the respective version of our logic. We used our tool support to verify C versions of the bounded FIFO example and the CLH lock. The tool we used and the annotated .c files can be found at [10, 26, 27].

2 A Guide on Verifying Termination of Busy Waiting

When we try to verify termination of busy-waiting programs, multiple challenges arise. Throughout this section, we describe these challenges and our approach to overcome them. In § 2.1 we start by discussing the core ideas of our logic. In order to simplify the presentation we initially consider a simple language with built-in thread-safe *signals* and a corresponding minimal example where one thread busy-waits for such a signal. Signals are heap cells containing boolean values that are specially marked as being solely used for busy waiting. Throughout this section, we generalize our setting as well as our example towards one that allows to verify programs with busy waiting for arbitrary conditions over arbitrary shared data structures. In § 2.2 we present the concepts necessary to verify data-race-, deadlock-freedom and termination in the presence of built-in signals that are not thread safe. In § 2.3 we explain how to use these non-thread-safe signals to verify programs that wait for arbitrary conditions over shared data structures. We illustrate this by an example waiting for a shared heap cell to be set. In § 2.4 we erase the signals from our program and lift them to the verification level in the form of a concept we call *ghost signals*.

2.1 Simplest Setting: Thread-Safe Physical Signals

We want to verify programs that busy-wait for arbitrary conditions over arbitrary shared data structures. As a first step towards achieving this, we first consider programs that busy-wait for simple boolean flags, specially marked as being used for the purpose of busy waiting. We call these flags *signals*. For now, we assume that read and write operations on signals are thread-safe. Consider a simple programming language with built-in signals and with the following commands: (i) **new_signal** for creating a new unset signal, (ii) **set_signal**(x) for setting x and (iii) **await is_set**(x) for busy-waiting until x is set. Fig. 1 presents a minimal example where two threads communicate via a shared signal sig. The main thread creates the signal sig and forks a new thread that busy-waits for sig to be set. Then, the main thread sets the signal. As we assume signal operations to be thread-safe in this example, we do not have to care about potential data races. Notice that like all busy-waiting programs, this program is guaranteed to terminate only under fair thread scheduling: Indeed, it does not terminate if the main thread is never scheduled after it forks the new thread. In this paper we verify termination under fair scheduling.

```
let sig := new_signal in
fork await is_set(sig);
set_signal(sig)
```

Fig. 1: Minimal example with two threads communicating via a physical thread-safe signal.

Augmented Semantics

Obligations The only construct in our language that can lead to non-termination are busy-waiting loops of the form **await is_set**(sig). In order to prove that programs terminate it is therefore sufficient to prove that all created signals are eventually set. We use so-called *obligations* [5,6,16,19] to ensure this. These are *ghost resources* [13], i.e., resources that do not exist during runtime and can hence not influence a program's runtime behaviour. They carry, however, information relevant to the program's verification. Generally, holding an obligation requires a thread to discharge it by performing a certain action. For instance, when the main thread in our example creates signal sig, it simultaneously creates an obligation to set it. The only way to discharge this obligation is to set sig.

We denote thread IDs by θ and describe which obligations a thread θ holds by bundling them into an obligations chunk $\theta.\mathsf{obs}(O)$, where O is a multiset of signals. We denote multisets by double braces $\{\!\{\ldots\}\!\}$ and multiset union by $\uplus$. Each occurrence of a signal s in O corresponds to an obligation by thread θ to set s. Consequently, $\theta.\mathsf{obs}(\emptyset)$ asserts that thread θ does not hold any obligations.

Augmented Semantics In the *real* semantics of the programming language we consider here, ghost resources such as obligations do not exist during runtime. To prove termination, we consider an *augmented* version of it that keeps track of ghost resources during runtime. In this semantics, we maintain the invariant that every thread holds exactly one obs chunk. That is, for every running thread θ, our heap contains a unique heap cell $\theta.\mathsf{obs}$ that stores the thread's bag of obligations. Further, we let a thread get stuck if it tries to finish while it still holds undischarged obligations. Note that we use the term *finish* to refer to thread-local behaviour while we write *termination* to refer to program-global behaviour, i.e., meaning that every thread finishes. For every augmented execution there trivially exists a corresponding execution in the real semantics.

Fig. 2 presents some of the reduction rules we use to define the augmented semantics. We use $\widehat{h}$ to refer to augmented heaps, i.e., heaps that can contain ghost resources. A reduction step has the form $\widehat{h}, c \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h}', c', T$ expresses that thread θ reduces heap $\widehat{h}$ (which is shared by all threads) and command c to heap $\widehat{h}'$ and command c'. Further, T represents the set of threads forked during this step. It is either empty or a singleton containing the new thread's ID and the command it is going to execute, i.e., $\{(\theta_f, c_f)\}$. We omit it whenever it is clear from the context that no thread is forked. Further, we denote disjoint union of sets by $\sqcup$.

Our reduction rules comply with the intuition behind obligations we outlined above. Aug-Red-NewSignal creates a new signal and simultaneously a corre-

sponding obligation. The only way to discharge it is by setting the signal using Aug-Red-SetSignal.

Aug-Red-NewSignal
$$\frac{id \notin \mathsf{ids}(\widehat{h}) \qquad L \in \mathcal{L}evs}{\widehat{h} \sqcup \{\theta.\mathsf{obs}(O)\}, \mathbf{new\_signal} \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus \{\!|(id, L)|\!\})\}, \mathsf{signal}((id, L))\}, id}$$

Aug-Red-SetSignal
$$\widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus \{\!|s|\!\})\}, \mathbf{set\_signal}(s.\mathsf{id}) \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O), \mathsf{signalSet}(s)\}, \mathsf{tt}$$

Aug-Red-Fork
$$\frac{\theta_f \notin \mathsf{thIds}(\widehat{h})}{\widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus O_f)\}, \mathbf{fork}\ c \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O), \theta_f.\mathsf{obs}(O_f)\}, \mathsf{tt}, \{(\theta_f, c)\}}$$

Aug-Red-Await
$$\frac{\theta.\mathsf{obs}(O) \in \widehat{h} \qquad \mathsf{signal}(s) \in \widehat{h} \qquad \mathsf{signalSet}(s) \notin \widehat{h} \qquad s.\mathsf{lev} \prec_{\mathsf{L}} O}{\widehat{h}, \mathbf{await\ is\_set}(s.\mathsf{id}) \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h}, \mathbf{await\ is\_set}(s.\mathsf{id})}$$

Fig. 2: Reduction rules for augmented semantics.

Forking Whenever a thread forks a new thread, it can pass some of its obligations to the newly forked thread, cf. Aug-Red-Fork. Forking a new thread with ID θ_f also allocates a new heap cell $\theta_f.\mathsf{obs}$ to store its bag of obligations. Since this is the only way to allocate a new obs heap cell, we will never run into a heap $\widehat{h} \sqcup \{\theta.\mathsf{obs}(O)\} \sqcup \{\theta.\mathsf{obs}'(O')\}$ that contains multiple obligations chunks belonging to the same thread θ. Remember that threads cannot finish while holding obligations. This prevents them from dropping obligations via dummy forks.

Levels In order to prove that a busy-waiting loop $\mathbf{await\ is\_set}(\mathsf{sig})$ terminates, we must ensure that the waiting thread does not directly or indirectly wait for itself. We could just check that it does not hold an obligation for the signal it is waiting for, but that is not sufficient as the following example demonstrates: Consider a program with two signals $\mathsf{sig}_1, \mathsf{sig}_2$ and two threads. Let one thread hold the obligation for sig_2 and execute $\mathbf{await\ is\_set}(\mathsf{sig}_1);\ \mathbf{set\_signal}(\mathsf{sig}_2)$. Likewise, let the other thread hold the obligation for sig_1 and let it execute $\mathbf{await\ is\_set}(\mathsf{sig}_2);\ \mathbf{set\_signal}(\mathsf{sig}_1)$.

To prevent such *wait cycles* modularly, we apply the usual approach [3,4,19]. For every program that we want to execute in our augmented semantics, we choose a partially ordered set of levels $\mathcal{L}evs$. Further, during every reduction step in the augmented semantics that creates a signal s, we pick a level $L \in \mathcal{L}evs$ and associate it with s. Note that much like obligations, levels do not exit during runtime in the real semantics. Signal chunks in the augmented semantics have the form $\mathsf{signal}((id, L))$ where id is the unique signal identifier returned by **new_signal**. The level assigned to any signal can be chosen freely, cf. Aug-Red-NewSignal. In practice, determining levels boils down to solving a set of constraints that reflect the dependencies. In our example, however, the choice

is trivial as it only involves a single signal. We choose $\mathcal{L}evs = \{0\}$ and 0 as level for sig and thereby get $\mathsf{signal}((\mathsf{sig}, 0))$. Generally, we denote signal tuples by $s = (id, L)$. Now we can rule out cyclic wait dependencies by only allowing a thread to busy-wait for a signal s if its level $s.\mathsf{lev}$ is smaller than the level of each held obligation, cf. Aug-Red-Await [1]. Given a bag of obligations O, we denote this by $s.\mathsf{lev} \prec_{\mathsf{L}} O$.

Proving Termination As we will explain below, the augmented semantics has no fair infinite executions. We can use this as follows to prove that a program c terminates under fair scheduling: For every fair infinite execution of c, show that we can construct a corresponding augmented execution. (This requires that each step's side conditions in the augmented semantics are satisfied. Note that we thereby prove certain properties for the real execution, like absence of cyclic wait dependencies.) As there are no fair infinite executions in the augmented semantics, we get a contradiction. It follows that c has no fair infinite executions in the real semantics.

Soundness In order to prove soundness of our approach, we must prove that there indeed are no fair infinite executions in the augmented semantics. This boils down to proving that no signal can be waited for infinitely often. Consider any program and any fair augmented execution of it. Consider the execution's *program order graph*, (i) whose nodes are the execution steps and (ii) which has an edge from a step to the next step of the same thread and to the first step of the forked thread, if it is a fork step. Notice that for each obligation created during the execution, the set of nodes corresponding to a step made by a thread while that thread holds the obligation constitutes a path that ends when the obligation is discharged. We say that this path *carries* the obligation.

It is not possible that a signal is waited for infinitely often. Indeed, suppose some signals S^{∞} are. Take $s_{\min} \in S^{\infty}$ with minimal level. Since $s_{\min}$ is never set, the path in the program order graph that carries the obligation must be infinite as well. Indeed, suppose it is finite. The final node N of the path cannot discharge the obligation without setting the signal, so it must pass the obligation on either to the next step of the same thread or to a newly forked thread. By fairness of the scheduler, both of these threads will eventually be scheduled. This contradicts N being the final node of the path.

The path carrying the obligation for $s_{\min}$ waits only for signals that are waited for finitely often. (Remember that Aug-Red-Await requires the signal waited for to be of a lower level than all held obligations, i.e., a lower level than that of $s_{\min}$.) It is therefore a finite path. A contradiction.

Notice that the above argument relies on the property that every non-empty set of levels has a minimal element. For this reason, for termination verification we require that $\mathcal{L}evs$ is not just partially ordered, but also well-founded.

[1] For simplicity, our augmented semantics assumes that the level order and the level associated with any object remains fixed for the entire execution. However, following the approach presented in [18], it would be sound to add a step rule that allows a thread to change the level of an object it has exclusive access to (cf. § 2.2).

Program Logic

Directly using the augmented semantics to prove that our example program terminates is cumbersome. In the following, we present a separation logic that simplifies this task.

Safety We call a program c *safe* under a (partial) heap $\widehat{h}$ if it provides all the resources necessary such that both c and any threads it forks can execute without getting stuck in the augmented semantics. (This depends on the angelic choices.) We denote this by $\mathsf{safe}(\widehat{h}, c)$ [33] [2].

Consider a program c that is safe under an augmented heap $\widehat{h}$. Let h be the real heap that matches $\widehat{h}$ apart from the ghost resources. Then, for every real execution that starts with h we can construct a corresponding augmented execution.

Specifications We use Hoare triples $\{A\}\ c\ \{\lambda r.\, B(r)\}$ [8] to specify the behaviour of a program c. Such a triple expresses the following: Consider any evaluation context E, such that for every return value v, running $E[v]$ from a state that satisfies $B(v)$ is safe. Then, running $E[c]$ from a state that satisfies A is safe.

Proof System We define a proof relation $\vdash$ which ensures that whenever we can prove $\vdash \{A\}\ c\ \{\lambda r.\, B(r)\}$, then c complies with the specification $\{A\}\ c\ \{\lambda r.\, B(r)\}$. Fig. 3b presents some of the proof rules we use to define $\vdash$. As we evolve our setting throughout this section, we also adapt our proof rules. Rules that will be changed later are marked with a prime in their name. The full set of rules is presented in the extended version of this paper [28]. Our proof rules PR-SetSignal' and PR-Await' are similar to the rules for sending and receiving on a channel presented in [19].

Notice how the proof rules enforce the side-conditions of the augmented semantics. Hence, all we have to do to prove that a program c terminates is to prove that every thread eventually discharges all its obligations. That is, we have to prove $\vdash \{\mathsf{obs}(\emptyset)\}\ c\ \{\mathsf{obs}(\emptyset)\}$. Fig. 3a illustrates how we can apply our rules to verify that our minimal example terminates.

2.2 Non-Thread-Safe Physical Signals

As a step towards supporting waiting for arbitrary conditions over shared data structures, including non-thread-safe ones, we now move to non-thread-safe signals. For simplicity, in this paper we consider programs that use mutexes to synchronize concurrent accesses to shared data structures. (Our ideas apply equally to programs that use other constructs, such as atomic machine instructions.) Fig. 4 presents our updated example.

[2] For a formal definition see this paper's extended version [28] and the technical report [29].

$\{\mathsf{obs}(\emptyset)\}$	
let sig := **new_signal in**	PR-NewSignal' with $L = 0$
$\{\mathsf{obs}(\{\!\{(\mathsf{sig}, 0)\}\!\}) * \mathsf{signal}((\mathsf{sig}, 0))\}$	$s := (\mathsf{sig}, 0)$
fork $(\{\mathsf{obs}(\emptyset) * \mathsf{signal}(s)\}$	
await is_set(sig)	$s.\mathsf{lev} = 0 \prec_{\mathsf{L}} \emptyset$
$\{\mathsf{obs}(\emptyset) * \mathsf{signal}(s)\})$;	
$\{\mathsf{obs}(\{\!\{s\}\!\})\}$	
set_signal(sig)	
$\{\mathsf{obs}(\emptyset)\}$	

(a) Proof outline for program from Fig. 1. Applied proof rule marked in purple. Abbreviation marked in brown. General hint marked in red.

$$\text{PR-NewSignal'}\quad \frac{L \in \mathcal{L}evs}{\vdash \{\mathsf{obs}(O)\}\ \textbf{new\_signal}\ \{\lambda r.\, \mathsf{obs}(O \uplus \{\!\{(r, L)\}\!\}) * \mathsf{signal}((r, L))\}}$$

$$\text{PR-SetSignal'}\quad \vdash \{\mathsf{obs}(O \uplus \{\!\{s\}\!\})\}\ \textbf{set\_signal}(s.\mathsf{id})\ \{\mathsf{obs}(O)\}$$

$$\text{PR-Fork'}\quad \frac{\vdash \{\mathsf{obs}(O_f) * A\}\ c\ \{\mathsf{obs}(\emptyset) * B\}}{\vdash \{\mathsf{obs}(O_m \uplus O_f) * A\}\ \textbf{fork}\ c\ \{\mathsf{obs}(O_m)\}}$$

$$\text{PR-Await'}\quad \frac{s.\mathsf{lev} \prec_{\mathsf{L}} O}{\vdash \{\mathsf{obs}(O) * \mathsf{signal}(s)\}\ \textbf{await is\_set}(s.\mathsf{id})\ \{\mathsf{obs}(O) * \mathsf{signal}(s)\}}$$

$$\text{PR-Let}\quad \frac{\vdash \{A\}\ c\ \{\lambda r.\, C(r)\} \qquad \forall v.\ \vdash \{C(v)\}\ c'[v/x]\ \{B\}}{\vdash \{A\}\ \textbf{let}\ x := c\ \textbf{in}\ c'\ \{B\}}$$

(b) Proof rules. Rules only used in this section marked with '.

Fig. 3: Verifying termination of minimal example with physical thread-safe signal.

```
let sig := new_signal in
let mut := new_mutex in
fork with mut await is_set(sig);
acquire mut;
set_signal(sig);
release mut
```

(a) Code.

```
with mut await c := (while acquire mut;
                         let r := c in
                         release mut;
                         ¬r
                     do skip)
```

(b) Syntactic sugar. r not free in mut.

Fig. 4: Minimal example with two threads communicating via a physical non-thread-safe signal protected by a mutex.

As signal sig is no longer thread-safe, the two threads can no longer use it directly to communicate. Instead, we have to synchronize accesses to avoid data races. Hence, we protect the signal by a mutex mut created by the main thread. In each iteration, the forked thread acquires the mutex, checks whether sig has been set and releases it again. After forking, the main thread acquires the mutex, sets the signal and releases it again.

Exposing Signal Values Signals are specially marked heap cells storing boolean values. We make this explicit by extending our signal chunks from $\mathsf{signal}(s)$ to $\mathsf{signal}(s, b)$ where b is the current value of s and by updating our proof rules accordingly. Upon creation, signals are unset. Hence, creating a signal sig now spawns an *unset* signal chunk $\mathsf{signal}((\mathsf{sig}, L), \mathsf{False})$ for some freely chosen level L and an obligation for (sig, L), cf. PR-NewSignal". We present our new proof rules in Fig. 6 and demonstrate their application in Fig. 5.

$\{\mathsf{obs}(\emptyset)\}$	
let sig := **new_signal in**	PR-NewSignal" with $L = 1$
$\{\mathsf{obs}(\{\!\{(\mathsf{sig}, 1)\}\!\}) * \mathsf{signal}((\mathsf{sig}, 1), \mathsf{False})\}$	PR-ViewShift & VS-SemImp
$\{\mathsf{obs}(\{\!\{(\mathsf{sig}, 1)\}\!\}) * \exists b.\ \mathsf{signal}((\mathsf{sig}, 1), b)\}$	$s := (\mathsf{sig}, 1),\ P := \exists b.\ \mathsf{signal}(s, b)$
let mut := **new_mutex in**	PR-NewMutex" with $L = 0$
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m, P)\}$	PR-ViewShift
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m, P) * \mathsf{mutex}(m, P)\}$	& VS-CloneMut"
fork $(\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m, P)\}$	
with m **await**	$m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} \emptyset$
$\{\mathsf{obs}(\{\!\{m\}\!\}) * P\}$	PR-Exists
$\forall b.\ \{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{signal}(s, b)\}$	
is_set(sig)	
$\{\lambda r.\ \mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{signal}(s, b) \wedge r = b\}$	PR-ViewShift & VS-SemImp
$\left\{\begin{array}{l}\lambda r.\ \mathsf{obs}(\{\!\{m\}\!\}) \\ \quad * \mathsf{if}\ r\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{signal}(s, \mathsf{False})\end{array}\right\}$	
$\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m, P)\}$	PR-ViewShift & VS-SemImp
$\{\mathsf{obs}(\emptyset)\}$);	
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m, P)\}$	
acquire mut;	$m.\mathsf{lev} = 0 < 1 = s.\mathsf{lev}$
$\{\mathsf{obs}(\{\!\{s, m\}\!\}) * \mathsf{locked}(m, P) * \exists b.\ \mathsf{signal}(s, b)\}$	PR-Exists
$\forall b.\ \{\mathsf{obs}(\{\!\{s, m\}\!\}) * \mathsf{locked}(m, P) * \mathsf{signal}(s, b)\}$	
set_signal(sig);	
$\{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{locked}(m, P) * \mathsf{signal}(s, \mathsf{True})\}$	PR-ViewShift & VS-SemImp
$\{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{locked}(m, P) * P\}$	
release mut	
$\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m, P)\}$	PR-ViewShift & VS-SemImp
$\{\mathsf{obs}(\emptyset)\}$	

Fig. 5: Proof outline for program 4, verifying termination with mutexes & non-thread safe signals. Applied proof and view shift rules marked in purple. Abbreviations marked in brown. General hints marked in red.

Data Races As read and write operations on signals are no longer thread-safe, our logic has to ensure that two threads never try to access sig at the same time. Hence, in our logic possession of a signal chunk $\mathsf{signal}(s, b)$ expresses (temporary) *exclusive ownership* of s. Further, our logic requires threads to own any signal they are trying to access. Specifically, when a thread wants to set sig, it must hold a chunk of the form $\mathsf{signal}((\mathsf{sig}, L), b)$, cf. PR-SetSignal". The same holds for reading a signal's value, cf. PR-IsSignalSet". Note that signal chunks are not duplicable and only created upon creation of the signal they refer to. Therefore, holding a signal chunk for sig indeed guarantees that the holding thread has the exclusive right to access sig (while holding the signal chunk).

Synchronization & Lock Invariants After the main thread creates sig, it exclusively owns the signal. The main thread can transfer ownership of this resource during forking, cf. PR-Fork', and thereby allow the forked thread to busy-wait for sig. This would, however, leave the main thread without any permission to set the signal and thereby discharge its obligation.

We use mutexes to let multiple threads share ownership of a common set of resources in a synchronized fashion. Every mutex is associated with a *lock invariant* P, an assertion chosen by the proof author that specifies which resources the mutex protects. In our example, we want both threads to share sig. To reflect the fact that the signal's value changes over time, we choose a lock invariant that abstracts over its concrete value. We choose $P := \exists b.\ \mathsf{signal}((\mathsf{sig}, L), b)$. Let us ignore the chosen signal level L for now. Creating the mutex mut consumes this lock invariant and binds it to mut by creating a mutex chunk $\mathsf{mutex}((\mathsf{mut}, \dots), P)$, cf. PR-NewMutex". Thereby, the main thread loses access to sig. The only way to regain access is by acquiring mut, cf. PR-Acquire". Once the thread releases mut, it again loses access to all resources protected by the mutex, cf. PR-Release".

Deadlocks We have to ensure that any acquired mutex is eventually released, again. Hence, acquiring a mutex spawns a release obligation for this mutex and the only way to discharge this obligation is indeed by releasing it, cf. PR-Acquire" and PR-Release".

Any attempt to acquire a mutex will block until the mutex becomes available. In order to prove that our program terminates, we have to prove that it does not get stuck during an acquisition attempt. To prevent wait cycles involving mutexes, we require the proof author to associate every mutex as well (just like signals) with a level L. This level can be freely chosen during the mutex' creation, cf. PR-NewMutex". Mutex chunks therefore have the form $\mathsf{mutex}((\ell, L), P)$ where ℓ is the heap location the mutex is stored at. Their only purpose is to record the level and lock invariant a mutex is associated with. Hence, these chunks can be freely duplicated as we will see later. Generally, we denote mutex tuples by $m = (\ell, L)$. We only allow to acquire a mutex if its level is lower than the level of each held obligation, cf. PR-Acquire". This also prevents any thread from attempting to acquire mutexes twice, e.g., **acquire** mut; **acquire** mut or **with** mut **await acquire** mut.

PR-NewSignal"
$$\frac{L \in \mathcal{L}evs}{\vdash \{\mathsf{obs}(O)\}\ \mathbf{new\_signal}\ \{\lambda id.\, \mathsf{obs}(O \uplus \{\!\{(id, L)\}\!\}) * \mathsf{signal}((id, L), \mathsf{False})\}}$$

PR-SetSignal"
$$\vdash \{\mathsf{obs}(O \uplus \{\!\{s\}\!\}) * \mathsf{signal}(s, \_)\}\ \mathbf{set\_signal}(s.\mathsf{id})\ \{\mathsf{obs}(O) * \mathsf{signal}(s, \mathsf{True})\}$$

PR-IsSignalSet"
$$\vdash \{\mathsf{signal}(s, b)\}\ \mathbf{is\_set}(s.\mathsf{id})\ \{\lambda r.\, \mathsf{signal}(s, b) \wedge r = b\}$$

PR-Await"
$$\frac{m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} O \qquad \mathsf{signal}(s, \mathsf{False}) * R \Rrightarrow P \qquad \vdash \{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * P\}\ c\ \{\lambda r.\, \mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{if}\ r\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{signal}(s, \mathsf{False}) * R\}}{\vdash \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\}\ \mathbf{with}\ m.\mathsf{loc}\ \mathbf{await}\ c\ \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\}}$$

(a) Signals & busy waiting.

PR-NewMutex"
$$\frac{L \in \mathcal{L}evs}{\vdash \{P\}\ \mathbf{new\_mutex}\ \{\lambda \ell.\, \mathsf{mutex}((\ell, L), P)\}}$$

PR-Acquire"
$$\{\mathsf{obs}(O) * \mathsf{mutex}(m, P) \wedge m.\mathsf{lev} \prec_{\mathsf{L}} O\} \vdash \mathbf{acquire}\ m.\mathsf{loc}\ \{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{locked}(m, P) * P\}$$

PR-Release"
$$\{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{locked}(m, P) * P\} \vdash \mathbf{release}\ m.\mathsf{loc}\ \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\}$$

(b) Mutexes.

PR-Frame
$$\frac{\vdash \{A\}\ c\ \{B\}}{\vdash \{A * F\}\ c\ \{B * F\}}$$

PR-Exists
$$\frac{\forall a \in A.\ \vdash \{a\}\ c\ \{B\}}{\vdash \{\bigvee A\}\ c\ \{B\}}$$

PR-Fork
$$\frac{\vdash \{\mathsf{obs}(O_f) * A\}\ c\ \{\mathsf{obs}(\emptyset)\}}{\vdash \{\mathsf{obs}(O_m \uplus O_f) * A\}\ \mathbf{fork}\ c\ \{\mathsf{obs}(O_m)\}}$$

PR-ViewShift
$$\frac{A \Rrightarrow A' \qquad \vdash \{A'\}\ c\ \{B'\} \qquad B' \Rrightarrow B}{\vdash \{A\}\ c\ \{B\}}$$

(c) Standard rules.

VS-SemImp
$$\frac{\forall H.\ \mathsf{consistent}_{\mathsf{lh}}(H) \wedge H \vDash_{\mathsf{A}} A \Rightarrow H \vDash_{\mathsf{A}} B}{A \Rrightarrow B}$$

VS-Trans
$$\frac{A \Rrightarrow C \qquad C \Rrightarrow B}{A \Rrightarrow B}$$

VS-CloneMut"
$$\mathsf{mutex}(m, P) \Rrightarrow \mathsf{mutex}(m, P) * \mathsf{mutex}(m, P)$$

(d) View shifts.

Fig. 6: Proof rules & view shift rules for mutexes & non-thread safe signals. Rules only used in this section marked with ".

View Shifts When verifying a program, it can be necessary to reformulate the proof state and to draw semantic conclusions. To allow this we introduce a so-called *view shift* relation $\Rrightarrow$ [14]. By applying proof rule PR-ViewShift and VS-SemImp we can strengthen the precondition and weaken the postcondition. In our example, we use this to convert the unset signal chunk into the lock invariant which abstracts over the signal's value, i.e., $\mathsf{signal}(s, \mathsf{False}) \Rrightarrow \exists b.\ \mathsf{signal}(s, b)$.

The logic we present in this work is an intuitionistic separation logic that allows us to drop chunks. [3] This allows us to simplify the postcondition of our fork proof rule's premise from $\mathsf{obs}(\emptyset) * B$ to $\mathsf{obs}(\emptyset)$, cf. PR-Fork, and drop all unneeded chunks via a semantic implication $\mathsf{obs}(\emptyset) * B \Rrightarrow \mathsf{obs}(\emptyset)$.

We also allow to clone mutex chunks via view shifts, cf. VS-CloneMut". In our example, this is necessary to inform both threads which level and lock invariant mutex mut is associated with. That is, the main thread clones the mutex chunk $\mathsf{mutex}(m, P)$ and passes one chunk on when it forks the busy-waiting thread.

In § 2.4 we extend our view shift relation and revisit our interpretation of what a view shift expresses. The full set of rules we use to define $\Rrightarrow$ is presented in the extended version of this paper [28].

Busy Waiting In the approach presented in this paper, for simplicity we only support busy-waiting loops of the form **with** mut **await** c, which is syntactic sugar for **while acquire** mut; **let** $r := c$ **in release** mut; $\neg r$ **do skip** where r denotes a fresh variable. [4] In each iteration, the loop tries to acquire mut, executes c, releases mut again and lets the result returned by c determine whether the loop continues. Such loops can fail to terminate for two reasons: (i) Acquiring mut can get stuck and (ii) the loop could diverge.

We prevent the loop from getting stuck by requiring mut's level to be lower than the level of each held obligation, cf. PR-Await". Further, we enforce termination by requiring the loop to wait for a signal. That is, when verifying a busy-waiting loop using our approach, the proof author must choose a fixed signal and prove that this signal remains unset at the end of every non-finishing iteration. This way, we can prove that the loop terminates by proving that every signal is eventually set, just as in § 2.1. And just as before, our logic requires the level of the waited-for signal to be lower than the level of each held obligation.

Acquiring the mutex in every iteration makes the lock invariant available during the verification of the loop body c. This lock invariant has to be restored at the end of the iteration such that it can be consumed during the mutex's release. PR-Await" allows for an additional view shift to restore the invariant. In our example, we end our busy-waiting loop's non-finishing iterations with the

[3] This allows a thread to drop its obligations chunk $\mathsf{obs}(O)$. Note, however, that by dropping this chunk the thread does not drop its obligations, but only its ability to show what its obligations are. In particular the thread would be unable to present an empty obligations chunk upon termination.

[4] As we discuss in § 5, in the technical report accompanying this paper we present a more general logic that imposes no such syntactic restrictions.

assertion $\mathsf{signal}(s, \mathsf{False})$. We use a semantic implication view shift to convert the signal chunk into the mutex invariant $\exists b.\ \mathsf{signal}(s, b)$.

Choosing Levels In our example, we have to assign levels to the mutex mut and to the signal sig. Our proof rules for mutex acquisition and busy waiting impose some restrictions on the levels of the involved mutexes and signals. By analysing the corresponding rule applications that occur in our proof, we can derive which constraints our level choice must comply with. Our example's verification involves one application of PR-Acquire" and one application of PR-Await": (i) Our main thread tries to acquire mut while holding an obligation to set sig. (ii) The forked thread busy-waits for sig while not holding any obligations. Our assignment of levels must therefore satisfy the single constraint $m.\mathsf{lev} <_{\mathsf{L}} s.\mathsf{lev}$. So, we choose $\mathcal{L}evs = \{0, 1\}$, $m.\mathsf{lev} = 0$ and $s.\mathsf{lev} = 1$.

2.3 Arbitrary Data Structures

The proof rules we introduced in § 2.2 allow us to verify programs busy-waiting for arbitrary conditions over arbitrary shared data structures as follows: For every condition C the program waits for, the proof author inserts a signal s into the program. They ensure that s is set at the same time the program establishes C and prove an invariant stating that the signal's value expresses whether C holds. Then, the waiting thread can use s to wait for C. We illustrate this here for the simplest case of setting a single heap cell in Fig. 7a.

```
let x := cons(0) in
let mut := new_mutex in
fork with mut await [x] = 1;
acquire mut;
[x] := 1;
release mut
```

(a) Example program with busy waiting for heap cell x to be set.

```
let x := cons(0) in
let sig := new_signal in
let mut := new_mutex in
fork with mut await [x] = 1;
acquire mut;
[x] := 1;
set_signal(sig);
release mut
```

(b) Example program 7a with additional signal sig inserted, marked in green. sig and x are kept in sync.

$$[e] = e' \ := \ (\textbf{let}\ r := [e]\ \textbf{in}\ r = e')$$

(c) Syntactic sugar. r free in e'.

Fig. 7: Minimal example illustrating busy waiting for condition over heap cell.

The program involves three new non-thread-safe commands: (i) $\mathbf{cons}(v)$ for allocating a new heap cell and initializing it with value v, (ii) $[\ell] := v$ for assigning value v to heap location ℓ, (iii) $[\ell]$ for reading the value stored in heap location ℓ. We use $[\ell] = v$ as syntactic sugar for $\textbf{let}\ r := [e]\ \textbf{in}\ r = e'$.

In our example, the main thread allocates x, initializes it with the value 0 and protects it using mutex mut. It forks a new thread busy-waiting for x to be set. Afterwards, the main thread sets x. As explained above, we verify the program by inserting a signal sig that reflects whether x has been set, yet. Fig. 7b presents the resulting code. The main thread creates the signal and sets it when it sets x.

$\{\mathsf{obs}(\emptyset)\}$
let x := **cons**(0) **in**
$\{\mathsf{obs}(\emptyset) * \mathsf{x} \mapsto 0\}$
let sig := **new_signal in** — PR-NewSignal" with $L = 1$
let mut := **new_mutex in** — PR-NewMutex" with $L = 0$
$s := (\mathsf{sig}, 1),\ m := (\mathsf{mut}, 0)$
$P := \exists v.\ \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)$
$\{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{mutex}(m, P) * \mathsf{mutex}(m, P)\}$
fork ($\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m, P)\}$
  with m **await** — $m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} \emptyset$
    $\{\mathsf{obs}(\{\!|m|\!\}) * P\}$
    $\forall v.\ \{\mathsf{obs}(\{\!|m|\!\}) * \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\}$
      $[\mathsf{x}] = 1$
      $\left\{\begin{array}{l} \lambda r.\, \mathsf{obs}(\{\!|m|\!\}) \\ \quad * \text{ if } r \text{ then } P \\ \quad\ \ \text{else } \mathsf{x} \mapsto v \wedge v \neq 1 * \mathsf{signal}(s, \mathsf{False}) \end{array}\right\}$
  $\{\mathsf{obs}(\emptyset)\}$);
$\{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{mutex}(m, P)\}$
acquire mut; — $m.\mathsf{lev} = 0 < 1 = s.\mathsf{lev}$
$\forall v.\ \{\mathsf{obs}(\{\!|s, m|\!\}) * \mathsf{locked}(m, P) * \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\}$
  $[\mathsf{x}] := 1$;
  $\{\mathsf{obs}(\{\!|s, m|\!\}) * \mathsf{locked}(m, P) * \mathsf{x} \mapsto 1 * \mathsf{signal}(s, v = 1)\}$
  set_signal(sig);
  $\{\mathsf{obs}(\{\!|m|\!\}) * \mathsf{locked}(m, P) * \mathsf{x} \mapsto 1 * \mathsf{signal}(s, \mathsf{True})\}$
  release mut
  $\{\mathsf{obs}(\emptyset)\}$

(a) Proof outline for program 7b. Applied proof rules marked in purple. Abbreviations marked in brown. General hints marked in red.

PR-Cons
$\vdash \{\mathsf{True}\}\ \mathbf{cons}(v)\ \{\lambda \ell.\, \ell \mapsto v\}$

PR-AssignToHeap
$\vdash \{\ell \mapsto \_\}\ [\ell] := v\ \{\ell \mapsto v\}$

PR-ReadHeapLoc"'
$\vdash \{\ell \mapsto v\}\ [\ell]\ \{\lambda r.\, r = v * \ell \mapsto v\}$

PR-Exp
$$\frac{[\![e]\!] \in \mathit{Values}}{\vdash \{\mathsf{True}\}\ e\ \{\lambda r.\, r = [\![e]\!]\}}$$

(b) Proof rules. Evaluation function $[\![\cdot]\!]$. Rules only used in this section marked with "'.

Fig. 8: Verifying termination of busy waiting for condition over heap cell.

Heap Cells Verifying this example does not conceptually differ from the example we presented in § 2.2. Fig. 8b presents the new proof rules we need and Fig. 8a sketches our example's verification. As with non-thread-safe signals, we have

to prevent multiple threads from trying to access x at the same time in order to prevent data races. For this we use so-called *points-to* chunks [24, 31]. They have the form $\ell \mapsto v$ and express that heap location ℓ stores the value v. When a thread holds such a chunk, it exclusively owns the right to access heap location ℓ.

Heap locations are unique and the only way to create a new points-to chunk is to allocate and initialize a new heap cell via $\mathbf{cons}(v)$, cf. PR-Cons. Hence, there will never be two points-to chunks involving the same heap location. In order to read or write a heap cell via $[\ell]$ or $[\ell] := e$, the acting thread must first acquire possession of the corresponding points-to chunk, cf. PR-AssignToHeap and PR-ReadHeapLoc"'.

Relating Signals to Conditions In our example, the forked thread busy-waits for x to be set while our proof rules require us to justify each iteration by showing an unset signal. That is, we must prove an invariant stating that the value of x matches sig. As this invariant must be shared between both threads, we encode it in the lock invariant: $P := \exists v.\ \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)$. This does not only allow both threads to share the heap cell and the signal but it also automatically enforces that they maintain the invariant whenever they acquire and release the mutex.

2.4 Signal Erasure

In the program from Fig. 7b signal sig is never read and does hence not influence the waiting thread's runtime behaviour. Therefore, we can verify the original program presented in Fig. 7a by erasing the physical signal and treating it as ghost code.

Ghost Signals Central aspects of the proof sketch we presented in Fig. 8a are that (i) the main thread was obliged to set sig and that (ii) the value of sig reflected whether x was already set. *Ghost signals* allow us to keep this information but at the same to remove the physical signals from the code. Ghost signals are essentially identical to the physical non-thread-safe signals we used so far. However, as ghost resources they cannot influence the program's runtime behaviour. They merely carry information we can use during the verification process.

View Shifts Revisited We implement ghost signals by extending our view shift relation. In particular, we introduce two new view shift rules: VS-NewSignal and VS-SetSignal presented in Fig. 9b. The former creates a new unset signal and simultaneously spawns an obligation to set it. The latter can be used to set a signal and thereby discharge a corresponding obligation. We say that these rules change the *ghost state* and therefore call their application a *ghost proof step*. With this extension, a view shift $A \Rrightarrow B$ expresses that we can reach postcondition B from precondition A by (i) drawing semantic conclusions or by (ii) manipulating the ghost state. In Fig. 9a we use ghost signals to verify the program from 7a.

Note that lifting signals to the verification level does not affect the soundness of our approach. The argument we presented in § 2.1 still holds. We formalize our logic and provide a formal soundness proof in the extended version of this paper [28] and in the technical report [29]. The latter contains a more general version of the presented logic that (i) is not restricted to busy-waiting loops of the form **with** mut **await** c and that (ii) is easier to integrate into existing tools like VeriFast [12], as explained in § 5.

$\{\mathsf{obs}(\emptyset)\}$
let x := **cons**(0) **in**
$\{\mathsf{obs}(\emptyset) * \mathsf{x} \mapsto 0\}$
new_ghost_signal; — VS-NewSignal with $L = 1$.
$\{\exists \mathsf{sig}.\ \mathsf{obs}(\{\!|(\mathsf{sig}, 1)|\!\}) * \mathsf{x} \mapsto 0 * \mathsf{signal}((\mathsf{sig}, 1), \mathsf{False})\}$ — $s := (\mathsf{sig}, 1)$
$\forall \mathsf{sig}.\ \{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{x} \mapsto 0 * \mathsf{signal}(s, \mathsf{False})\}$ — $P := \exists v.\ \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)$
  let mut := **new_mutex in** — PR-NewMutex" with $L = 0$
  $\left\{\begin{array}{l}\mathsf{obs}(\{\!|s|\!\}) * \mathsf{mutex}((\mathsf{mut}, 0), P)\\ * \mathsf{mutex}((\mathsf{mut}, 0), P)\end{array}\right\}$ — $m := (\mathsf{mut}, 0)$
  fork $(\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m, P)\}$
    with m **await** — $m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} \emptyset$
      $\{\mathsf{obs}(\{\!|m|\!\}) * P\}$
      $\forall v.\ \{\mathsf{obs}(\{\!|m|\!\}) * \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\}$
        $[\mathsf{x}] = 1$
        $\left\{\begin{array}{l}\lambda r.\ \mathsf{obs}(\{\!|m|\!\}) *\\ \quad \mathsf{if}\ r\ \mathsf{then}\ P\\ \quad \mathsf{else}\ \mathsf{x} \mapsto v \wedge v \neq 1 * \mathsf{signal}(s, \mathsf{False})\end{array}\right\}$
    $\{\mathsf{obs}(\emptyset)\}$);
  $\{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{mutex}(m, P)\}$
  acquire mut; — $m.\mathsf{lev} = 0 < 1 = s.\mathsf{lev}$
  $\forall v.\ \left\{\begin{array}{l}\mathsf{obs}(\{\!|s, m|\!\}) * \mathsf{locked}(m, P)\\ * \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\end{array}\right\}$
    $[\mathsf{x}] := 1$;
    set_ghost_signal(s);
    $\left\{\begin{array}{l}\mathsf{obs}(\{\!|m|\!\}) * \mathsf{locked}(m, P)\\ * \mathsf{x} \mapsto 1 * \mathsf{signal}(s, \mathsf{True})\end{array}\right\}$
    release mut
    $\{\mathsf{obs}(\emptyset)\}$

(a) Proof outline for the program presented in Fig. 7a. Auxiliary commands hinting at view shifts and general hints marked in red. Applied proof and view shift rules marked in purple. Abbreviations marked in brown.

VS-NewSignal

$$\frac{L \in \mathcal{L}evs}{\mathsf{obs}(O) \Rrightarrow \exists id.\ \mathsf{obs}(O \uplus \{\!|(id, L)|\!\}) * \mathsf{signal}((id, L), \mathsf{False})}$$

VS-SetSignal

$$\mathsf{obs}(O \uplus \{\!|s|\!\}) * \mathsf{signal}(s, \_) \Rrightarrow \mathsf{obs}(O) * \mathsf{signal}(s, \mathsf{True})$$

(b) Proof rules.

Fig. 9: Verifying termination with ghost signals.

3 A Realistic Example

To demonstrate the expressiveness of the presented verification approach, we verified the termination of the program presented in Fig. 10a. It involves two threads, a consumer and a producer, communicating via a shared bounded FIFO with a maximal capacity of 10. The producer enqueues numbers 100, ..., 1 into the FIFO and the consumer dequeues those. Whenever the queue is full, the producer busy-waits for the consumer to dequeue an element. Likewise, whenever the queue is empty, the consumer busy-waits for the producer to enqueue the next element. Each thread's finishing depends on the other thread's productivity. This is, however, no cyclic dependency. For instance, in order to prove that the producer eventually pushes number i into the queue, we only need to rely on the consumer to pop $i + 10$. A similar property holds for the consumer.

```
alloc_ghost_signal_IDs(id^i_pop, id^i_push)   for 1 ≤ i ≤ 100;
L^i_pop := 102 − i,  L^i_push := 101 − i,  s^i_x := (id^i_x, L^i_x)  for 1 ≤ i ≤ 100
init_ghost_signals(s^100_pop, s^100_push);
{obs({|s^100_pop, s^100_push|}) ∗ ...}
let fifo10 := cons(nil) in let mut := new_mutex in
let cp := cons(100) in let cc := cons(100) in
fork (while (                                  cp decreases in each iteration.
        with mut await (                       Busy-wait for fifo10 not being full.
          {obs({|s^cp_push, (mut, 0)|}) ∗ ...}  → Wait for consumer to pop.
          let f := [fifo10] in
          if size(f) < 10 then (               If fifo10 not full, push next element.
            let c := [cp] in [fifo10] := f ·⟨c⟩;  [cp] := c − 1;
            set_ghost_signal(s^c_push);
            if c − 1 ≠ 0 then init_ghost_signal(s^{c−1}_push));
          size(f) ≠ 10);                       if size(f) = 10 then wait for s^{cp+10}_pop
        [cp] ≠ 0)                              L^{cp+10}_pop = 92 − cp < 101 − cp = L^cp_push
      do skip);
while (                                        cc decreases in each iteration.
  with mut await (                             Busy-wait for fifo10 not being empty.
    {obs({|s^cc_pop, (mut, 0)|}) ∗ ...}        → Wait for producer to push.
    let f := [fifo10] in
    if size(f) > 0 then (                      If fifo10 not empty, pop next element.
      let c := [cc] in [fifo10] := tail(f);  [cc] := c − 1;
      set_ghost_signal(s^c_pop);
      if c − 1 ≠ 0 then init_ghost_signal(s^{c−1}_pop));
    size(f) > 0);                              if size(f) = 0 then wait for s^cc_push
  [cc] ≠ 0)                                    L^cc_push = 101 − cc < 102 − cc = L^cc_push
do skip);
```

(a) Example program with two threads communicating via a shared bounded FIFO with maximal size 10. Auxiliary commands hinting at view shifts and general hints marked in red. Abbreviations marked in brown. Hints on proof state marked in blue.

VS-AllocSigID
$\mathsf{True} \Rrightarrow \exists id.\ \mathsf{uninitSig}(id)$

VS-SigInit
$\mathsf{obs}(O) * \mathsf{uninitSig}(id) \Rrightarrow \mathsf{obs}(O \uplus \{|(id, L)|\}) * \mathsf{signal}((id, L), \mathsf{False})$

(b) Fine-grained view shift rules for signal creation.

Fig. 10: Realistic example program.

Fine-Tuning Signal Creation To simplify complex proofs involving many signals we refine the process of creating a new ghost signal. For simplicity, we combined the allocation of a new signal ID and its association with a level and a boolean in one step. For some proofs, such as the one we outline in this section, it can be helpful to fix the IDs of all signals that will be created throughout the proof already at the beginning. To realize this, we replace view shift rule VS-NewSignal by the rules presented in Fig. 10b and adapt our signal chunks accordingly. With these more fine-grained view shifts, we start by allocating a signal ID, cf. VS-AllocSigID. Thereby we obtain an *uninitialized* signal $\mathsf{uninitSig}(id)$ that is not associated with any level or boolean, yet. Also, allocating a signal ID does not create any obligation because threads can only wait for *initialized* (and unset) signals. When we initialize a signal, we bind its already allocated ID to a level of our choice and associate the signal with False, cf. VS-SigInit. This creates an obligation to set the signal.

Loops & Signals In our program, both threads have a local counter initially set to 100 and run a nested loop. The outer loops are controlled by their thread's counter, which is decreased in each iteration until it reaches 0 and the loop stops. For such loops, we introduce a conventional proof rule for total correctness of loops, cf. this paper's extended version [28]. Verifying termination of the inner loops is a bit more tricky and requires the use of ghost signals.

So far, we had to fix a single signal for the verification of every **await** loop. We can relax this restriction to considering a finite set of signals the loop may wait for, cf. PR-Await presented in [28]. Apart from being a generalisation, this rule does not differ from PR-Await" introduced in § 2.2.

Initially, we allocate 200 signal IDs $id^{100}_{\text{push}}, \ldots, id^{1}_{\text{push}}, id^{100}_{\text{pop}}, \ldots, id^{1}_{\text{pop}}$. We are going to ensure that always at most one push signal and at most one pop signal are initialized and unset. The producer and consumer are going to hold the obligation for the push and pop signal, respectively. The producer will hold the obligation for s^i_{push} while i is the next number to be pushed into the FIFO and it will set s^i_{push} when it pushes the number i into the FIFO. Meanwhile, the consumer will use s^i_{push} to wait for the number i to arrive in the queue when it is empty. Similarly, the consumer will hold the obligation for s^i_{pop} while number i is the next number to be popped from the FIFO and will set s^i_{pop} when it pops the number i. The producer uses s^i_{pop} to wait for the consumer to pop i from the queue when it is full. At any time, we let the mutex mut protect the two active signals and thereby make them accessible to both threads.

Choosing the Levels Note that we ignored the levels so far. The producer and the consumer both acquire the mutex while holding an obligation for a signal. Hence, we choose $\mathcal{L}evs = \mathbb{N}$, $\mathsf{m.lev} = 0$ and $s.\mathsf{lev} > 0$ for every signal s. Both threads will justify iterations of their respective **await** loop by using an unset signal at the end of such an iteration. Our proof rules allow us to ignore the mutex obligation during this step. Hence, the mutex level does not interfere with the level of the unset signal. Whenever the queue is full, the producer waits for the consumer

to pop an element and whenever the queue is empty, the consumer waits for the producer to push. That is, the producer waits for s^{i+10}_{pop} while holding an obligation for s^{i}_{push} and the consumer waits for s^{i}_{push} while holding an obligation for s^{i}_{pop}. So, we have to choose the signal levels such that $s^{i+10}_{\text{pop}}.\mathsf{lev} < s^{i}_{\text{push}}.\mathsf{lev}$ and $s^{i}_{\text{push}}.\mathsf{lev} < s^{i}_{\text{pop}}.\mathsf{lev}$ hold. We solve this by choosing $s^{i}_{\text{pop}}.\mathsf{lev} = 102 - i$ and $s^{i}_{\text{push}}.\mathsf{lev} = 101 - i$.

Verifying Termination This setup suffices to verify the example program. Via the lock invariant, each thread has access to both active signals. Whenever the producer pushes a number i into the queue, it sets s^{i}_{push} which discharges the held obligation and decreases its counter. Afterwards, if $i > 1$, it uses the uninitialized signal chunk $\mathsf{uninitSig}(id^{i-1}_{\text{push}})$ to initialize $s^{i-1}_{\text{push}} = (id^{i-1}_{\text{push}}, 101 - (i-1))$ and replaces s^{i}_{push} in the lock invariant by s^{i-1}_{push} before it releases the lock. If $i = 1$, the counter reached 0 and the loop ends. In this case, the producer holds no obligation. The consumer behaves similarly. Since we proved that each thread discharged all its obligations, we proved that the program terminates. Fig. 10a illustrates the most important proof steps. We present the program's verification in full detail in the extended version of this paper [28] and in the technical report [29]. Furthermore, we encoded [27] the proof in VeriFast [12].

The number of threads in this program is fixed. However, our approach also supports the verification of programs where the number of threads is not even statically bounded. In [28] we present and verify such a program. It involves N producer and N consumer threads that communicate via a shared buffer of size 1, for a random number $N > 0$ determined during runtime.

4 Specifying Busy-Waiting Concurrent Objects

Our approach can be used to verify busy-waiting concurrent objects with respect to abstract specifications. For example, we have verified [26] the CLH lock [7] against a specification that is very similar to our proof rules for built-in mutexes shown in Fig. 6. The main difference is that it is slightly more abstract: when a lock is initialized, it is associated with a *bounded infinite set* of levels rather than with a single particular level. (To make this possible, an appropriate universe of levels should be used, such as the set of lists of natural numbers, ordered lexicographically.) To acquire a lock, the levels of the obligations held by the thread must be above the elements of the set; the new obligation's level is an element of the set.

5 Tool Support

We have extended the VeriFast tool [10] for separation logic-based modular verification of C and Java programs so that it supports verifying termination of busy-waiting C or Java programs. When verifying termination, VeriFast consumes a *call permission* at each recursive call or loop iteration. In the technical

report [29] we define a generalised version of our logic that instead of providing a special proof rule for busy-waiting loops, provides *wait permissions* and a *wait view shift*. A call permission of a *degree* δ can be turned into a wait permission of a degree $\delta' < \delta$ for a given signal s. A wait view shift for an unset signal s for which a wait permission of degree δ exists produces a call permission of degree δ, which can be used to fuel a busy-waiting loop. When busy-waiting for some signal s, we can generate new permissions to justify each iteration as long as s remains unset.

VeriFast allows threads to freely exchange permissions. This is useful to verify termination of non-blocking algorithms involving compare-and-swap loops [11]. However, we must be careful to prevent self-fueling busy-waiting loops. Hence, we restrict where a permission can be consumed based on the *thread phase* it was created in. The main thread's initial phase is ϵ. When a thread in phase p forks a new thread, its phase changes to $p.\mathsf{Forker}$ and the new thread starts in phase $p.\mathsf{Forkee}$. We allow a thread in phase p to consume a permission only if it was produced in an *ancestor thread phase* $p' \sqsubseteq p$.

The only change we had to make to VeriFast's symbolic execution engine was to enforce the thread phase rule. We encoded the other aspects of the logic simply as axioms in a *trusted header file*. We used this tool support to verify the bounded FIFO (§ 3) and the CLH lock (§ 4). The bounded FIFO proof [27] contains 160 lines of proof annotations for 37 lines of code (an annotation overhead of 435%) and takes 0.08s to verify. The CLH lock proof [26] contains 343 lines of annotations for 49 lines of code (an overhead of 700%) and takes 0.1s to verify.

6 Integrating Higher-Order Features

The logic we presented in this paper does not support higher-order features such as assertions that quantify over assertions, or storing assertions in the (logical) heap as the values of ghost cells. While we did not need such features to carry out our example proofs, they are generally useful to verify higher-order program modules against abstract specifications. The typical way to support such features in a program logic is by applying *step indexing* [1,17], where the domain of logical heaps is indexed by the number of execution steps left in the (partial) program trace under consideration. Assertions stored in a logical heap at index $n+1$ talk about logical heaps at index n; i.e., they are meaningful only *later*, after at least one more execution step has been performed.

It follows that such logics apply directly only to *partial* correctness properties. Fortunately, we can reduce a termination property to a safety property by writing our program in a programming language *instrumented* with runtime checks that guarantee termination. Specifically, we can write our program in a programming language that fulfils the following criteria: It tracks signals, obligations and permissions at runtime and has constructs for signal creation, waiting and setting a signal. The **fork** command takes as an extra operand the list of obligations to be transferred to the new thread (and the other constructs similarly take sufficient operands to eliminate any need for angelic choice). Threads get stuck when these constructs' preconditions are not satisfied, such as when a

thread waits for a signal while holding the obligation for that signal. We can then use a step-indexing-based higher-order logic such as Iris [14] to verify that no thread in our program ever gets stuck. Once we established this, we know none of the instrumentation has any effect and can be safely *erased* from the program.

7 Related & Future Work

In recent work [30] we propose a separation logic to verify termination of programs where threads busy-wait to be abruptly terminated. We generalize this work to support busy waiting for arbitrary conditions.

In [11] we propose an approach based on *call permissions* to verify termination of single- and multithreaded programs that involve loops and recursion. However, that work does not consider busy-waiting loops. In the technical report, we present a generalised logic that uses call permissions and allows busy waiting to be implemented using arbitrary looping and/or recursion. Furthermore, the use of call permissions allowed us to encode our case studies in our VeriFast tool which also uses call permissions for termination verification.

Liang and Feng [20, 21] propose LiLi, a separation logic to verify liveness of blocking constructs implemented via busy waiting. In contrast to our verification approach, theirs is based on the idea of contextual refinement. In their approach, client code involving calls of blocking methods of the concurrent object is verified by first applying the contextual refinement result to replace these calls by code involving primitive blocking operations and then verifying the resulting client code using some other approach. In contrast, specifications in our approach are regular Hoare-style triples and proofs are regular Hoare-style proofs.

In [9] we propose a Hoare logic to verify liveness properties of the I/O behaviour of programs that do not perform busy waiting. By combining that approach with the one we proposed in this paper, we expect to be able to verify I/O liveness of realistic concurrent programs involving both I/O and busy waiting, such as a server where one thread receives requests and enqueues them into a bounded FIFO, and another one dequeues them and responds. To support this claim, we encoded the combined logic in VeriFast and verified a simple server application where the receiver and responder thread communicate via a shared buffer [25].

8 Conclusion

We propose what is to the best of our knowledge the first separation logic for verifying termination of programs with busy waiting. We offer a soundness proof of the system of the paper in its extended version [28], and of a more general system in the technical report [29]. Further, we demonstrated its usability by verifying a realistic example. We encoded our logic and the realistic example in VeriFast [27] and used this encoding also to verify the CLH lock [26]. Moreover, we expect that our approach can be integrated into other existing concurrent separation logics such as Iris [14].

References

1. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (Sep 2001). https://doi.org/10.1145/504709.504712
2. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.639
3. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA (2002). https://doi.org/10.1145/582419.582440
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI '02 (2002). https://doi.org/10.1145/512529.512558
5. Hamin, J., Jacobs, B.: Deadlock-free monitors. In: ESOP (2018). https://doi.org/10.1007/978-3-319-89884-1_15
6. Hamin, J., Jacobs, B.: Transferring Obligations Through Synchronizations. In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 19:1–19:58 (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.19
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1968). https://doi.org/10.1145/363235.363259
9. Jacobs, B.: Modular verification of liveness properties of the I/O behavior of imperative programs. In: ISoLA (2020). https://doi.org/10.1007/978-3-030-61362-4_29
10. Jacobs, B. (ed.): VeriFast 21.04. Zenodo (2021). https://doi.org/10.5281/zenodo.4705416
11. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification of single-threaded and multithreaded programs. ACM Trans. Program. Lang. Syst. **40**, 12:1–12:59 (2018). https://doi.org/10.1145/3210258
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods (NFM 2011). vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4, `https://lirias.kuleuven.be/95720`
13. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (2016). https://doi.org/10.1145/2951913.2951943
14. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
15. Kim, J., Sjöberg, V., Gu, R., Shao, Z.: Safety and liveness of mcs lock—layer by layer. In: Asian Symposium on Programming Languages and Systems (2017)
16. Kobayashi, N.: A new type system for deadlock-free processes. In: CONCUR (2006). https://doi.org/10.1007/11817949_16
17. Lars Birkedal, Kristian Støvring, J.T.: The category-theoretic solution of recursive metric-space equations. Theoretical Computer Science **411**(47), 4102 – 4122 (2010). https://doi.org/10.1016/j.tcs.2010.07.010

18. Leino, K., Müller, P.: A basis for verifying multi-threaded programs. In: ESOP '09 Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. p. 378 (March 2009). https://doi.org/10.1007/978-3-642-00590-9_27
19. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: Proceedings of the 19th European Conference on Programming Languages and Systems. p. 407–426. ESOP'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_22
20. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL (2016). https://doi.org/10.1145/2837614.2837635
21. Liang, H., Feng, X.: Progress of concurrent objects with partial methods. Proc. ACM Program. Lang. **2**, 20:1–20:31 (2017). https://doi.org/10.1145/3158108
22. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**, 21–65 (1991). https://doi.org/10.1145/103727.103729
23. Mühlemann, K.: Method for reducing memory conflicts caused by busy waiting in multiple processor synchronisation. IEE Proceedings E - Computers and Digital Techniques **127**(3), 85–87 (1980). https://doi.org/10.1049/ip-e.1980.0017
24. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL (2001). https://doi.org/10.1007/3-540-44802-0_1
25. Reinhard, T., Jacobs, B.: VeriFast proof of I/O liveness for a simple server with a receiver and a responder thread communicating via a shared buffer. (2020), `https://github.com/verifast/verifast/blob/master/examples/busywaiting/ioliveness/echo_live_mt.c`
26. Reinhard, T., Jacobs, B.: VeriFast proof of safety for CLH lock. (2020), `https://github.com/verifast/verifast/blob/master/examples/busywaiting/clhlock/clhlock.c`
27. Reinhard, T., Jacobs, B.: VeriFast proof of termination for consumer-producer problem with bounded FIFO. (2020), `https://github.com/verifast/verifast/blob/master/examples/busywaiting/bounded_fifo.c`
28. Reinhard, T., Jacobs, B.: Ghost signals: Verifying termination of busy waiting (extended version) (2021), `https://arxiv.org/abs/2010.11762`
29. Reinhard, T., Jacobs, B.: Ghost signals: Verifying termination of busy waiting (technical report). Zenodo (2021). https://doi.org/10.5281/zenodo.4775181
30. Reinhard, T., Timany, A., Jacobs, B.: A Separation Logic to Verify Termination of Busy-Waiting for Abrupt Program Exit, p. 26–32. Association for Computing Machinery, New York, NY, USA (2020), `https://doi.org/10.1145/3427761.3428345`
31. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. Proceedings 17th Annual IEEE Symposium on Logic in Computer Science pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
32. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 176–201. Springer (2016). https://doi.org/10.1007/978-3-662-49498-1_8
33. Vafeiadis, V.: Concurrent separation logic and operational semantics. Electronic Notes in Theoretical Computer Science **276**, 335–351 (2011). https://doi.org/10.1016/j.entcs.2011.09.029, twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII)