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, Belgium
{tobias.reinhard,bart.jacobs}@kuleuven.be

Abstract. Programs for multiprocessor machines commonly perform busy-waiting for synchronization. We propose a separation logic using so-called *ghost signals* to modularly verify termination of such programs under fair scheduling. Intuitively spoken, ghost signals lift the runtime concept of *wait-notify* synchronization to the verification level and allow a thread to busy-wait for an event X while another thread promises to trigger X.

1 Introduction

Programs for multiprocessor machines commonly perform busy-waiting for synchronization [23,25]. In this paper, we propose a separation logic [26,33] to modularly verify termination of such programs under fair scheduling where some threads busy-wait for other threads to trigger a certain event, e.g., setting a memory flag. 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 an event X only terminates if the thread responsible for the event is sufficiently often scheduled to trigger X.

The approach we present in this paper to verify termination of busy-waiting programs builds on two of our earlier works: (i) In [14] we proposed so-called *call permissions* to verify termination of programs involving recursion and loops. (ii) In [31, 32] we proposed a separation logic to verify termination of busy-waiting for abrupt termination. We call said permissions *iteration permissions* as, for simplicity, in this paper we focus on loops rather than on recursion. An extension would, however, be straight-forward.

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 choose a well-founded set of degrees Δ, associate each iteration permission with a degree and start our verification with a finite number of these permissions. We let each loop iteration consume an iteration permission and allow to create new ones via weakening, i.e., replacing an iteration permission by finitely many of a lower degree. Then, we show that the finite stock of iteration permissions in the system is monotonically decreasing and strictly decreases with each loop iteration. The idea of using well-founded orders to verify termination is well-known in the literature [6, 18, 24].

However, the number of iterations that a busy-waiting loop performs typically depends on the runtime scheduling. In such a case, starting with a fixed number of iteration permissions and allowing weakening is not sufficient to prove termination. To overcome this limitation, we introduce *ghost signals* and *wait permissions*. This verification concept roughly corresponds to runtime *wait-notify* synchronization where one party waits for an event to happen and the party responsible for triggering the event notifies the waiting one when the event occurred. Ghost signals and wait permissions allow a thread to wait for an event.

We choose a well-founded set of levels $\mathcal{L}evs$ and associate each signal with one as well as with a boolean to indicate whether it is *set* or *unset*. When a thread creates a signal, it is initially unset and the thread obtains a so-called *obligation* [9, 10, 19, 20] to set the signal. Generally, an obligation requires the thread holding the obligation to discharge it by performing a certain action. This way, when a threat *knows* that some signal s has not been set, yet, while it does not hold an obligation for s, it knows that another thread holds the obligation and will eventually set s.

Just as with iteration permissions, we associate each wait permission with a degree. Before a thread can wait for a signal s, it has to obtain a wait permission for s, which consumes an iteration permission of a higher degree. A thread holding a wait permission of degree δ for an unset signal s can generate an iteration permission of degree δ by waiting for s. We ensure that no thread (directly or indirectly) waits for itself by requiring the level of s to be lower than the level of each held obligation.

Ghost signals and permissions are *ghost resources* [16], i.e., resources that only exist on the verification level and hence do not affect the program's runtime behaviour. In particular, setting a signal does not by definition correspond to any runtime event. So, in order to use a signal s effectively, we have to prove an invariant stating that s is set if and only if the event of interest has occurred.

We can prove that, assuming fair scheduling, in a verified program each signal is waited-for only finitely often. This allows us to view a wait permission as the finite collection of iteration permissions that will be generated by threads waiting for it. We extend the above soundness argument by (i) not just considering the stock of iteration permissions available at a certain point in time but also (ii) the stock of future iteration permissions that will be created by waiting for signals (via already available wait permissions). As above, this stock of iteration permissions decreases with each loop iteration. Hence, we can prove that verified programs terminate.

We start by gradually introducing the intuition behind our verification approach and the concepts we use. In § 2.1 and § 2.2, we introduce the aspects related to verifying data-race-freedom and deadlock-freedom, respectively, which are standard [5, 8, 14, 20]. In § 2.3, we introduce our approach for verifying termination of busy waiting. We continue in § 3 and § 4 by formally presenting the language and the logic, respectively. In the latter, we define the proof system and state the soundness theorem whose proof we outline in § 5. In § 6 we sketch the verification of a realistic example program to demonstrate our approach's

usability and address fine grained concurrency in § 7. Further, we sketch an extension of our logic that allows permission transfer between threads in § 8, describe the available tool support in § 9, and discuss integrating higher-order features in § 10. We conclude by comparing our approach to related work and reflecting on it in § 11 and § 12.

2 Verifying Termination of Busy-Waiting in a Nutshell

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. We will use the program presented in Fig. 1 as running example. It shows two threads communicating via a shared memory cell x that is initialized with 0 and protected by a mutex m. One thread sets x to 1 and the other thread busy-waits for x to change its value.

We introduce our program logic gradually. In § 2.1 and § 2.2, we introduce the aspects related to verifying data-race-freedom and deadlock-freedom, respectively, which are standard [5, 8, 14, 20]. In § 2.3, we introduce our approach for verifying termination of busy waiting.

```
let x := cons(0) in
let m := new_mutex in
fork ( while (acquire m;
                let y :=[x] in
                release m;
                y = 0)
       do skip);
acquire m;
[x] := 1;
release m
```

Fig. 1: Minimal Example Program.

2.1 Verifying Data Race Freedom

The two threads both access the shared heap cell x. Due to non-deterministic runtime scheduling, data races can occur if both try to access it simultaneously. To prevent this, we protect x by a mutex m and synchronize accesses. In this section, we introduce the aspects of our logic for verifying data-race-freedom; they are standard [5, 8].

Specifications We use Hoare triples $\{A\}\ c\ \{\lambda r.\, B(r)\}$ [12] to specify the behaviour of a program c. Such a triple expresses that if we execute c in a state that fulfils precondition A, then two properties hold: (i) c does not get stuck, i.e., it either terminates or diverges. (ii) If c terminates and r denotes the value returned by c, then postcondition $B(r)$ holds in its final state.

Proof System We define a proof relation $\vdash$ which ensures that whenever we can prove $\vdash \{A\}\ c\ \{\lambda r.\, B(r)\}$, then no race condition occurs during the execution of c. Fig. 2b presents some of the proof rules we use to define $\vdash$. The full set of rules is presented in the appendix (cf. Fig. 9 and 10). Fig. 2a illustrates how we can apply them to verify data race freedom of our minimal example. For now, ignore the obligations chunks $\mathsf{obs}(O)$ and levels marked in grey in the verification sketch. We are going to explain their use in the next subsection.

Accessing Heap Cells Our logic extends plain separation logic [26,33]. When we allocate x by applying PR-Cons, we get a points-to chunk $\ell_\mathsf{x} \mapsto 0$. It expresses that heap location ℓ_x, which we store in x, points to the value 0. We can split the chunk and give each half, i.e., $[\frac{1}{2}]\ell_\mathsf{x} \mapsto 0$, to one thread [3, 4]. This allows both to read x, cf. proof rule PR-ReadHeapLoc. However, in order to prevent data races, we require threads to own the full chunk $\ell_\mathsf{x} \mapsto 0$ to write x, cf. PR-AssignToHeap.

Lock Invariants Splitting x's points-to chunk does not work for the example program since one thread requires write access. Hence, they have to share the chunk and we must synchronize their accesses of x. We use the mutex m for this. In order to specify which resources m protects, we associate it with a lock invariant P. The mutex protects x whose value varies over time. Hence, we choose the invariant $P := \exists v_\mathsf{x}.\ \ell_\mathsf{x} \mapsto v_\mathsf{x}$ which abstracts over the concrete value v_x of x.

Ghost State & Ghost Proof Steps Directly after the creation of a mutex by proof rule PR-NewMutex, it is uninitialized. That is, it has not been associated with any lock invariant, yet. We must choose this invariant explicitly by performing a proof step that consumes invariant P and binds it to m. This step does not affect the resources in any way that would be relevant to the program's runtime behaviour. The resources that P describes still exist but their ownership has been transferred to the mutex. Hence, we say that it changes the *ghost state* and call it a *ghost proof step*. We express such steps in the form of a *view shift* relation $\Rrightarrow$ [17]. 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. View shift rule VS-MutInit' presented in Fig. 2b allows us to initialize m with invariant P. The full set of rules we use to define $\Rrightarrow$ are presented in the appendix (cf. Fig. 8).

Synchronization Since initializing m consumes lock invariant P, the shared heap cell x becomes unavailable to both threads. In order to read or write x, a thread must obtain the *exclusive* right to access x by acquiring m. The corresponding proof rule PR-Acquire changes the mutex's state from *unlocked* to *locked* and produces the lock invariant P. When we release m, we must apply PR-Release', which changes its state to *unlocked* and consumes P. That is, x becomes unavailable to the thread again, when it releases m. Mutex acquisition will not succeed during runtime if some thread already holds the mutex. Hence, our proof rules statically ensure that only one thread can access x at any time.

$\{\mathsf{obs}(\emptyset)\}$
let x := **cons**(0) **in let** m := **new_mutex in**
$\forall \ell_x, \ell_m.$
$\{\mathsf{obs}(\emptyset) * \ell_x \mapsto 0 * \mathsf{uninit}(\ell_m)\}$
init_mutex; $P := \exists v_x.\ \ell_x \mapsto v_x$
$\{\mathsf{obs}(\emptyset) * \mathsf{mutex}((\ell_m, 0), P)\}$
fork ($\{\mathsf{obs}(\emptyset) * [\frac{1}{2}]\mathsf{mutex}((\ell_m, 0), P)\}$
  while (**acquire** m
    $\{\mathsf{obs}(\{\!|(\mathsf{m}, 0)|\!\}) * \mathsf{locked}((\mathsf{m}, 0), P, \frac{1}{2}) * \exists v_x.\ \ell_x \mapsto v_x\}$
    let y :=[x] **in release** m; y = 0
    $\{\mathsf{obs}(\emptyset) * [\frac{1}{2}]\mathsf{mutex}((\mathsf{m}, 0), P)\}$)
  do skip);
$\{\mathsf{obs}(\emptyset) * [\frac{1}{2}]\mathsf{mutex}((\ell_m, 0), P)\}$
acquire m; [x] := 1;
$\{\mathsf{obs}(\{\!|(\mathsf{m}, 0)|\!\}) * \mathsf{locked}((\ell_m, 0), P, \frac{1}{2}) * \ell_x \mapsto 1\}$
release m
$\{\mathsf{obs}(\emptyset) * [\frac{1}{2}]\mathsf{mutex}((\ell_m, 0), P)\}$

(a) Proof outline verifying data race and deadlock freedom. Hints on ghost proof steps highlighted in red. Auxiliary abbreviations marked in brown.

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

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

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

PR-NewMutex
$\vdash \{\mathsf{True}\}\ \mathbf{new\_mutex}\ \{\lambda \ell.\ \mathsf{uninit}(\ell)\}$

PR-Acquire
$$\frac{\forall o \in O.\ \mathsf{lev}(m) <_{\mathsf{L}} \mathsf{lev}(o)}{\begin{array}{l}\{\mathsf{obs}(O) * [f]\mathsf{mutex}(m, P)\} \\ \vdash\ \mathbf{acquire}\ m.\mathsf{loc} \\ \{\mathsf{obs}(O \uplus \{\!|m|\!\}) * \mathsf{locked}(m, P, f) * P\}\end{array}}$$

PR-Release'
$\{\mathsf{obs}(O \uplus \{\!|m|\!\}) * \mathsf{locked}(m, P, f) * A\}$
$\vdash\ \mathbf{release}\ m.\mathsf{loc}$
$\{\mathsf{obs}(O) * [f]\mathsf{mutex}(m, P) * B\}$

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

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

VS-MutInit'
$\mathsf{uninit}(\ell) * P \Rrightarrow \mathsf{mutex}((\ell, L), P)$

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

(b) Proof rules and view shift rules for verification of data race and deadlock freedom.

Fig. 2: Verifying data race and deadlock freedom. Obligations chunks and levels marked in grey. Ignore them while reading § 2.1. Obligations and levels will be introduced in § 2.2.

2.2 Verifying Deadlock Freedom

Obligations During runtime, any attempt to acquire m will block if another thread already holds the mutex until this thread releases m again. If it does not release m, the thread trying to acquire m can never proceed with its execution. We use so-called *obligations* [9, 10, 19, 20] to prevent this. These are *ghost resources* [16], 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. Whenever a thread acquires m we generate an obligation to release it and releasing m discharges this obligation, cf. proof rules PR-Acquire and PR-Release' presented in Fig. 2b. We describe which obligations a thread holds by bundling them into obligations chunks $\mathsf{obs}(O)$, cf. Fig. 2a (obs chunks marked in grey). Consequently, $\mathsf{obs}(\emptyset)$ asserts that a thread does not hold any obligations. We can prove deadlock-freedom of a program c by proving that it discharges all its obligations, i.e., by proving $\vdash \{\mathsf{obs}(\emptyset)\}\ c\ \{\mathsf{obs}(\emptyset)\}$.

Whenever a thread forks a new thread, it can transfer ownership of resources to the newly forked thread, cf. PR-Fork. In particular, the forking thread can pass some of its obligations to the new thread. However, in order to prevent threads from dropping obligations via dummy forks, we only allow forking if the forked thread provably discharges all its obligations.

Levels It is not always straight-forward to see that a mutex will be released eventually. Consider a program with two mutexes m_1, m_2 and two threads. Let one thread execute **acquire** m_1; **acquire** m_2; **release** m_2; **release** m_1; and the other **acquire** m_2; **acquire** m_1; **release** m_1; **release** m_2;. Both threads' execution can get stuck as both mutexes' release depends on successful acquisition. To prevent such *wait cycles*, we choose a partially ordered set of levels $\mathcal{L}evs$ and associate every mutex with a level. In the example program, we choose 0 for m, cf. Fig. 2a (levels marked in grey). Additionally, we only allow a thread to acquire a mutex if its level is smaller than the level of each held obligation, cf. PR-Acquire. This also prevents any thread from attempting to acquire mutexes twice, e.g., **acquire** m; **acquire** m.

2.3 Verifying Termination

The approach we present in this paper to verify termination of busy-waiting programs builds on two of our earlier works: (i) In [14] we proposed so-called *call permissions* to verify termination of programs involving recursion and loops. (ii) In [31, 32] we proposed a separation logic to verify termination of busy-waiting for abrupt termination. We call said permissions *iteration permissions* as, for simplicity, in this paper we focus on loops rather than on recursion. An extension would, however, be straight-forward.

$\{\mathsf{obs}(\emptyset) * \mathsf{itperm}(1)\}$
let x := **cons**(0) **in let** m := **new_mutex in new_signal**;
$\forall \ell_x, \ell_m, id_s.$
$\{\mathsf{obs}(\{\!|(id_s, 1)|\!\}) * \mathsf{itperm}(1) * \ell_x \mapsto 0 * \mathsf{uninit}(\ell_m) * \mathsf{signal}((id_s, 1), \mathsf{False})\}$
$s := (id_s, 1), \quad mut := (\ell_m, 0), \quad P := \exists v_x.\ \ell_x \mapsto v_x * \mathsf{signal}(s, v_x \neq 0)$
$\{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{itperm}(1) * \mathsf{mutex}(mut, P)\}$
new_waitPerm;
$\{\mathsf{obs}(\{\!|s|\!\}) * \mathsf{wperm}(id_s, 0) * \mathsf{mutex}(mut, P)\}$
fork $(\{\mathsf{obs}(\emptyset) * \mathsf{wperm}(id_s, 0) * [\frac{1}{2}]\mathsf{mutex}(mut, P)\}$
  while (**acquire** m
    $\forall v_x.$
    $\{\mathsf{obs}(\{\!|mut|\!\}) * \mathsf{wperm}(id_s, 0) * \ell_x \mapsto v_x * \mathsf{signal}(s, v_x \neq 0) * \mathsf{locked}(mut, P, \frac{1}{2})\}$
    let y :=[x] **in**
    release m + **release_view_shift**
      $\{\mathsf{obs}(\emptyset) * \mathsf{wperm}(id_s, 0) * \ell_x \mapsto v_x * \mathsf{signal}(s, v_x \neq 0)\}$
      wait
      $\{(v_x = 0 \leftrightarrow \mathsf{itperm}(0)) * \mathsf{obs}(\emptyset) * \mathsf{wperm}(id_s, 0) * P\};$
    y = 0
    $\{\mathsf{obs}(\emptyset) * (v_x = 0 \to \mathsf{itperm}(0) * \mathsf{wperm}(id_s, 0) * [\frac{1}{2}]\mathsf{mutex}(mut, P))\}$
  do skip);
  $\{\mathsf{obs}(\emptyset)\}$
$\{\mathsf{obs}(\{\!|s|\!\}) * [\frac{1}{2}]\mathsf{mutex}(mut, P)\}$ $\quad \mathsf{lev}(mut) < \mathsf{lev}(s)$
acquire m; [x] := 1; **set_signal**;
$\{\mathsf{obs}(\{\!|mut|\!\}) * \mathsf{locked}(mut, P, \frac{1}{2}) * \ell_x \mapsto 1 * \mathsf{signal}(s, \mathsf{True})\}$
release m
$\{\mathsf{obs}(\emptyset) * [\frac{1}{2}]\mathsf{mutex}(mut, P)\}$
$\{\mathsf{obs}(\emptyset)\}$

(a) Proof outline verifying termination. Hints on ghost proof steps highlighted in red. Auxiliary abbreviations marked in brown.

PR-While
$$\vdash \{I\}\ c_b \left\{ \begin{array}{l} \lambda b.\ (b = \mathsf{True} \vee b = \mathsf{False}) \\ \quad * (b \to \mathsf{itperm}(\delta) * I) \\ \quad * (\neg b \to B) \end{array} \right\} \over \vdash \{I\}\ \mathbf{while}\ c_b\ \mathbf{do\ skip}\ \{B\}$$

VS-WaitPerm
$$\delta' <_\Delta \delta \over \mathsf{itperm}(\delta) \Rrightarrow \mathsf{wperm}(id, \delta')$$

PR-Release
$$\mathsf{obs}(O) * A \Rrightarrow \mathsf{obs}(O) * P * B \over \{\mathsf{obs}(O \uplus \{\!|m|\!\}) * \mathsf{locked}(m, P, f) * A\} \vdash \mathbf{release}\ m.\mathsf{loc}\ \{\mathsf{obs}(O) * [f]\mathsf{mutex}(m, P) * B\}$$

VS-Wait
$$\forall o \in O.\ \mathsf{lev}(s) <_\mathsf{L} \mathsf{lev}(o) \over \mathsf{obs}(O) * \mathsf{wperm}(s.\mathsf{id}, \delta) * \mathsf{signal}(s, b) \Rrightarrow \mathsf{obs}(O) * \mathsf{wperm}(s.\mathsf{id}, \delta) * \mathsf{signal}(s, b) * (\neg b \leftrightarrow \mathsf{itperm}(\tau, \delta))$$

VS-WeakPerm
$$\delta' <_\Delta \delta \qquad N \in \mathbb{N} \over \mathsf{itperm}(\delta) \Rrightarrow \mathop{\Large *}_{1,\ldots,N} \mathsf{itperm}(\delta')$$

VS-NewSignal
$$\mathsf{obs}(O) \Rrightarrow \exists id.\ \mathsf{obs}(O \uplus \{\!|(id, L)|\!\}) * \mathsf{signal}((id, L), \mathsf{False})$$

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

VS-MutInit
$$\neg\mathsf{mentionsPerms}(P) \over \mathsf{uninit}(\ell) * P \Rrightarrow \mathsf{mutex}((\ell, L), P)$$

(b) Proof rules and view shift rules for termination verification.

Fig. 3: Verifying termination.

Iteration Permissions In order to prove termination we need to prove that loops perform only finitely many iterations. Fig. 3b shows the proof and view shift rules we use and Fig. 3a presents our termination verification of the minimal example. We will now explain their use. Consider the rule PR-While. We only consider busy-waiting loops of the form **while** c_b **do skip**, where c_b is a command whose result controls the loop and **skip** is a no-op. We start our verification with a finite number of iteration permissions. Each loop iteration consumes one of these permissions. We have to prove that the condition c_b returns a boolean value to prevent the execution from getting stuck. As the loop busy-waits while c_b yields True, we must prove that, in this case, it preserves some invariant I and that we have an additional iteration permission for the iteration to consume.

To avoid the need of fixing the number of needed permissions at the beginning of a verification, we allow to generate finitely many permissions by *weakening*, cf. view shift rule VS-WeakPerm. We choose a well-founded set of degrees Δ and associate each permission with one. Weakening allows us to replace one iteration permission of degree δ by finitely many of a lower degree δ'. The idea of using well-founded orders to verify termination is well-known in the literature [6,18,24].

Ghost Signals & Wait Permissions How many iterations a busy-waiting loop performs often depends on the concrete runtime scheduling as in the minimal example. In such a case, starting with a fixed number of iteration permissions and allowing weakening is not sufficient to prove termination. To overcome this limitation, we introduce *ghost signals* and *wait permissions*. This verification concept roughly corresponds to runtime *wait-notify* synchronization where one party waits for an event to happen and the party responsible for triggering the event notifies the waiting one when the event occurred. Ghost signals and wait permissions allow a thread to wait for an event, e.g., x to be set, in a similar manner.

Just like permissions, ghost signals are ghost resources. We associate each signal with a unique ID and a boolean $b \in \mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$. We can create a signal via a view shift with VS-NewSignal. Initially, the signal is unset, i.e., associated with False. Creating a signal automatically creates an obligation to set the signal. In our termination verification proof, the main thread holds this obligation and will eventually discharge it by setting the signal when it sets x.

We associate each wait permission with a signal ID and a degree. A wait permission for some signal s enables a thread to wait until s is set. By applying VS-WaitPerm, we can replace an iteration permission by a wait permission of a lower degree. Rule VS-Wait allows a thread holding a signal s and a wait permission for s of degree δ to generate an iteration permission of degree δ. The thread can use this iteration permission to justify another loop iteration.

In our termination verification, both threads share the created signal via the lock invariant P. The main thread eventually sets the signal and the busy-waiting thread uses it and the wait permission it holds to justify loop iterations until s is set. However, the latter thread actually loops until x is set. So, in order to use the ghost signal effectively, we have to prove an invariant stating that the signal is consistent with the event we wait for, i.e., we need to prove that s is

associated with True if and only if x has been set. We fix this as part of the lock invariant P.

Furthermore, we need to ensure that threads do not wait for themselves. Hence, we associate every signal with a level and only allow a thread to wait for s if its level is lower than the level of each held obligation, cf. VS-Wait. In our example, the main thread acquires the mutex m while it holds the obligation for s and sets s while still holding m. The lock invariant stating consistency between s and x makes this necessary. Hence, we choose the level of m to be lower than the level of s as required by PR-Acquire. This, however, means that the forked thread can only wait for s (i.e. apply VS-Wait) after it discharged the obligation for m. We solve this by allowing a view shift when releasing m that takes place after the obligation for m has been discharged but before the mutex's lock invariant P is consumed. In the verification outline, this view shift is marked as *release_view_shift*.

So far we allowed threads to share arbitrary resources via lock invariants. However, this becomes problematic when permissions are involved. Consider a non-terminating program with two infinitely looping threads t_1 and t_2 sharing a signal s. t_1 holds the obligation for s and hence cannot wait for it. In each iteration t_2 can wait for s and thereby create two iteration permissions of which it transfers one through the lock invariant to t_1. This way, both threads can justify their infinite loop and t_1 essentially busy-waits for itself to set s. We prevent such inconsistencies by prohibiting lock invariants from mentioning permissions and restrict VS-MutInit' accordingly. The new view shift rule VS-MutInit is presented in Fig. 3b.

Note that if we could prophesy ahead of time how many times we would need to busy-wait before the signal s was set, we could just use the weakening rule VS-WeakPerm. Consider any execution of any program verified using our proof rules and the graph depicting its control flow at runtime, i.e., the graph where each path represents the execution of a particular thread including ghost steps corresponding to view shifts, e.g., creating a signal and waiting. We call this a *program order graph*. From the perspective of a path in this graph that only waits for signals that are waited for only finitely often, a wait permission is effectively an encapsulated prophecy: $\mathsf{wperm}(id_s, \delta) = \ast_{1,\dots,N} \mathsf{itperm}(\delta)$ where N is the number of times s will be waited for (or 0 if s is waited for infinitely often). By this interpretation of wait permissions, the stock of iteration permissions on such a path decreases at each loop iteration. 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. Then the path that carries the obligation for $s_{\min}$ waits only for signals that are waited for finitely often. By the above argument, it is a finite path. This contradicts the fact that it carries an obligation that is never discharged. Notice that the above argument relies on the property that every nonempty 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.

3 Language

In this section and the next, we present our approach formally. In this section, we define the programming language; in § 4 we define the proof system. We state the soundness theorem and outline our soundness proof in § 5.

We consider a simple imperative programming language with support for multi-threading, shared memory and synchronization via mutexes. Fig. 4 presents its syntax and semantics. For its definition we assume (i) an infinite set of program variables $x \in \mathcal{X}$, (ii) an infinite set of heap locations $\ell \in \mathcal{L}ocs$, (iii) a set of values $v \in \mathit{Values}$ which includes heap locations, booleans $\mathbb{B} = \{\mathsf{True}, \mathsf{False}\}$ and the unit value tt , (iv) a set of operations $op \in \mathit{Ops}$ and (v) an infinite, totally ordered and well-founded set of thread IDs $\theta \in \Theta$.

The language contains standard sets of pure expressions *Exps* and (potentially) side-effectful commands *Cmds*, cf. Fig. 4b. The latter includes commands for heap allocation and manipulation, forking and loops. We define *physical heaps* [14] (as opposed to *logical heaps* [14] presented in the next section) as a finite set of *physical resource chunks*, cf. Fig. 4c. A points-to chunk $\ell \mapsto v$ expresses that heap location ℓ points to value v [14,33]. Moreover, we have chunks to represent unlocked and locked mutexes.

We represent a program state by a physical heap and a *thread pool*, which we define as a partial function mapping a finite number of thread IDs to threads, cf. Fig. 4d. Thread IDs are unique and never reused. Hence, we represent running threads by commands and terminated ones by term instead of removing threads from the pool. We define the operational semantics of our language in terms of two small-step reduction relations: $\rightsquigarrow_{\mathsf{st}}$ for single threads and $\rightsquigarrow_{\mathsf{tp}}$ for thread pools, cf. Fig. 4e and 4f. Since expressions are pure and their evaluation is deterministic (cf. Def. 13 in appendix) we identify closed expressions with their ascribed value. (i) $h, c \rightsquigarrow_{\mathsf{st}} h', c', T$ expresses that heap h and command c are reduced in a single step to h' and c' and that this thread forks a set of threads T. This set is either empty or a singleton as no step forks more than one thread. (ii) $h, P \overset{\theta}{\rightsquigarrow}_{\mathsf{tp}} h', P'$ expresses that heap h and thread pool P are reduced in a single step to h' and P'. ID θ identifies the thread reduced in this step.

As thread scheduling is non-deterministic, so is our thread pool reduction relation $\rightsquigarrow_{\mathsf{tp}}$. Consider the minimal example presented in the previous section in Fig. 1. It does not terminate if the main thread is never scheduled after the new thread was forked. Hence, our verification approach relies on the assumption of fair scheduling. That is, we assume that every thread is always eventually scheduled while it remains running. Further, we represent program executions by sequences of reduction steps. As we primarily consider infinite sequences in this paper, we define reduction sequences to be infinite to simplify our terminology.

Definition 1 (Reduction Sequence). *Let $(h_i)_{i\in\mathbb{N}}$ and $(P_i)_{i\in\mathbb{N}}$ be infinite sequences of physical heaps and thread pools, respectively. We call $(h_i, P_i)_{i\in\mathbb{N}}$ a* reduction sequence *if there exists a sequence of thread IDs $(\theta_i)_{i\in\mathbb{N}}$ such that $h_i, P_i \overset{\theta_i}{\rightsquigarrow}_{\mathsf{tp}} h_{i+1}, P_{i+1}$ holds for every $i \in \mathbb{N}$.*

Definition 2 (Fairness). *We call a reduction sequence* $(h_i, P_i)_{i\in\mathbb{N}}$ fair *iff for all* $i \in \mathbb{N}$ *and* $\theta \in \mathsf{dom}(P_i)$ *with* $P_i(\theta) \neq \mathsf{term}$ *there exists some* $k \geq i$ *with*

$$h_k, P_k \overset{\theta}{\rightsquigarrow}_{\mathsf{tp}} h_{k+1}, P_{k+1}.$$

4 Logic

In this section we formalize the logic we sketched in § 2. Fig. 5 defines the assertions we use to express which resources a thread holds. For the definition we assume (i) an infinite set of ghost signal IDs $id \in \mathcal{ID}$ and (ii) infinite, partially ordered and well-founded sets of levels $L \in \mathcal{L}evs$ and degrees $\delta \in \Delta$. We denote the latter sets' order relations by $<_{\mathsf{L}}$ and $<_{\Delta}$, respectively. We define ghost resources as described in § 2, cf. Fig. 5b, and the syntax of our assertion language to match the presented verification sketches, cf. Fig. 5c [1].

We define *logical heaps* and *logical resources* [14] in Fig. 5d. These concepts correspond to physical heaps and physical resources but additionally encompass ghost resources and ownership. For instance, logical resources include signal chunks and initialized mutex chunks are associated with a lock invariant. Rather than being a set of resources, logical heaps map logical resources to fractions. This allows us to express which portion of a resource a thread owns. We interpret assertions in terms of a model relation, cf. Fig. 5e. $H \vDash_{\mathsf{A}} a$ expresses that assertion a holds with respect to logical heap H. Further, we define various predicates to characterize a heap's contents.

Definition 3 (Logical Heap Predicates). *We call a logical heap* H complete *and write* $\mathsf{complete}_{\mathsf{lh}}(H)$ *if it contains exactly one obligations chunk, i.e., if there exist a bag of obligations* O *with* $H(\mathrm{obs}_{\mathsf{lRes}}(O)) = 1$ *and if there does not exist any bag of obligations* O' *with* $O \neq O'$ *and* $H(\mathrm{obs}_{\mathsf{lRes}}(O')) > 0$.

We call a logical heap H finite *and write* $\mathsf{finite}_{\mathsf{lh}}(H)$ *if it contains only finitely many resources, i.e., if the set* $\{r^{\mathsf{l}} \in \mathcal{R}^{\mathsf{log}} \mid H(r^{\mathsf{l}}) > 0\}$ *is finite.*

We already presented the most important rules that we use to define our proof relation and view shift relation in Fig. 2b and 3b in § 2 (excluding auxiliary rules VS-MutInit' and PR-Release'). We present the full collection of proof rules and view shift rules in the appendix.

Definition 4 (View Shift & Proof Relations). *We define a view shift relation* $\Rrightarrow \subset \mathcal{A} \times \mathcal{A}$ *and a proof relation* $\vdash \subset \mathcal{A} \times \mathit{Cmds} \times (\mathit{Values} \to \mathcal{A})$ *according to the rules presented in Fig. 8, 9 and 10 presented in the appendix. We state the provability of a Hoare triple in the form of* $\vdash \{A\}\ c\ \{\lambda r.\, B(r)\}$ *where* r *captures the value returned by* c*. To simplify the notation, we omit the result value if it is clear from the context or irrelevant.*

[1] That is, we define the set of assertions $\mathcal{A}$ as the least fixpoint of F where $F(A) = \{\mathsf{True}, \mathsf{False}\} \cup \{\neg a \mid a \in A\} \cup \{a_1 \wedge a_2 \mid a_1, a_2 \in A\} \cup \cdots \cup \{\bigvee A' \mid A' \subseteq A\} \cup \ldots$. Since F is a monotonic function over a complete lattice, it has a least fixpoint according to the Knaster-Tarski theorem [35].

$x \in \mathcal{X}$: Program variables $\quad \ell \in \mathcal{L}ocs$: Heap locations $\quad op \in Ops$: Operations
$v \in Values \supseteq \{\mathsf{tt}\} \cup \mathbb{B} \cup \mathcal{L}ocs$: Values $\quad \theta \in \Theta$: Thread IDs

(a) Assumed sets and variables. $\mathcal{X}$, $\mathcal{L}ocs$, Θ infinite. Θ totally ordered and well-founded.

$$
\begin{array}{lll}
e \in Exps & ::= & x \mid v \mid e = e \mid \neg e \mid op(\bar{e}) \\
c \in Cmds & ::= & e \mid \textbf{while } c \textbf{ do skip} \mid \textbf{fork } c \mid \textbf{let } x := c \textbf{ in } c \mid \textbf{if } c \textbf{ then } c \mid \\
 & & \textbf{cons}(e) \mid [e] \mid [e] := e \mid \textbf{new\_mutex} \mid \textbf{acquire } e \mid \textbf{release } e \\
E \in EvalCtxts & ::= & \textbf{if } \square \textbf{ then } c \mid \textbf{let } x := \square \textbf{ in } c
\end{array}
$$

(b) Syntax.

$$
\begin{array}{lll}
r^{\mathsf{p}} \in \mathcal{R}^{\mathsf{phys}} & ::= & \ell \mapsto v \mid \mathrm{unlocked}_{\mathsf{pRes}}(\ell) \mid \mathrm{locked}_{\mathsf{pRes}}(\ell) \\
h \in Heaps^{\mathsf{phys}} & := & \mathcal{P}_{\mathsf{fin}}(\mathcal{R}^{\mathsf{phys}}) \\
\mathsf{locs}_{\mathsf{pRes}}(h) & := & \{\ell \in \mathcal{L}ocs \mid \mathrm{unlocked}_{\mathsf{pRes}}(\ell) \in h \vee \mathrm{locked}_{\mathsf{pRes}}(\ell) \in h \vee \exists v.\, \ell \mapsto v \in h\}
\end{array}
$$

(c) Physical resources & heaps.

$$
\begin{array}{l}
P \in \mathcal{TP} \;:=\; \Theta \rightharpoonup_{\mathsf{fin}} (Cmds \cup \{\mathsf{term}\}). \\
\emptyset_{\mathsf{tp}} \;:\; \Theta \rightharpoonup_{\mathsf{fin}} (Cmds \cup \{\mathsf{term}\}),\ \mathsf{dom}(\emptyset_{\mathsf{tp}}) = \emptyset. \\
+_{\mathsf{tp}} :\ \mathcal{TP} \times \{C \subset Cmds \mid |C| \leq 1\} \to \mathcal{TP} \\
P +_{\mathsf{tp}} \emptyset \;:= P, \\
P +_{\mathsf{tp}} \{c\} := P[\theta_{\mathrm{new}} := c] \quad \text{for} \quad \theta_{\mathrm{new}} := \min(\Theta \setminus \mathsf{dom}(P)).
\end{array}
$$

(d) Thread pools. Θ denotes a set of thread IDs and term a terminated thread.

$$\frac{h, c \rightsquigarrow_{\mathsf{st}} h', c', T}{h, E[c] \rightsquigarrow_{\mathsf{st}} h', E[c'], T} \qquad h, \textbf{fork } c \rightsquigarrow_{\mathsf{st}} h, \mathsf{tt}, \{c\} \qquad h, \textbf{if } \mathsf{True} \textbf{ then } c \rightsquigarrow_{\mathsf{st}} h, c$$

$$h, \textbf{if } \mathsf{False} \textbf{ then } c \rightsquigarrow_{\mathsf{st}} h, \mathsf{tt} \qquad h, \textbf{let } x := v \textbf{ in } c \rightsquigarrow_{\mathsf{st}} h, c[v/x]$$

$$h, \textbf{while } c \textbf{ do skip} \rightsquigarrow_{\mathsf{st}} h, \textbf{if } c \textbf{ then while } c \textbf{ do skip}$$

$$\frac{\ell \notin \mathsf{locs}_{\mathsf{pRes}}(h)}{h, \textbf{cons}(v) \rightsquigarrow_{\mathsf{st}} h \cup \{\ell \mapsto v\}, \ell} \qquad \frac{\ell \mapsto v \in h}{h, [\ell] \rightsquigarrow_{\mathsf{st}} h, v}$$

$$h \sqcup \{\ell \mapsto v'\}, [\ell] := v \rightsquigarrow_{\mathsf{st}} h \sqcup \{\ell \mapsto v\}, \mathsf{tt}$$

$$\frac{\ell \notin \mathsf{locs}_{\mathsf{pRes}}(h)}{h, \textbf{new\_mutex} \rightsquigarrow_{\mathsf{st}} h \cup \{\mathrm{unlocked}_{\mathsf{pRes}}(\ell)\}, \ell}$$

$$h \sqcup \{\mathrm{unlocked}_{\mathsf{pRes}}(\ell)\}, \textbf{acquire } \ell \rightsquigarrow_{\mathsf{st}} h \sqcup \{\mathrm{locked}_{\mathsf{pRes}}(\ell)\}, \mathsf{tt}$$

$$h \sqcup \{\mathrm{locked}_{\mathsf{pRes}}(\ell)\}, \textbf{release } \ell \rightsquigarrow_{\mathsf{st}} h \sqcup \{\mathrm{unlocked}_{\mathsf{pRes}}(\ell)\}, \mathsf{tt}$$

(e) Single thread reduction rules.

$$\frac{P(\theta) = c \qquad h, c \rightsquigarrow_{\mathsf{st}} h', c', T}{h, P \overset{\theta}{\rightsquigarrow}_{\mathsf{tp}} h', P[\theta := c'] +_{\mathsf{tp}} T} \qquad \frac{P(\theta) = v}{h, P \overset{\theta}{\rightsquigarrow}_{\mathsf{tp}} h, P[\theta := \mathsf{term}]}$$

(f) Thread pool reduction rules

Fig. 4: Language definition.

In § 2 we verified the termination of the minimal example by proving that it discharges all its obligations (while starting with no obligations and a single iteration permission). The following theorem states that this verification approach is sound.

Theorem 1 (Soundness). *Let* $\vdash \{\mathsf{obs}(\emptyset) * \mathop{\Large *}_{i=1,\dots,N} \mathsf{itperm}(\delta_i)\}\ c\ \{\mathsf{obs}(\emptyset)\}$ *hold. There exists no fair, infinite reduction sequence* $(h_i, P_i)_{i\in\mathbb{N}}$ *with* $h_0 = \emptyset$ *and* $P_0 = \{(\theta_0, c)\}$ *for any choice of* θ_0.

5 Soundness

We already sketched the high-level intuition behind our soundness argument in § 2.3. In this section, we go into more detail, provide the most important lemmas we need to prove our soundness theorem 1, sketch their proofs and finally prove the soundness theorem itself. A detailed soundness proof and all definitions can be found in the appendix and in the technical report [27].

Bridging the Gap During runtime, all threads share one physical heap where every thread is free to access every resource. This does not reflect the notions of ownership and lock invariants which we maintain on the verification level. It also does not allow us to restrict actions based on levels, e.g., only allowing the acquisition of a mutex m if its level is lower than the levels of all held obligations. Hence: (i) We annotate every thread by a logical heap to express which resources it owns (including ghost resources) and thereby obtain an *annotated thread pool*. (ii) We represent the program state by an *annotated heap* that keeps track of lock invariants and levels. In particular, we associate unlocked mutexes with logical heaps to represent the resources they protect. Since annotated heaps keep track of levels, they also keep track of signals. We denote annotated thread pools and heaps by P^{a} and h^{a}, respectively.

We define annotated versions $\rightsquigarrow_{\mathsf{atp}}$ and $\rightsquigarrow_{\mathsf{ast}}$ of the relations $\rightsquigarrow_{\mathsf{tp}}$ and $\rightsquigarrow_{\mathsf{st}}$, respectively. The annotated reduction semantics we thereby obtain needs to reflect ghost proof steps implemented by view shifts. Hence, we define $\rightsquigarrow_{\mathsf{atp}}$ in terms of two relations: (i) $\rightsquigarrow_{\mathsf{ghost}}$ for ghost steps and (i) $\rightsquigarrow_{\mathsf{real}}$ for actual program execution steps. The annotated semantics ensure that a reduction gets stuck if a thread violates any of the restrictions formulated by our proof rules. Fig. 6 presents three of the rules that we use to define these relations. The upper two rules show that a loop iteration consumes an iteration permission and that the reduction gets stuck if the looping thread does not own one. The lower rule shows that a thread can generate iteration permissions by waiting for an unset signal, but only if the signal's level is lower than the level of each held obligation. We see that these rules comply with PR-While and VS-Wait.

Interpreting Hoare Triples We interpret program specifications $\{A\}\ c\ \{B\}$ in terms of a model relation $\vDash_{\mathsf{H}} \{A\}\ c\ \{B\}$ and an auxiliary safety relation

$$id \in \mathcal{ID} : \text{Signal IDs} \quad L \in \mathcal{L}evs : \text{Levels} \quad \delta \in \Delta : \text{Degrees} \quad b \in \mathbb{B} : \text{Booleans}$$

(a) Assumed sets and variables. $\mathcal{ID}$, $\mathcal{L}evs$, Δ infinite. $\mathcal{L}evs$, Δ partially ordered and well-founded.

$$s \in \mathcal{S} := \mathcal{ID} \times \mathcal{L}evs \quad \Omega := \mathcal{ID} \times \Delta \quad \Lambda := \Delta \quad o \in \mathcal{O} := (\mathcal{L}ocs \cup \mathcal{ID}) \times \mathcal{L}evs$$
$$(id, L).\mathsf{id} := id \quad (\ell, L).\mathsf{loc} := \ell \quad \mathsf{lev}((\_, L)) := L$$

(b) Ghost resources: Ghost signals $\mathcal{S}$, wait/ iteration permissions Ω/ Λ, obligations $\mathcal{O}$.

$$f \in \mathcal{F} := \{f \in \mathbb{Q} \mid 0 < f \leq 1\} \quad A \subseteq \mathcal{A} \quad O \in \mathit{Bags}(\mathcal{O}) \quad b \in \mathbb{B} \quad \text{Index set } I$$

$$\begin{array}{lll} a \in \mathcal{A} ::= \mathsf{True} \mid \mathsf{False} \mid \neg a \mid a \wedge a \mid a \vee a \mid a * a \mid [f]\ell \mapsto v \mid \bigvee A \\ \quad \mid \mathsf{uninit}(\ell) \mid [f]\mathsf{mutex}((\ell, L), a) \mid \mathsf{locked}((\ell, L), a, f) \\ \quad \mid \mathsf{signal}((id, L), b) \mid \mathsf{obs}(O) \mid \mathsf{wperm}(id, \delta) \mid \mathsf{itperm}(\delta) \\ a_1 \to a_2 := \neg a_1 \vee a_2 & a_1 \leftrightarrow a_2 := (a_1 \to a_2) \wedge (a_2 \to a_1) \\ \exists i \in I.\, a(i) := \bigvee\{a(i) \mid i \in I\} & \forall i \in I.\, a(i) := \neg\exists i \in I.\, \neg a(i) \end{array}$$

(c) Assertion syntax. We omit quantification domain I when it is clear from the context.

$$\begin{array}{l} r^{\mathsf{l}} \in \mathcal{R}^{\mathsf{log}} ::= \ell \mapsto v \mid \mathrm{signal}_{\mathsf{lRes}}((id, L), b) \mid \mathrm{uninit}_{\mathsf{lRes}}(\ell) \mid \mathrm{mutex}_{\mathsf{lRes}}((\ell, L), a) \\ \quad \mid \mathrm{locked}_{\mathsf{lRes}}((\ell, L), a, f) \mid \mathrm{obs}_{\mathsf{lRes}}(O) \mid \mathrm{wperm}_{\mathsf{lRes}}(id, \delta) \mid \mathrm{itperm}_{\mathsf{lRes}}(\delta) \\ H \in \mathit{Heaps}^{\mathsf{log}} := \mathcal{R}^{\mathsf{log}} \to \{q \in \mathbb{Q} \mid q \geq 0\} \\ \emptyset_{\mathsf{log}} \in \mathit{Heaps}^{\mathsf{log}} : \_ \mapsto 0 \qquad (H_1 + H_2)(r^{\mathsf{l}}) := H_1(r^{\mathsf{l}}) + H_2(r^{\mathsf{l}}) \\ \{r^{\mathsf{l}}_1, \ldots, r^{\mathsf{l}}_n\} := \begin{cases} r^{\mathsf{l}}_i \mapsto 1 \\ x \mapsto 0 \text{ if } x \notin \{r^{\mathsf{l}}_1, \ldots, r^{\mathsf{l}}_n\} \end{cases} \qquad (q \cdot H)(r^{\mathsf{l}}) := q \cdot (H(r^{\mathsf{l}})) \end{array}$$

(d) Logical resources and heaps.

$H \vDash_{\mathsf{A}} \mathsf{True}$	
$H \nvDash_{\mathsf{A}} \mathsf{False}$	
$H \vDash_{\mathsf{A}} \neg a$	if $H \nvDash_{\mathsf{A}} a$
$H \vDash_{\mathsf{A}} a_1 \wedge a_2$	if $H \vDash_{\mathsf{A}} a_1 \wedge H \vDash_{\mathsf{A}} a_2$
$H \vDash_{\mathsf{A}} a_1 \vee a_2$	if $H \vDash_{\mathsf{A}} a_1 \vee H \vDash_{\mathsf{A}} a_2$
$H \vDash_{\mathsf{A}} a_1 * a_2$	if $\exists H_1, H_2 \in \mathit{Heaps}^{\mathsf{log}}.\ H = H_1 + H_2 \wedge H_1 \vDash_{\mathsf{A}} a_1 \wedge H_2 \vDash_{\mathsf{A}} a_2$
$H \vDash_{\mathsf{A}} [f]\ell \mapsto v$	if $H(\ell \mapsto v) \geq f$
$H \vDash_{\mathsf{A}} \bigvee A$	if $\exists a \in A.\ H \vDash_{\mathsf{A}} a$
$H \vDash_{\mathsf{A}} [f]\mathsf{uninit}(\ell)$	if $H(\mathrm{uninit}_{\mathsf{lRes}}(\ell)) \geq f$
$H \vDash_{\mathsf{A}} [f]\mathsf{mutex}(m, P)$	if $H(\mathrm{mutex}_{\mathsf{lRes}}(m, P)) \geq f$
$H \vDash_{\mathsf{A}} [f]\mathsf{locked}(m, P, f_u)$	if $H(\mathrm{locked}_{\mathsf{lRes}}(m, P, f_u)) \geq f$
$H \vDash_{\mathsf{A}} [f]\mathsf{signal}(s, b)$	if $H(\mathrm{signal}_{\mathsf{lRes}}(s, b)) \geq f$
$H \vDash_{\mathsf{A}} \mathsf{obs}(O)$	if $H(\mathrm{obs}_{\mathsf{lRes}}(O)) \geq 1$
$H \vDash_{\mathsf{A}} \mathsf{wperm}(id, \delta)$	if $H(\mathrm{wperm}_{\mathsf{lRes}}(id, \delta)) \geq 1$
$H \vDash_{\mathsf{A}} \mathsf{itperm}(\delta)$	if $H(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) \geq 1$

(e) Assertion model relation. $H \nvDash_{\mathsf{A}} a$ expresses that $H \vDash_{\mathsf{A}} a$ does not hold.

Fig. 5: Assertions.

$$h^{\mathsf{a}}, H, \textbf{while } c \textbf{ do skip} \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, \textbf{if } c \textbf{ then } (\textbf{consumeItPerm}; \textbf{while } c \textbf{ do skip})$$

$$h^{\mathsf{a}}, H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\}, \textbf{consumeItPerm} \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, \mathsf{tt}$$

$$\frac{\mathrm{signal}_{\mathsf{aRes}}(s, \mathsf{False}) \in h^{\mathsf{a}} \qquad P^{\mathsf{a}}(\theta) = (H, c) \qquad H(\mathrm{wperm}_{\mathsf{lRes}}(s.\mathsf{id}, \delta)) \geq 1 \qquad H(\mathrm{obs}_{\mathsf{lRes}}(O)) \geq 1 \qquad \forall o \in O.\, \mathsf{lev}(s) <_{\mathsf{L}} \mathsf{lev}(O)}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}}, P^{\mathsf{a}}[\theta := (H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},\, c)]}$$

Fig. 6: Example reduction rules for annotated reduction semantics. Auxiliary command **consumeItPerm** expresses consumption of an iteration permission.

$\mathsf{safe}(H, c)$. Intuitively, a command c is *safe* under a logical heap H if H provides the necessary resources so that for every execution of c, there is a corresponding annotated execution of c that does not get stuck. We write: (i) $\mathsf{annot}_{\mathsf{tp}}(P^{\mathsf{a}}, P)$ to express that P^{a} is an annotated version of P. (ii) $h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h$ to express that h^{a} is *compatible* with h. (iii) And $\mathsf{consistent}_{\mathsf{conf}}(h^{\mathsf{a}}, P^{\mathsf{a}})$ to express that the annotated machine configuration $(h^{\mathsf{a}}, P^{\mathsf{a}})$ is *consistent* in the sense that h^{a} is compatible with the logical heap H containing all resources owned by threads in P^{a} or protected by unlocked locks in h^{a} and that heap locations in H are unique.

Definition 5 (Safety). *We define the safety predicate* $\mathsf{safe} \subseteq \mathit{Heaps}^{\mathsf{log}} \times \mathit{Cmds}$ *coinductively as the greatest solution (with respect to* $\subseteq$*) of the following equation:*

$$\begin{array}{l}
\mathsf{safe}(H, c) \\
\Longleftrightarrow \\
\mathsf{complete}_{\mathsf{lh}}(H) \ \rightarrow \\
\forall P, P'.\, \forall \theta \in \mathsf{dom}(P).\, \forall h, h'.\, \forall P^{\mathsf{a}}.\, \forall h^{\mathsf{a}}. \\
\quad \mathsf{consistent}_{\mathsf{conf}}(h^{\mathsf{a}}, P^{\mathsf{a}}) \ \wedge \ h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h \ \wedge \\
\quad P(\theta) = c \ \wedge \ P^{\mathsf{a}}(\theta) = (H, c) \ \wedge \ \mathsf{annot}_{\mathsf{tp}}(P^{\mathsf{a}}, P) \ \wedge \ h, P \overset{\theta}{\rightsquigarrow}_{\mathsf{tp}} h', P' \ \rightarrow \\
\quad \exists P^{\mathsf{G}}, P^{\mathsf{a}\prime}.\, \exists h^{\mathsf{G}}, h^{\mathsf{a}\prime}. \\
\qquad h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}^{*}_{\mathsf{ghost}} h^{\mathsf{G}}, P^{\mathsf{G}} \ \wedge \ h^{\mathsf{G}}, P^{\mathsf{G}} \overset{\theta}{\rightsquigarrow}_{\mathsf{real}} h^{\mathsf{a}\prime}, P^{\mathsf{a}\prime} \ \wedge \ \mathsf{annot}_{\mathsf{tp}}(P^{\mathsf{a}\prime}, P') \ \wedge \\
\qquad h^{\mathsf{a}\prime}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h' \ \wedge \\
\qquad \forall (H_f, c_f) \in \mathsf{range}(P^{\mathsf{a}\prime}) \setminus \mathsf{range}(P^{\mathsf{a}}).\, \mathsf{safe}(H_f, c_f).
\end{array}$$

Definition 6 (Hoare Triple Model Relation). *We define the model relation for Hoare triples* $\vDash_{\mathsf{H}} \subset \mathcal{A} \times \mathit{Cmds} \times (\mathit{Values} \to \mathcal{A})$ *such that:*

$$\begin{array}{c}
\vDash_{\mathsf{H}} \{A\}\ c\ \{\lambda r.\, B(r)\} \\
\Longleftrightarrow \\
\forall H_F.\, \forall E.\, (\forall v.\, \forall H_B.\, H_B \vDash_{\mathsf{A}} B(v) \ \rightarrow \ \mathsf{safe}(H_B + H_F,\, E[v])) \\
\rightarrow \ \ \forall H_A.\, H_A \vDash_{\mathsf{A}} A \ \rightarrow \ \mathsf{safe}(H_A + H_F,\, E[c])
\end{array}$$

We can instantiate context E in above definition to $\textbf{let } x := \square \textbf{ in } \mathsf{tt}$, which yields the consequent $\mathsf{safe}(H_A + H_F, \textbf{let } x := c \textbf{ in } \mathsf{tt})$. Note that this implies

$\mathsf{safe}(H_A + H_F, c)$. Further, every specification we can derive in our proof system also holds in our model.

Lemma 1 (Hoare Triple Soundness). *Let* $\vdash \{A\}\ c\ \{B\}$ *hold; then also* $\vDash_{\mathsf{H}} \{A\}\ c\ \{B\}$ *holds.*

Proof. Proof by induction on the derivation of $\vdash \{A\}\ c\ \{B\}$.

Constructing Annotated Executions Given a command c which provably discharges all obligations and a fair reduction sequence for c, we can construct a corresponding annotated reduction sequence $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$. This is a useful tool to analyse program executions as $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ carries much more information than the original sequence, e.g., which obligations and permissions a thread holds. Note that our definition of fairness forbids $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ to perform ghost steps forever and that we use $h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H$ to express that h^{a} is compatible with H.

Definition 7 (Fairness of Annotated Reduction Sequences). *We call an annotated reduction sequence* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ fair *iff for all* $i \in \mathbb{N}$ *and* $\theta \in \mathsf{dom}(P_i^{\mathsf{a}})$ *with* $P_i^{\mathsf{a}}(\theta).\mathsf{cmd} \neq \mathsf{term}$ *there exists some* $k \geq i$ *with*

$$h_k^{\mathsf{a}}, P_k^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{real}} h_{k+1}^{\mathsf{a}}, P_{k+1}^{\mathsf{a}}.$$

Lemma 2 (Construction of Annotated Reduction Sequences). *Suppose we can prove* $\vDash_{\mathsf{H}} \{A\}\ c\ \{\mathsf{obs}(\emptyset)\}$. *Let* H_A *be a logical heap with* $H_A \vDash_{\mathsf{A}} A$ *and* $\mathsf{complete}_{\mathsf{lh}}(H_A)$ *and* h_0^{a} *an annotated heap with* $h_0^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H_A$. *Let* $(h_i, P_i)_{i\in\mathbb{N}}$ *be a fair plain reduction sequence with* $h_0^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h_0$ *and* $P_0 = \{(\theta_0, c)\}$ *for some thread ID* θ_0 *and command* c.

Then, there exists a fair annotated reduction sequence $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *with* $P^{\mathsf{a}} = \{(\theta_0, (H_A, c))\}$ *and* $\mathsf{consistent}_{\mathsf{conf}}(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})$ *for all* $i \in \mathbb{N}$.

Proof. We can construct the annotated reduction sequence inductively from the plain reduction sequence.

Program Order Graph In the remainder of this section, we prove that programs where every thread discharges all obligations terminate. For this, we need to analyse each thread's control flow, i.e., the subsequence of execution steps belonging to the thread. We do this by taking a sequence $(h_i, P_i)_{i\in\mathbb{N}}$ representing a program execution, constructing an annotation $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ carrying additional information and then analysing its *program order graph* $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$ which represents the execution's control flow. The graph has the form $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}) = (\mathbb{N}, E)$ for $E \subset (\mathbb{N} \times \Theta \times N^r \times \mathbb{N})$ where N^r is the set of (annotated) reduction rule names. Every node $i \in \mathbb{N}$ represents the i^{th} reduction step, i.e. $h_i^{\mathsf{a}}, P_i^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h_{i+1}^{\mathsf{a}}, P_{i+1}^{\mathsf{a}}$. Consider an edge $(i, \theta, n, j) \in E$ between two nodes i, j. n describes the combination of reduction rules applied in step i. The edge expresses one of two things: (i) Either, steps i and j belong to the same thread θ and j denotes the first scheduling of θ after i. (ii) Or, step i forks the new thread θ and j is the first

time θ is scheduled. A path in this graph represents the control flow of a directly related line of threads, i.e., parent, child, etc. In case the reduction sequence $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ is clear from the context, we denote the graph by G to simplify the notation.

Signal Capacity Threads can wait for signals to generate iteration permissions and thereby justify loop iterations. Consider some path p in a program order graph that represents the execution of some thread (and potentially some direct descendants) in a verified program. We prove that p is finite by (i) considering the stock of held iteration permissions and of future iteration permissions created via waiting and (ii) proving that this stock decreases. To do this, we need a notion that precisely captures these *future iteration permissions*. As a first step, we consider those signals that are waited for only finitely often along p. To each such signal s, we assign a *capacity* at every node i on p, i.e., $\mathsf{sigCap}_p(i)$, which is the bag of iteration permissions that will be created by waiting for s after i along the remaining suffix of p.

For the following definition, note that we represent paths in a program order graph by subsgraphs and that S_G^∞ denotes the set of signals waited for infinitely often within a subgraph G. Further, $\mathsf{waitEdges}_G(s)$ denotes the set of edges (a, θ, n, b) in G where a represents a wait step. We call these edges *wait edges*. For any edge, $\mathsf{itperms}_G((a, \theta, n, b))$ denotes the bag of iteration permissions generated by step a.

Definition 8 (Signal Capacity). *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ be a fair annotated reduction sequence and $G = (V, E)$ be a subgraph of the sequence's program order graph. We define the function $\mathsf{sigCap}_G : (\mathcal{S} \setminus S_G^\infty) \times \mathbb{N} \to \mathit{Bags}_{\mathsf{fin}}(\Lambda)$ mapping signals and indices to bags of iteration permissions as follows:*

$$\mathsf{sigCap}_G(s, i) \quad := \biguplus_{\substack{(a,\theta,n,b) \in \mathsf{waitEdges}_G(s) \\ a \geq i}} \mathsf{itperms}_G((a, \theta, n, b)).$$

We call $\mathsf{sigCap}_G(s, i)$ the capacity *of signal s at index i.*

Consider any path p representing some thread's execution on which no signal is waited for infinitely often. For every node i on p, we consider the stock of iteration permissions that are either held by the thread in step i or that will be created along the rest of the path. We can prove that this stock of permissions decreases at each loop iteration and that p must hence be finite.

Lemma 3. *Let $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$ be a program order graph and let $p = (V, E)$ be a path in G with $S_p^\infty = \emptyset$. For every $\theta \in \mathsf{dom}(P_0^{\mathsf{a}})$ let $P_0^{\mathsf{a}}(\theta).\mathsf{heap}$ be finite and complete. Then, p is finite.*

Proof (Sketch). Assume p is infinite. We prove a contradiction by assigning a finite capacity to every node along the path. Let θ_i be the ID of the thread reduced in step i.

Consider the function $\mathsf{nodeCap} : V \to \mathit{Bags}_{\mathsf{fin}}(\Lambda)$ defined as

$$\mathsf{nodeCap}(i) \;:=\; \mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap}) \;\uplus \bigbiguplus_{\substack{id \in \{id' \mid (id',\_) \in \mathsf{waitPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})\} \\ L \in \mathcal{L}evs}} \mathsf{sigCap}_p((id, L), i).$$

where $\mathsf{itPerms}_{\mathsf{lh}} : \mathit{Heaps}^{\mathsf{log}} \to \mathit{Bags}(\Lambda)$ and $\mathsf{waitPerms}_{\mathsf{lh}} : \mathit{Heaps}^{\mathsf{log}} \to \mathit{Bags}(\Omega)$ map any logical heap to the bag of contained iteration and wait permissions.

For every $i \in V$, the capacity of node i, i.e., $\mathsf{nodeCap}(i)$, is the union of two finite iteration permission bags: (i) To the left $\mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ captures all iteration permissions held by θ_i in step i. (ii) To the right $\bigbiguplus \mathsf{sigCap}_p((id, L), i)$ captures all iteration permissions that will be created along the suffix of p that starts at node i by waiting for signals for which thread θ_i already holds a wait permission (id, δ) in step i.

Note that for every $i \in V$, the bag of iteration permissions returned by $\mathsf{nodeCap}(i)$ is indeed finite. All initial thread-local heaps are finite. Consequently, $\mathsf{itPerms}_{\mathsf{lh}}(P_0^{\mathsf{a}}.\mathsf{heap})$ and $\mathsf{waitPerms}_{\mathsf{lh}}(P_0^{\mathsf{a}}.\mathsf{heap})$ are finite. Our annotated semantics preserves this finiteness and hence $\mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ and $\mathsf{waitPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ are finite as well. Since signal IDs are unique, for every fixed choice of i and id, there is at most one level L, for which $\mathsf{sigCap}_p((id, L), i) \neq \emptyset$. By assumption, along p all signals are waited for only finitely often, i.e., $S_p^\infty = \emptyset$. Hence, also the big union $\bigbiguplus \mathsf{sigCap}_p((id, L), i)$ is defined and finite.

The well-founded partial order $<_{\Delta} \subseteq \Lambda \times \Lambda$ induces a well-founded partial order $\prec_\Lambda \subset \mathit{Bags}_{\mathsf{fin}}(\Lambda) \times \mathit{Bags}_{\mathsf{fin}}(\Lambda)$ for finite iteration permission bags [6]. Consider the sequence $(\mathsf{nodeCap}(i))_{i \in V}$. Since every element is a finite bag of iteration permissions, we can order it by $\prec_\Lambda$. We can prove that $\mathsf{nodeCap}$ is monotonically decreasing and that $\mathsf{nodeCap}(j) \prec_\Lambda \mathsf{nodeCap}(i)$ holds for edges (i, θ, n, j) corresponding to a loop iteration consuming an iteration permission. We can also prove that every infinite path, such as p, contains infinitely many such edges. Hence, $(\mathsf{nodeCap}(i))_{i \in V}$ is an infinitely descending chain. This contradicts the well-foundedness of $\prec_\Lambda$.

We proceed to prove that no signals are waited for infinitely often.

Lemma 4. *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i \in \mathbb{N}}$ be a fair annotated reduction sequence with $P_0^{\mathsf{a}} = \{(\theta_0, (H_0, c))\}$, $\mathsf{finite}_{\mathsf{ah}}(h_0^{\mathsf{a}})$, $\mathsf{complete}_{\mathsf{lh}}(H_0)$, $\mathsf{finite}_{\mathsf{lh}}(H_0)$ and $\mathsf{consistent}_{\mathsf{conf}}(h_0^{\mathsf{a}}, P_0^{\mathsf{a}})$. Let H_0 contain no signal or wait permission chunks. Further, let h_0^{a} contain no chunks $\mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P)$ where H_P contains any signal chunks. Let G be the program order graph of $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i \in \mathbb{N}}$. Then, $S_G^\infty = \emptyset$.*

Proof (Sketch). Suppose $S_G^\infty \neq \emptyset$. Since $\mathcal{L}evs$ is well-founded, the same holds for the set $\{\mathsf{lev}(s) \mid s \in S^\infty\}$. Hence, there is some $s_{\min} \in S^\infty$ for which no $z \in S^\infty$ with $\mathsf{lev}(z) <_{\mathsf{L}} \mathsf{lev}(s_{\min})$ exists.

Since neither the initial logical heap H_0 nor any unlocked lock invariant stored in h_0^{a} does contain any signals, $s_{\min}$ must be created during the reduction sequence. The reduction step creating signal $s_{\min}$ simultaneously creates an obligation to set $s_{\min}$. Analogous to our proof rules, the annotated semantics only allow

threads to wait for a signal s (i) if the signal's level is lower than the level of each held obligation and (ii) if s has not been set yet. Hence, we know for every wait edge (a, θ, n, b) referring to $s_{\min}$ that (i) thread θ does not hold any obligation for $s_{\min}$ (i.e., there is exactly one obligations chunk with $P_a^{\mathsf{a}}(\theta).\mathsf{heap}(\mathrm{obs}_{\mathsf{lRes}}(O)) = 1$ and for this chunk $s_{\min} \notin O$ holds) and (ii) $s_{\min}$ has not been set, yet (i.e. $\mathrm{signal}_{\mathsf{aRes}}(s_{\min}, \mathsf{False}) \in h_a^{\mathsf{a}}$). Hence, in step a another thread $\theta_{\mathrm{ob}} \neq \theta$ must hold the obligation for $s_{\min}$ (i.e. $P_a^{\mathsf{a}}(\theta_{\mathrm{ob}}).\mathsf{heap}(\mathrm{obs}_{\mathsf{lRes}}(O)) = 1$ for some bag of obligations O with $s_{\min} \in O$). Since there are infinitely many wait edges concerning $s_{\min}$ in G, the signal is never set.

By fairness, for every wait edge as above, there must be a non-ghost reduction step $h_k^{\mathsf{a}}, P_k^{\mathsf{a}} \overset{\theta_{\mathrm{ob}}}{\rightsquigarrow}_{\mathsf{atp}} h_{k+1}^{\mathsf{a}}, P_{k+1}^{\mathsf{a}}$ of the thread θ_{ob} holding the obligation for $s_{\min}$ with $k \geq a$. Hence, there exists an infinite path p_{ob} in G where each edge $(e, \theta_{\mathrm{ob}}, n, f) \in \mathsf{edges}(p_{\mathrm{ob}})$ concerns some thread θ_{ob} holding the obligation for $s_{\min}$. (Note that this thread ID does not have to be constant along the path, since the obligation can be passed on during fork steps.)

The path p_{ob} does not contain wait edges $(e, \theta_{\mathrm{ob}}, n, f)$ for any $s^\infty \in S^\infty$, since this would require s^∞ to be of a lower level than all held obligations. This restriction implies $\mathsf{lev}(s^\infty) <_{\mathsf{L}} \mathsf{lev}(s_{\min})$ and would hence contradict the minimality of $s_{\min}$. That is, $S^\infty_{p_{\mathrm{ob}}} = \emptyset$.

The annotated reduction semantics preserve the finiteness of thread-local heaps. Since H_0 is finite, the same holds for every logical heap associated with the root of p_{ob}. This allows us to apply Lemma 3, by which we get that p_{ob} is finite. A contradiction.

Lemma 5. *Let* $\vDash_{\mathsf{H}} \{\mathsf{obs}(\emptyset) * \overset{i=1,\ldots,N}{\ast} \mathsf{itperm}(\delta_i)\}\ c\ \{\mathsf{obs}(\emptyset)\}$ *hold. There exists no fair, infinite annotated reduction sequence* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *with* $P_0^{\mathsf{a}} = \{(\theta_0, (H_0, c))\}$, $h_0^{\mathsf{a}} = \emptyset$ *and* $H_0 = \{\mathrm{obs}_{\mathsf{lRes}}(\emptyset), \mathrm{itperm}_{\mathsf{lRes}}(\delta_1), \ldots, \mathrm{itperm}_{\mathsf{lRes}}(\delta_N)\}$.

Proof. Suppose a reduction sequence as described above exists. We are going to prove a contradiction by considering its infinite program order graph G.

Since P_0^{a} contains only a single thread, G is a binary tree with an infinite set of vertices. By the Weak König's Lemma [34] G has an infinite branch, i.e. an infinite path p starting at root 0.

The initial logical heap H_0 is complete and finite and the initial annotated machine configuration $(h_0^{\mathsf{a}}, P_0^{\mathsf{a}})$ is consistent. By Lemma 4 we know that $S^\infty_G = \emptyset$. Since $S^\infty_p \subseteq S^\infty_G$, we get $S^\infty_p = \emptyset$. This allows us to apply Lemma 3, by which we get that p is finite, which is a contradiction.

Theorem 1 (Soundness). *Let* $\vdash \{\mathsf{obs}(\emptyset) * \overset{i=1,\ldots,N}{\ast} \mathsf{itperm}(\delta_i)\}\ c\ \{\mathsf{obs}(\emptyset)\}$ *hold. There exists no fair, infinite reduction sequence* $(h_i, P_i)_{i\in\mathbb{N}}$ *with* $h_0 = \emptyset$ *and* $P_0 = \{(\theta_0, c)\}$ *for any choice of* θ_0.

Proof. Assume that such a reduction sequence exists. By Hoare triple soundness, Lemma 1, we get $\vDash_{\mathsf{H}} \{\mathsf{obs}(\emptyset) * \overset{i=1,\ldots,N}{\ast} \mathsf{itperm}(\tau, \delta_i)\}\ c\ \{\mathsf{obs}(\emptyset)\}$ from the assumption $\vdash \{\mathsf{obs}(\emptyset) * \overset{i=1,\ldots,N}{\ast} \mathsf{itperm}(\tau, \delta_i)\}\ c\ \{\mathsf{obs}(\emptyset)\}$. Consider the logical heap $H_0 =$

$\{\mathrm{obs}_{\mathsf{lRes}}(\emptyset), \mathrm{itperm}_{\mathsf{lRes}}(\delta_1), \ldots, \mathrm{itperm}_{\mathsf{lRes}}(\delta_N)\}$ and the annotated heap $h_0^{\mathsf{a}} = \emptyset$. It holds $H_0 \vDash_{\mathsf{A}} \mathsf{obs}(\emptyset) * \bigstar_{i=1,\ldots,N} \mathsf{itperm}(\tau, \delta_i)$, $h_0^{\mathsf{a}} \,_{\mathsf{ah}}\!\sim_{\mathsf{lh}} H_0$ (since H_0 does not contain any logical resources with an annotated counterpart) and $h_0^{\mathsf{a}} \,_{\mathsf{ah}}\!\sim_{\mathsf{ph}} h_0$ (since both heaps are empty). This allows us to apply Lemma 2, by which we can construct a corresponding fair annotated reduction sequence $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ that starts with $h_0^{\mathsf{a}} = \emptyset$ and $P_0^{\mathsf{a}} = \{(\theta_0, (H_0, c))\}$. By Lemma 5 $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ does not exist. A contradiction.

6 A Realistic Example

To demonstrate the expressiveness of the presented verification approach, we verified the termination of the program presented in Fig. 7a. 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 termination 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.

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. 7b and adapt our logical resource 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 $\mathsf{uninitSig}(\ell)$ 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.

This change does not affect the soundness of our verification approach. However, it drastically simplifies proofs, such as the one discussed below, that would normally require the conditional creation of new signals within loops. Using the finer-grained view shifts, we can create all the signals we need centrally at the beginning of the proof. Since we thereby do not create any obligations, this step does not affect the acquisition of locks later on, as long as the signals remain uninitialized. Moreover, since we already know all signal IDs, we can centrally create all the wait permissions we need.

```
let fifo10 := cons(nil) in let m := new_mutex in
let cp := cons(100) in let cc := cons(100) in
fork (
  while (
    acquire m;
    let f := [fifo10] in
    if size(f) < 10 then (
      let c := [cp] in
      [fifo10] := f · (c :: nil);
      [cp] := c − 1
    );
    release m;
    let c := [cp] in
    c ≠ 0
  ) do skip;
);
```

```
while (
  acquire m;
  let f := [fifo10] in
  if size(f) > 0 then (
    let c := [cc] in
    [fifo10] := tail(f);
    [cc] := c − 1
  );
  release m;
  let c := [cc] in
  c ≠ 0
) do skip
```

(a) Realistic example program with a producer and a consumer thread communicating via a shared bounded FIFO.

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. 7: Realistic verification example.

Creating Permissions & Signals We choose $\Delta = \mathbb{N}$ and start our verification with a single iteration permission $\mathsf{itperm}(2)$ and without obligations. Initially we create 400 iteration permissions of degree 1 by weakening the initial one. Additionally, we allocate 200 signal IDs $id^{100}_{\text{push}}, \ldots, id^{1}_{\text{push}}, id^{100}_{\text{pop}}, \ldots, id^{1}_{\text{pop}}$. We use 200 of our iteration permissions to create one wait permission of degree 0 for each signal. 100 of the remaining 200 iteration permissions will be used by each thread to justify the iterations in which it pushes or pops, respectively, i.e., the productive iterations in which it does not have to wait.

Using the Signals 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. Eventually, the producer will hold the obligation for s^i_{push} while the signal is initialized and it will set s^i_{push} when it pushes the number i into the FIFO. Meanwhile, the consumer will use s^i_{push} and $\mathsf{wperm}(id^i_{\text{push}}, 0)$ 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} and will set s^i_{pop} when it pops the number i. The producer uses s^i_{pop} and $\mathsf{wperm}(id^i_{\text{pop}}, 0)$ to wait for the consumer to pop i from the queue when it is full.

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 $\mathcal{L}evs = \mathbb{N}$, $\mathsf{lev}(\mathsf{m}) = 0$ and $\mathsf{lev}(s) > 0$ for every signal s. Both threads will wait at the end of an iteration inside a release view shift, i.e., after they discharged the obligation to release the mutex. So, the mutex level does not interfere with the wait steps. The producer waits when the queue is full and the consumer when it is empty. 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 $\mathsf{lev}(s^{i+10}_{\text{pop}}) < \mathsf{lev}(s^i_{\text{push}})$ and $\mathsf{lev}(s^i_{\text{push}}) < \mathsf{lev}(s^i_{\text{pop}})$ hold. We solve this by choosing $\mathsf{lev}(s^i_{\text{pop}}) = 102 - i$ and $\mathsf{lev}(s^i_{\text{push}}) = 101 - i$.

Distributing Resources Both threads count from 100 downwards. So, we use the allocated IDs id^{100}_{push}, id^{100}_{pop} to initialize our first two signals $s^{100}_{\text{push}} = (id^{100}_{\text{push}}, 1)$, $s^{100}_{\text{pop}} = (id^{100}_{\text{pop}}, 2)$ and transfer their ownership to the mutex. We split the mutex chunk and pass one half to each thread such that both can access the initialized signals. Additionally, the producer thread receives the 100 iteration permissions, the obligation for s^{100}_{push}, the uninitialized signal chunks $\mathsf{uninitSig}(id^{99}_{\text{push}})$, …, $\mathsf{uninitSig}(id^{1}_{\text{push}})$ and the wait permissions $\mathsf{wperm}(id^{100}_{\text{pop}}, 0)$, …, $\mathsf{wperm}(id^{1}_{\text{pop}}, 0)$ upon its forking. The rest remains with the consumer, i.e., 100 iteration permissions, the obligation for s^{100}_{pop}, uninitialized signal chunks $\mathsf{uninitSig}(id^{99}_{\text{pop}})$, …, $\mathsf{uninitSig}(id^{1}_{\text{pop}})$ and wait permissions $\mathsf{wperm}(id^{100}_{\text{pop}}, 0)$, …, $\mathsf{wperm}(id^{1}_{\text{pop}}, 0)$.

Verifying Termination This setup suffices to verify the example program. Each thread holds (i) the iteration permissions it needs to justify its productive iter-

ations and (ii) the wait permissions that in combination with the acquired lock invariant allow to wait until the next productive iteration. 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. We present the verification sketch in the technical report [27]. Furthermore, we encoded [30] the proof in VeriFast [15].

7 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 [29] the CLH lock [11] against a specification that is very similar to our proof rules for built-in mutexes shown in Fig. 2. 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.

8 Allowing Permission Transfer

In the logic presented so far, transferring permissions between threads is not allowed, to prevent self-fueling busy-waiting loops. However, as we showed in earlier work [14], permission transfer is useful for verifying termination of non-blocking algorithms involving compare-and-swap loops. For this reason, we relax our logic as follows: we qualify each permission by the *thread phase* that produced it, and we allow a thread in phase p to consume a permission only if it was produced by an *ancestor thread phase* $p' \sqsubseteq p$. Initially, the main thread is in the *root phase* ϵ, and all permissions provided as part of the program's precondition are considered to have been produced by the root phase. When a thread t running in phase p forks a new thread, the new thread's initial phase is the new phase $p.\mathsf{Forkee}$ and t itself moves into the new phase $p.\mathsf{Forker}$. Weakening an iteration permission and converting an iteration permission into a wait permission do not affect the permission's phase. As a result, in a program that does not involve busy-waiting, all permissions are qualified by the root phase and can be transferred between threads freely. Permissions produced by waiting in a phase p, however, can be transferred only between descendants of p. We formalize this relaxed logic in the technical report [27].

9 Tool Support

We have extended our VeriFast tool for separation logic-based modular verification of C and Java programs so that it supports verifying termination of busy-waiting C or Java programs. We had already added *call permissions* (similar to iteration permissions) to VeriFast as part of earlier work [14] on verifying termination of programs using primitive blocking constructs without assuming fairness of the thread scheduler. The only change we had to make to VeriFast's symbolic execution engine was to enforce the thread phase rule mentioned in § 8. 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 (§ 6) and the CLH lock (§ 7).

10 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* [1,2], 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 run-time checks that guarantee termination. Specifically, we can write our program in a programming language that tracks signals, obligations, iteration permissions, and wait permissions at run time, has constructs for iteration permission weakening, signal creation, wait permission creation, waiting, and setting a signal, and where 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), and that gets stuck when these constructs' preconditions are not satisfied, such as when a thread waits for a signal while holding the obligation for that signal, or when it performs a loop iteration without holding an iteration permission. We can then use a step-indexing-based higher-order logic such as Iris [17] to verify that our program never gets stuck. Once we established this, we know none of the instrumentation has any effect and can be safely *erased* from the program.

11 Related & Future Work

Liang and Feng [21, 22] 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.

D'Osualdo et al. [7] propose TaDA Live, a separation logic for verifying termination of busy-waiting. This logic allows to modularly reason about fine-grained concurrent programs and blocking operations that are implemented in terms of busy-waiting and non-blocking primitives. It uses the concept of obligations to express thread-local liveness invariants, e.g., that a thread eventually releases an acquired lock. One difference with our work is that TaDA Live's assertions are not syntactically stable; that is, certain proof rules require the user to prove stability of certain assertions with respect to environment actions as a side condition. This may make it more difficult to provide effective tool support: whereas we could easily implement our logic in our VeriFast tool, and we expect our logic to be easy to integrate into other separation logics such as Iris [17] and their tool support, we are not aware of tool support for TaDA Live at this time.

In recent work [13] we proposed 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 present one, 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 [28].

12 Conclusion

In this paper we proposed a separation logic to verify termination of programs with busy-waiting. We proved our logic sound and demonstrated its usability by verifying a realistic example. Further, we encoded our logic and the realistic example in VeriFast [30] and used this encoding also to verify the CLH lock [29]. Moreover, we expect that our approach can be straightforwardly integrated into other existing concurrent separation logics such as Iris [17].

References

1. 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
2. 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
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.J.: Permission accounting in separation logic. In: POPL '05 (2005). https://doi.org/10.1145/1040305.1040327

4. Boyland, J.: Checking interference with fractional permissions. In: SAS (2003). https://doi.org/10.1007/3-540-44898-5_4
5. Brookes, S., O'Hearn, P.: Concurrent separation logic. ACM SIGLOG News **3**, 47–65 (2016). https://doi.org/10.1145/2984450.2984457
6. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. In: ICALP (1979). https://doi.org/10.1007/3-540-09510-1_15
7. D'Osualdo, E., Farzan, A., Gardner, P., Sutherland, J.: Tada live: Compositional reasoning for termination of fine-grained concurrent programs. CoRR **abs/1901.05750** (2019), `http://arxiv.org/abs/1901.05750`
8. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, S.: Local reasoning for storable locks and threads. In: APLAS (2007). https://doi.org/10.1007/978-3-540-76637-7_3
9. Hamin, J., Jacobs, B.: Deadlock-free monitors. In: ESOP (2018). https://doi.org/10.1007/978-3-319-89884-1_15
10. Hamin, J., Jacobs, B.: Transferring Obligations Through Synchronizations. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 19:1–19:58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.19, `http://drops.dagstuhl.de/opus/volltexte/2019/10811`
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1968). https://doi.org/10.1145/363235.363259
13. Jacobs, B.: Modular verification of liveness properties of the I/O behavior of imperative programs. Accepted at International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (2020), `https://people.cs.kuleuven.be/~bart.jacobs/isola2020.pdf`
14. 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
15. 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`
16. 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
17. 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
18. Klop, J.: Term rewriting systems. In: LICS 1993 (1990)
19. Kobayashi, N.: A new type system for deadlock-free processes. In: CONCUR (2006). https://doi.org/10.1007/11817949_16
20. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: ESOP (2010). https://doi.org/10.1007/978-3-642-11957-6_22
21. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL 2016 (2016). https://doi.org/10.1145/2837614.2837635
22. 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

23. 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
24. Middeldorp, A., Zantema, H.: Simple termination revisited. In: CADE (1994). https://doi.org/10.1007/3-540-58156-1_33
25. 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
26. 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
27. Reinhard, T., Jacobs, B.: Ghost signals: Verifying termination of busy-waiting (technical report) (2020), `https://people.cs.kuleuven.be/~tobias.reinhard/ghostSignals--TR.pdf`
28. 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`
29. Reinhard, T., Jacobs, B.: VeriFast proof of safety for CLH lock. (2020), `https://github.com/verifast/verifast/blob/master/examples/busywaiting/clhlock/clhlock.c`
30. 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`
31. Reinhard, T., Timany, A., Jacobs, B.: A separation logic to verify termination of busy-waiting for abrupt program exit. Accepted at Formal Techniques for Java-like Programs (2020), `https://arxiv.org/abs/2010.07800`
32. Reinhard, T., Timany, A., Jacobs, B.: A separation logic to verify termination of busy-waiting for abrupt program exit: Technical report (2020), `https://arxiv.org/abs/2007.10215`
33. 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
34. Simpson, S.: Subsystems of second order arithmetic. In: Perspectives in mathematical logic (1999). https://doi.org/10.1017/CBO9780511581007
35. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**, 285–309 (1955). https://doi.org/10.2307/2963937

Appendix

A General

Definition 9 (Projections). *For any Cartesian product $C = \prod_{i\in I} A_i$ and any index $k \in I$, we denote the k^{th} projection by $\pi_k^C : \prod_{i\in I} A_i \to A_k$. We define*

$$\pi_k^C((a_i)_{i\in I}) \;:=\; a_k.$$

In case the domain C is clear from the context, we write π_k instead of π_k^C.

Definition 10 (Disjoint Union). *Let A, B be sets. We define their disjoint union as*

$$A \sqcup B \;:=\; A \cup B$$

if $A \cap B = \emptyset$ and leave it undefined otherwise.

Definition 11 (Bags). *For any set X we define the set of bags $Bags(X)$ and the set of finite bags $Bags_{\mathsf{fin}}(X)$ over X as*

$$\begin{aligned} Bags(X) &:= X \to \mathbb{N}, \\ Bags_{\mathsf{fin}}(X) &:= \{B \in Bags(X) \mid \{x \in B \mid B(x) > 0\} \textit{ finite}\}. \end{aligned}$$

We define union and subtraction of bags as

$$\begin{aligned} (B_1 \uplus B_2)(x) &:= B_1(x) + B_2(x), \\ (B_1 \setminus B_2)(x) &:= \max(0,\; B_1(x) - B_2(x)). \end{aligned}$$

For finite bags where the domain is clear from the context, we define the following set-like notation:

$$\begin{aligned} \emptyset &:= x \mapsto 0, \\ \{\!\{x\}\!\} &:= \begin{cases} x \mapsto 1 \\ y \mapsto 0 \textit{ for } y \neq x, \end{cases} \\ \{\!\{x_1, \ldots, x_n\}\!\} &:= \biguplus_{i=1}^{n} \{\!\{x_i\}\!\}. \end{aligned}$$

We define the following set-like notations for element and subset relationship:

$$\begin{aligned} x \in B &\;\Leftrightarrow\; B(x) > 0, \\ B_1 \subseteq B_2 &\;\Leftrightarrow\; \forall x \in B_1.\; B_1(x) \leq B_2(x), \\ B_1 \subset B_2 &\;\Leftrightarrow\; \exists C \subseteq B_1.\; C \neq \emptyset \;\wedge\; B_1 = B_2 \setminus C. \end{aligned}$$

For any bag $B \in Bags(X)$ and predicate $P \subseteq X$ we define the following refinement:

$$\{\!\{x \in B \mid P(x)\}\!\} \;:=\; \begin{cases} x \mapsto B(x) & \textit{if } P(x), \\ x \mapsto 0 & \textit{otherwise.} \end{cases}$$

Definition 12 (Disjoint Union). *Let A, B be sets. We define their disjoint union as*

$$A \sqcup B \;:=\; A \cup B$$

if $A \cap B = \emptyset$ and leave it undefined otherwise.

B Language

Definition 13 (Evaluation of Closed Expressions). *We define a partial evaluation function* $[\![\cdot]\!] : \mathit{Exps} \rightharpoonup \mathit{Values}$ *on expressions by recursion on the structure of expressions as follows:*

$$
\begin{array}{lcll}
[\![v]\!] & := & v & \text{if } v \in \mathit{Values} \\
[\![e = e']\!] & := & \mathsf{True} & \text{if } [\![e]\!] = [\![e']\!] \neq \bot \\
[\![e = e']\!] & := & \mathsf{False} & \text{if } [\![e]\!] \neq [\![e']\!] \ \wedge\ [\![e]\!] \neq \bot \ \wedge\ [\![e']\!] \neq \bot \\
[\![\neg e]\!] & := & \mathsf{False} & \text{if } [\![e]\!] = \mathsf{True} \\
[\![\neg e]\!] & := & \mathsf{True} & \text{if } [\![e]\!] = \mathsf{False} \\
[\![e]\!] & := & \bot & \text{otherwise}
\end{array}
$$

We identify closed expressions e with their ascribed value $[\![e]\!]$.

C Logic

Definition 14. *We define the predicate* $\mathsf{mentionsPerms} \subset \mathcal{A}$ *by recursion on the structure of assertions such that the following holds:*

$$
\begin{array}{lll}
\mathsf{mentionsPerms}(\mathsf{itperm}(\delta)), & & \\
\mathsf{mentionsPerms}(\mathsf{wperm}(id, \delta)), & & \\
\mathsf{mentionsPerms}(\neg a) & \text{if} & \mathsf{mentionsPerms}(a), \\
\mathsf{mentionsPerms}(a_1 \otimes a_2) & \text{if} & \mathsf{mentionsPerms}(a_1) \vee \mathsf{mentionsPerms}(a_2), \\
\mathsf{mentionsPerms}(\bigvee A) & \text{if} & \exists a \in A.\ \mathsf{mentionsPerms}(a), \\
\mathsf{mentionsPerms}(\mathsf{mutex}(m, a)) & \text{if} & \mathsf{mentionsPerms}(a), \\
\mathsf{mentionsPerms}(\mathsf{locked}(m, a, f)) & \text{if} & \mathsf{mentionsPerms}(a),
\end{array}
$$

where $\otimes \in \{\wedge, \vee, *\}$.

Definition 15. *We define the function* $\mathsf{getHLocs}_{\mathsf{lRes}} : \mathcal{R}^{\mathsf{log}} \to \mathcal{L}ocs$ *mapping logical resources to their respective (either empty or singleton) set of involved heap locations as*

$$
\begin{array}{lcll}
\mathsf{getHLocs}_{\mathsf{lRes}}(\ell \mapsto v) & := & \{\ell\}, & \\
\mathsf{getHLocs}_{\mathsf{lRes}}(\mathrm{uninit}_{\mathsf{lRes}}(\ell)) & := & \{\ell\}, & \\
\mathsf{getHLocs}_{\mathsf{lRes}}(\mathrm{mutex}_{\mathsf{lRes}}((\ell, L), a)) & := & \{\ell\}, & \\
\mathsf{getHLocs}_{\mathsf{lRes}}(\mathrm{locked}_{\mathsf{lRes}}((\ell, L), a, f)) & := & \{\ell\}, & \\
\mathsf{getHLocs}_{\mathsf{lRes}}(\_) & := & \emptyset & \text{otherwise.}
\end{array}
$$

Definition 16 (Logical Heap Consistency). *We call a logical heap H* consistent *and write* $\mathsf{consistent}_{\mathsf{lh}}(H)$ *if (i) it contains only full obligations, wait and iteration permission chunks, i.e., if*

$$
\begin{array}{ll}
H(\mathrm{obs}_{\mathsf{lRes}}(O)) & \in \mathbb{N}, \\
H(\mathrm{wperm}_{\mathsf{lRes}}(id, \delta)) & \in \mathbb{N}, \\
H(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) & \in \mathbb{N}
\end{array}
$$

holds for all $O \in \mathit{Bags}(\mathcal{O})$, $\mathit{id} \in \mathcal{ID}$ *and* $\delta \in \Delta$ *and if (ii) heap locations are unique in* H, *i.e., if there are no* $r_1^{\mathsf{I}}, r_2^{\mathsf{I}} \in \mathcal{R}^{\mathsf{log}}$ *with* $r_1^{\mathsf{I}} \neq r_2^{\mathsf{I}}$, $H(r_1^{\mathsf{I}}) > 0$, $H(r_2^{\mathsf{I}}) > 0$ *and with* $\mathsf{getHLocs}_{\mathsf{IRes}}(r_1^{\mathsf{I}}) \cap \mathsf{getHLocs}_{\mathsf{IRes}}(r_2^{\mathsf{I}}) \neq \emptyset$.

Definition 17 (View Shift). *We define a view shift relation* $\Rrightarrow \subset \mathcal{A} \times \mathcal{A}$ *according to the rules presented in Fig 8.*

$$\text{VS-SemImp}\quad \frac{\forall H.\ \mathsf{consistent}_{\mathsf{lh}}(H) \land H \vDash_{\mathsf{A}} A \Rightarrow H \vDash_{\mathsf{A}} B}{A \Rrightarrow B} \qquad \text{VS-Trans}\quad \frac{A \Rrightarrow C \qquad C \Rrightarrow B}{A \Rrightarrow B}$$

$$\text{VS-Or}\quad \frac{A_1 \Rrightarrow B \qquad A_2 \Rrightarrow B}{A_1 \lor A_2 \Rrightarrow B} \qquad \text{VS-NewSignal}\quad \mathsf{obs}(O) \Rrightarrow \exists \mathit{id}.\ \mathsf{obs}(O \uplus \{\!\{(\mathit{id}, L)\}\!\}) * \mathsf{signal}((\mathit{id}, L), \mathsf{False})$$

$$\text{VS-SetSignal}\quad \mathsf{obs}(O \uplus \{\!\{s\}\!\}) * \mathsf{signal}(s, \_) \Rrightarrow \mathsf{obs}(O) * \mathsf{signal}(s, \mathsf{True}) \qquad \text{VS-WaitPerm}\quad \frac{\delta' <_\Delta \delta}{\mathsf{itperm}(\delta) \Rrightarrow \mathsf{wperm}(\mathit{id}, \delta')}$$

$$\text{VS-Wait}\quad \frac{\forall o \in O.\ \mathsf{lev}(s) <_{\mathsf{L}} \mathsf{lev}(o)}{\begin{array}{l}\mathsf{obs}(O) * \mathsf{wperm}(s.\mathsf{id}, \delta) * \mathsf{signal}(s, b) \\ \Rrightarrow \mathsf{obs}(O) * \mathsf{wperm}(s.\mathsf{id}, \delta) * \mathsf{signal}(s, b) * (\neg b \leftrightarrow \mathsf{itperm}(\tau, \delta))\end{array}}$$

$$\text{VS-WeakPerm}\quad \frac{\delta' <_\Delta \delta \qquad N \in \mathbb{N}}{\mathsf{itperm}(\delta) \Rrightarrow \mathop{\Large *}_{1,\ldots,N} \mathsf{itperm}(\delta')} \qquad \text{VS-MutInit}\quad \frac{\neg \mathsf{mentionsPerms}(P)}{\mathsf{uninit}(\ell) * P \Rrightarrow \mathsf{mutex}((\ell, L), P)}$$

Fig. 8: View shift rules.

Definition 18 (Proof Relation). *We define a proof relation* $\vdash \subset \mathcal{A} \times \mathit{Cmds} \times (\mathit{Values} \to \mathcal{A})$ *according to the rules presented in Fig. 9 and 10. We state the provability of a Hoare triple in the form of* $\vdash \{A\}\ c\ \{\lambda r.\, B(r)\}$ *where* r *captures the value returned by* c. *To simplify the notation, we omit the result value if it is clear from the context or irrelevant.*

D Soundness

D.1 Annotated Semantics

Definition 19 (Intermediate Representation). *We define an extended set of commands* Cmds^+ *according to the syntax presented in Fig. 11.*

$$\textsc{PR-Frame}\ \frac{\vdash \{A\}\ c\ \{B\}}{\vdash \{A * F\}\ c\ \{B * F\}} \qquad \textsc{PR-ViewShift}\ \frac{A \Rrightarrow A' \qquad \vdash \{A'\}\ c\ \{B'\} \qquad B' \Rrightarrow B}{\vdash \{A\}\ c\ \{B\}}$$

$$\textsc{PR-Exp}\ \frac{[\![e]\!] \in \mathit{Values}}{\vdash \{\mathsf{True}\}\ e\ \{\lambda r.\, r = [\![e]\!]\}} \qquad \textsc{PR-Exists}\ \frac{\forall a \in A.\ \vdash \{a\}\ c\ \{B\}}{\vdash \{\bigvee A\}\ c\ \{B\}}$$

$$\textsc{PR-Fork}\ \frac{\vdash \{\mathsf{obs}(O_f) * A\}\ c\ \{\mathsf{obs}(\emptyset)\}}{\vdash \{\mathsf{obs}(O_m \uplus O_f) * A\}\ \mathbf{fork}\ c\ \{\lambda r.\, \mathsf{obs}(O_m) * r = \mathsf{tt}\}}$$

(a) Basic Proof Rules.

$$\textsc{PR-If}\ \frac{\vdash \{A\}\ c_b\ \{\lambda b.\, C(b) \wedge (b = \mathsf{True} \vee b = \mathsf{False})\} \quad \vdash \{C(\mathsf{True})\}\ c_t\ \{B\} \qquad C(\mathsf{False}) \Rrightarrow B}{\vdash \{A\}\ \mathbf{if}\ c_b\ \mathbf{then}\ c_t\ \{B\}}$$

$$\textsc{PR-While}\ \frac{\vdash \{I\}\ c_b\ \left\{\begin{array}{l}\lambda b.\ (b = \mathsf{True} \vee b = \mathsf{False}) \\ \quad * (b \to \mathsf{itperm}(\delta) * I) \\ \quad * (\neg b \to B)\end{array}\right\}}{\vdash \{I\}\ \mathbf{while}\ c_b\ \mathbf{do\ skip}\ \{B\}}$$

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

(b) Control Structures.

Fig. 9: Proof rules (part 1).

PR-Acquire

$$\frac{\forall o \in O.\ \mathsf{lev}(m) <_{\mathsf{L}} \mathsf{lev}(o)}{\begin{array}{l}\{\mathsf{obs}(O) * [f]\mathsf{mutex}(m, P)\} \\ \vdash\ \textbf{acquire}\ m.\mathsf{loc} \\ \{\lambda r.\ r = \mathsf{tt} * \mathsf{obs}(O \uplus \{\!|m|\!\}) * \mathsf{locked}(m, P, f) * P\}\end{array}}$$

PR-Release

$$\frac{\mathsf{obs}(O) * A \Rrightarrow \mathsf{obs}(O) * P * B}{\begin{array}{l}\{\mathsf{obs}(O \uplus \{\!|m|\!\}) * \mathsf{locked}(m, P, f) * A\} \\ \vdash\ \textbf{release}\ m.\mathsf{loc} \\ \{\lambda r.\ r = \mathsf{tt} * \mathsf{obs}(O) * [f]\mathsf{mutex}(m, P) * B\}\end{array}}$$

PR-NewMutex

$$\vdash \{\mathsf{True}\}\ \textbf{new\_mutex}\ \{\lambda \ell.\, \mathsf{uninit}(\ell)\}$$

(a) Mutexes.

PR-Cons

$$\vdash \{\mathsf{True}\}\ \textbf{cons}(v)\ \{\lambda \ell.\, \ell \mapsto v\}$$

PR-ReadHeapLoc

$$\vdash \{[f]\ell \mapsto v\}\ [\ell]\ \{\lambda r.\, r = v * [f]\ell \mapsto v\}$$

PR-AssignToHeap

$$\vdash \{\ell \mapsto \_\}\ [\ell] := v\ \{\lambda r.\, r = \mathsf{tt} * \ell \mapsto v\}$$

(b) Heap Access.

Fig. 10: Proof rules (part 2).

$$\begin{array}{lll} c \in \mathit{Cmds}^+ & ::= & e \mid \textbf{while}\ c\ \textbf{do skip} \mid \textbf{fork}\ c \mid \textbf{let}\ x := c\ \textbf{in}\ c \mid \textbf{if}\ c\ \textbf{then}\ c \\ & & \mid \textbf{cons}(e) \mid [e] \mid [e] := e \mid \textbf{new\_mutex} \mid \textbf{acquire}\ e \\ & & \mid \textbf{release}\ e \\ & & \mid \textbf{consumeItPerm} \end{array}$$

Fig. 11: Extended set of commands for intermediate representation.

For the rest of the appendix, commands c refer to the extended set of commands, i.e., $c \in \mathit{Cmds}^{+}$.

Definition 20 (Annotated Resources). *We define the set of annotated resources AnnoRes as*

$$\begin{aligned} r^{\mathsf{a}} \in \mathit{AnnoRes} \;::=\; & \ell \mapsto v \mid \mathrm{uninit}_{\mathsf{aRes}}(\ell) \mid \\ & \mathrm{unlocked}_{\mathsf{aRes}}((\ell, L), a, H) \mid \mathrm{locked}_{\mathsf{aRes}}((\ell, L), a, f) \mid \\ & \mathrm{signal}_{\mathsf{aRes}}((id, L), b) \end{aligned}$$

where H does not contain any obligations chunks.

Definition 21 (Annotated Heaps). *We define the set of annotated heaps as*

$$\mathit{Heaps}^{\mathsf{annot}} \;:=\; \mathcal{P}_{\mathsf{fin}}(\mathit{AnnoRes}),$$

the function $\mathsf{locs}_{\mathsf{ah}} : \mathit{Heaps}^{\mathsf{annot}} \to \mathcal{P}_{\mathsf{fin}}(\mathcal{L}ocs)$ mapping annotated heaps to the sets of allocated heap locations as

$$\begin{aligned} \mathsf{locs}_{\mathsf{ah}}(h^{\mathsf{a}}) \;:=\; \{\ell \in \mathcal{L}ocs \mid\; & \exists v \in \mathit{Values}.\ \exists L \in \mathcal{L}evs.\ \exists a \in \mathcal{A}. \\ & \exists H \in \mathit{Heaps}^{\mathsf{log}}.\ \exists f \in \mathcal{F}. \\ & \quad \ell \mapsto v \in h^{\mathsf{a}} \;\vee\; \mathrm{uninit}_{\mathsf{aRes}}(\ell) \in h^{\mathsf{a}} \;\vee \\ & \quad \mathrm{unlocked}_{\mathsf{aRes}}((\ell, L), a, H) \in h^{\mathsf{a}} \;\vee \\ & \quad \mathrm{locked}_{\mathsf{aRes}}((\ell, L), a, f) \in h^{\mathsf{a}}\} \end{aligned}$$

and the function $\mathsf{ids}_{\mathsf{ah}} : \mathit{Heaps}^{\mathsf{annot}} \to \mathcal{P}_{\mathsf{fin}}(\mathcal{ID})$ mapping annotated heaps to sets of allocated signal IDs as

$$\mathsf{ids}_{\mathsf{ah}}(h^{\mathsf{a}}) := \{id \in \mathcal{ID} \mid \exists L \in \mathcal{L}evs.\ \exists b \in \mathbb{B}.\ \mathrm{signal}_{\mathsf{aRes}}((id, L), b) \in h^{\mathsf{a}}\}.$$

We denote annotated heaps by h^{a}.

We call an annotated heap h^{a} finite *and write $\mathsf{finite}_{\mathsf{ah}}(h^{\mathsf{a}})$ if there exists no chunk $\mathrm{unlocked}_{\mathsf{aRes}}((\ell, L), a, H) \in h^{\mathsf{a}}$ for which $\mathsf{finite}_{\mathsf{lh}}(H)$ does not hold.*

Definition 22 (Compatibility of Annotated and Physical Heaps). *We inductively define a relation ${}_{\mathsf{ah}}{\sim}_{\mathsf{ph}} \subset \mathit{Heaps}^{\mathsf{annot}} \times \mathcal{R}^{\mathsf{phys}}$ between annotated and physical heaps such that the following holds:*

$$\begin{array}{ll} \emptyset & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ \emptyset, \\ \ell \mapsto v \;\cup\; h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ \ell \mapsto v \;\cup\; h, \\ \mathrm{uninit}_{\mathsf{aRes}}(\ell) \;\cup\; h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ \mathrm{unlocked}_{\mathsf{pRes}}(\ell) \;\cup\; h, \\ \mathrm{unlocked}_{\mathsf{aRes}}((\ell, L), P, H_P) \;\cup\; h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ \mathrm{unlocked}_{\mathsf{pRes}}(\ell) \;\cup\; h, \\ \mathrm{locked}_{\mathsf{aRes}}((\ell, L), P, f) \;\cup\; h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ \mathrm{locked}_{\mathsf{pRes}}(\ell) \;\cup\; h, \\ \mathrm{signal}_{\mathsf{aRes}}(s, b) \;\cup\; h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h, \end{array}$$

where $h^{\mathsf{a}} \in \mathit{Heaps}^{\mathsf{annot}}$ and $h \in \mathit{Heaps}^{\mathsf{phys}}$ are annotated and physical heaps with $h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{ph}}\ h$.

Definition 23 (Compatibility of Annotated and Logical Heaps). *We inductively define a relation* $\ {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}} \subset \mathit{Heaps}^{\mathsf{annot}} \times \mathit{Heaps}^{\mathsf{log}}$ *between annotated and logical heaps such that the following holds:*

$$
\begin{array}{ll}
\emptyset & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ \emptyset_{\mathsf{log}}, \\
h^{\mathsf{a}} \cup \{\ell \mapsto v\} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\ell \mapsto v\}, \\
h^{\mathsf{a}} \cup \{\mathrm{uninit}_{\mathsf{aRes}}(\ell)\} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{uninit}_{\mathsf{lRes}}(\ell)\}, \\
h^{\mathsf{a}} \cup \{\mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P)\} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{mutex}_{\mathsf{lRes}}(m, P)\} + H_P, \\
h^{\mathsf{a}} \cup \{\mathrm{locked}_{\mathsf{aRes}}(m, P, f)\} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{locked}_{\mathsf{lRes}}(m, P, f)\} \\
 & \qquad + (1-f) \cdot \{\mathrm{mutex}_{\mathsf{lRes}}(m, P)\}, \\
h^{\mathsf{a}} \cup \{\mathrm{signal}_{\mathsf{aRes}}(s, b)\} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{signal}_{\mathsf{lRes}}(s, b)\}, \\
h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{obs}_{\mathsf{lRes}}(O)\}, \\
h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{wperm}_{\mathsf{lRes}}(id, \delta)\}, \\
h^{\mathsf{a}} & {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},
\end{array}
$$

where $h^{\mathsf{a}} \in \mathit{Heaps}^{\mathsf{annot}}$ *and* $H \in \mathit{Heaps}^{\mathsf{log}}$ *are annotated and logical heaps with* $\ell, m.\mathsf{loc} \notin \mathsf{locs}_{\mathsf{ah}}(h^{\mathsf{a}})$, $s.\mathsf{id} \notin \mathsf{ids}_{\mathsf{ah}}(h^{\mathsf{a}})$ *and* $h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ H$.

Definition 24 (Annotated Single Thread Reduction Relation). *We define a reduction relation* $\rightsquigarrow_{\mathsf{ast}}$ *for annotated threads according to the rules presented in Fig. 12. A reduction step has the form*

$$h^{\mathsf{a}}, H, c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}\prime}, H', c', T^{\mathsf{a}}$$

for a set of annotated forked threads $T^{\mathsf{a}} \subset \mathit{Heaps}^{\mathsf{log}} \times \mathit{Cmds}$ *with* $|T^{\mathsf{a}}| \leq 1$.

It indicates that given annotated heap h^{a} *and a logical heap* H*, command* c *can be reduced to annotated heap* $h^{\mathsf{a}\prime}$*, logical heap* H' *and command* c'*. The either empty or singleton set* T^{a} *represents whether a new thread is forked in this step.*

For simplicity of notation we omit T^{a} *if it is clear from the context that no thread is forked and* $T^{\mathsf{a}} = \emptyset$.

Definition 25 (Annotated Thread Pools). *We define the set of annotated thread pools* $\mathcal{TP}^{\mathsf{a}}$ *as the set of finite partial functions mapping thread IDs to annotated threads:*

$$\mathcal{TP}^{\mathsf{a}} \ := \ \Theta \rightharpoonup_{\mathsf{fin}} \mathit{Heaps}^{\mathsf{log}} \times (\mathit{Cmds} \cup \{\mathsf{term}\}).$$

We denote annotated thread pools by P^{a} *and the empty thread pool by* $\emptyset_{\mathsf{atp}}$*, i.e.,*

$$
\begin{array}{l}
\emptyset_{\mathsf{atp}} \ : \ \Theta \rightharpoonup_{\mathsf{fin}} \mathit{Heaps}^{\mathsf{log}} \times (\mathit{Cmds} \cup \{\mathsf{term}\}), \\
\mathsf{dom}(\emptyset_{\mathsf{atp}}) \ = \ \emptyset.
\end{array}
$$

We define the extension operation $+_{\mathsf{atp}}$ *analogously to* $+_{\mathsf{tp}}$*, cf. Definition 4d.*

For convenience of notation we define selector functions for annotated threads as

$$
\begin{array}{l}
(H, c).\mathsf{heap} := H, \\
(H, c).\mathsf{cmd} \ := c.
\end{array}
$$

$$\textsc{AST-Red-EvalCtxt}\ \frac{h^{\mathsf{a}}, H, c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}\prime}, H', c', T}{h^{\mathsf{a}}, H, E[c] \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}\prime}, H', E[c'], T}$$

$$\textsc{AST-Red-Fork}\ h^{\mathsf{a}}, H_m + \{\mathrm{obs}_{\mathsf{lRes}}(O_m \uplus O_f)\} + H_f, \mathbf{fork}\ c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H_m + \{\mathrm{obs}_{\mathsf{lRes}}(O_m)\}, \mathsf{tt}, \{(\{\mathrm{obs}_{\mathsf{lRes}}(O_f)\} + H_f),\ c)\}$$

(a) Basic constructs.

$$\textsc{AST-Red-While}\ h^{\mathsf{a}}, H, \mathbf{while}\ c\ \mathbf{do\ skip} \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{consumeItPerm}; \mathbf{while}\ c\ \mathbf{do\ skip})$$

$$\textsc{AST-Red-IfTrue}\ h^{\mathsf{a}}, H, \mathbf{if}\ \mathsf{True}\ \mathbf{then}\ c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, c$$

$$\textsc{AST-Red-IfFalse}\ h^{\mathsf{a}}, H, \mathbf{if}\ \mathsf{False}\ \mathbf{then}\ c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, \mathsf{tt}$$

$$\textsc{AST-Red-ConsumeItPerm}\ h^{\mathsf{a}}, H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\}, \mathbf{consumeItPerm} \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, \mathsf{tt}$$

$$\textsc{AST-Red-Let}\ h^{\mathsf{a}}, H, \mathbf{let}\ x := v\ \mathbf{in}\ c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, c[v/x]$$

(b) Control structures.

$$\textsc{AST-Red-Cons}\ \frac{\ell \notin \mathsf{locs}_{\mathsf{ah}}(h^{\mathsf{a}})}{h^{\mathsf{a}}, H, \mathbf{cons}(v) \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}} \cup \{\ell \mapsto v\}, H + \{\ell \mapsto v\}, \ell}$$

$$\textsc{AST-Red-ReadHeapLoc}\ \frac{\ell \mapsto v \in h^{\mathsf{a}}}{h^{\mathsf{a}}, H, [\ell] \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}}, H, v}$$

$$\textsc{AST-Red-Assign}\ h \sqcup \{\ell \mapsto v\}, H + \{\ell \mapsto v\}, [\ell] := v \rightsquigarrow_{\mathsf{ast}} h \sqcup \{\ell \mapsto v'\}, H + \{\ell \mapsto v'\}, \mathsf{tt}$$

(c) Heap access.

$$\textsc{AST-Red-NewMutex}\ \frac{\ell \notin \mathsf{locs}_{\mathsf{ah}}(h^{\mathsf{a}})}{h^{\mathsf{a}}, H, \mathbf{new\_mutex} \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}} \cup \{\mathrm{uninit}_{\mathsf{aRes}}(\ell)\}, H + \{\mathrm{uninit}_{\mathsf{lRes}}(\ell)\}, \ell}$$

$$\textsc{AST-Red-Acquire}\ \frac{f \in \mathcal{F} \qquad \forall o \in O.\ \mathsf{lev}(m) <_{\mathsf{L}} \mathsf{lev}(o)}{\begin{array}{l} h^{\mathsf{a}} \sqcup \{\mathrm{unlocked}_{\mathsf{aRes}}(m, a, H_P)\},\ H + \{\mathrm{obs}_{\mathsf{lRes}}(O)\} + f \cdot \{\mathrm{mutex}_{\mathsf{lRes}}(m, P)\}, \\ \mathbf{acquire}\ m.\mathsf{loc} \\ \rightsquigarrow_{\mathsf{ast}}\ h^{\mathsf{a}} \sqcup \{\mathrm{locked}_{\mathsf{aRes}}(m, P, f)\},\ H + \{\mathrm{obs}_{\mathsf{lRes}}(O \uplus \{\!\{m\}\!\}),\ \mathrm{locked}_{\mathsf{lRes}}(m, P, f)\} + H_P, \\ \quad \mathsf{tt} \end{array}}$$

$$\textsc{AST-Red-Release}\ \frac{H_P \vDash_{\mathsf{A}} P \qquad \mathsf{consistent}_{\mathsf{lh}}(H_P) \qquad \exists O.\ H(\mathrm{obs}_{\mathsf{lRes}}(O)) \geq 1 \qquad \neg\exists \delta, id.\ (H_P(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) > 0\ \vee\ H_P(\mathrm{wperm}_{\mathsf{lRes}}(id, \delta)) > 0)}{\begin{array}{l} h^{\mathsf{a}} \sqcup \{\mathrm{locked}_{\mathsf{aRes}}(m, P, f)\},\ H + \{\mathrm{obs}_{\mathsf{lRes}}(O \uplus \{\!\{m\}\!\}),\ \mathrm{locked}_{\mathsf{lRes}}(m, P, f)\} + H_P, \\ \mathbf{release}\ m.\mathsf{loc} \\ \rightsquigarrow_{\mathsf{ast}}\ h^{\mathsf{a}} \sqcup \{\mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P)\},\ H + \{\mathrm{obs}_{\mathsf{lRes}}(O)\} + f \cdot \{\mathrm{mutex}_{\mathsf{lRes}}(m, P)\}, \\ \quad \mathsf{tt} \end{array}}$$

(d) Mutexes.

Fig. 12: Annotated single thread reduction rules.

Definition 26 (Ghost Reduction Relation). *We define a thread pool reduction relation $\rightsquigarrow_{\mathsf{ghost}}$ according to the rules presented in Fig. 13 to express ghost steps. A ghost reduction step has the form*

$$h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}\prime}, P^{\mathsf{a}\prime}.$$

We denote its reflexive transitive closure by $\rightsquigarrow^*_{\mathsf{ghost}}$.

$$\text{GTP-Red-NewSignal}\quad \frac{P^{\mathsf{a}}(\theta) = (H + \{\mathrm{obs}_{\mathsf{lRes}}(O)\},\, c) \qquad id \notin \mathsf{ids}_{\mathsf{ah}}(h^{\mathsf{a}}) \qquad H' = H + \{\mathrm{signal}_{\mathsf{lRes}}((id, L), \mathsf{False}),\, \mathrm{obs}_{\mathsf{lRes}}(O \uplus \{\!\{id, L\}\!\})\}}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}} \cup \{\mathrm{signal}_{\mathsf{aRes}}((id, L), \mathsf{False})\},\, P^{\mathsf{a}}[\theta := (H', c)]}$$

$$\text{GTP-Red-SetSignal}\quad \frac{P^{\mathsf{a}}(\theta) = (H + \{\mathrm{signal}_{\mathsf{lRes}}(s, \mathsf{False}),\, \mathrm{obs}_{\mathsf{lRes}}(O \uplus \{\!\{s\}\!\})\},\, c) \qquad H' = H + \{\mathrm{signal}_{\mathsf{lRes}}(s, \mathsf{False}),\, \mathrm{obs}_{\mathsf{lRes}}(O)\}}{h^{\mathsf{a}} \sqcup \{\mathrm{signal}_{\mathsf{aRes}}(s, \mathsf{False})\}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}} \sqcup \{\mathrm{signal}_{\mathsf{aRes}}(s, \mathsf{True})\},\, P^{\mathsf{a}}[\theta := (H', c)]}$$

$$\text{GTP-Red-WaitPerm}\quad \frac{\delta' <_{\Delta} \delta \qquad P^{\mathsf{a}}(\theta) = (H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},\, c)}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}}, P^{\mathsf{a}}[\theta := (H + \{\mathrm{wperm}_{\mathsf{lRes}}(id, \delta')\},\, c)]}$$

$$\text{GTP-Red-Wait}\quad \frac{\mathrm{signal}_{\mathsf{aRes}}(s, \mathsf{False}) \in h^{\mathsf{a}} \qquad P^{\mathsf{a}}(\theta) = (H, c) \qquad H(\mathrm{wperm}_{\mathsf{lRes}}(s.\mathsf{id}, \delta)) \geq 1 \qquad H(\mathrm{obs}_{\mathsf{lRes}}(O)) \geq 1 \qquad \forall o \in O.\, \mathsf{lev}(s) <_{\mathsf{L}} \mathsf{lev}(O)}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}}, P^{\mathsf{a}}[\theta := (H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},\, c)]}$$

$$\text{GTP-Red-WeakItPerm}\quad \frac{\delta' <_{\Delta} \delta \qquad N \in \mathbb{N} \qquad P^{\mathsf{a}}(\theta) = (H + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},\, c)}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}}, P^{\mathsf{a}}[\theta := (H + N \cdot \{\mathrm{itperm}_{\mathsf{lRes}}(\delta')\},\, c)]}$$

$$\text{GTP-Red-MutInit}\quad \frac{P^{\mathsf{a}}(\theta) = (H + \{\mathrm{uninit}_{\mathsf{lRes}}(\ell)\} + H_P,\, c) \qquad H' = H + \{\mathrm{mutex}_{\mathsf{lRes}}((\ell, L), H_P)\} \qquad H_P \vDash_{\mathsf{A}} P \qquad \mathsf{consistent}_{\mathsf{lh}}(H_P) \qquad \exists O.\, H(\mathrm{obs}_{\mathsf{lRes}}(O)) \geq 1 \qquad \neg\exists \delta, id.\, (H_P(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) > 0 \;\vee\; H_P(\mathrm{wperm}_{\mathsf{lRes}}(id, \delta)) > 0)}{h^{\mathsf{a}} \sqcup \{\mathrm{uninit}_{\mathsf{aRes}}(\ell)\}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{ghost}} h^{\mathsf{a}} \sqcup \{\mathrm{unlocked}_{\mathsf{aRes}}((\ell, L), a, H_P)\},\, P^{\mathsf{a}}[\theta := (H', c)]}$$

Fig. 13: Ghost thread pool reduction rules.

Definition 27 (Non-ghost Thread Pool Reduction Relation). *We define a thread pool reduction relation $\rightsquigarrow_{\mathsf{real}}$ according to the rules presented in Fig. 14 to express real (i.e. non-ghost) reduction steps. A reduction step has the form*

$$h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{real}} h^{\mathsf{a}\prime}, P^{\mathsf{a}\prime}.$$

$$\text{RTP-Red-Lift}\quad \frac{\theta_f = \min(\Theta \setminus \mathsf{dom}(P^{\mathsf{a}}))\qquad P^{\mathsf{a}}(\theta) = (H, c)\qquad h^{\mathsf{a}}, H, c \rightsquigarrow_{\mathsf{ast}} h^{\mathsf{a}\prime}, H', c', T^{\mathsf{a}}}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{real}} h^{\mathsf{a}\prime}, P^{\mathsf{a}}[\theta := (H', c')] +_{\mathsf{atp}} T^{\mathsf{a}}}$$

$$\text{RTP-Red-Term}\quad \frac{P^{\mathsf{a}}(\theta) = (H, v)\qquad H.\mathsf{obs} = \emptyset}{h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{real}} h^{\mathsf{a}}, P^{\mathsf{a}} -_{\mathsf{atp}} \theta}$$

Fig. 14: Non-ghost thread pool reduction rules.

Definition 28 (Annotated Thread Pool Reduction Relation). *We define the annotated thread pool reduction relation* $\rightsquigarrow_{\mathsf{atp}}$ *as*

$$\rightsquigarrow_{\mathsf{atp}} \;:=\; \rightsquigarrow_{\mathsf{ghost}} \cup \rightsquigarrow_{\mathsf{real}} .$$

Definition 29 (Annotated Reduction Sequence). *Let* $(h^{\mathsf{a}}{}_i)_{i\in\mathbb{N}}$ *and* $(P^{\mathsf{a}}{}_i)_{i\in\mathbb{N}}$ *be infinite sequences of annotated heaps and annotated thread pools, respectively. Let* $\mathsf{sig} : \mathbb{N} \rightharpoonup \mathcal{S}$ *be a partial function mapping indices to signals.*

We call $((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig})$ *an* annotated reduction sequence *if there exists a sequence of thread IDs* $(\theta_i)_{i\in\mathbb{N}}$ *such that the following holds for every* $i \in \mathbb{N}$*:*

- $h_i^{\mathsf{a}}, P_i^{\mathsf{a}} \overset{\theta_i}{\rightsquigarrow}_{\mathsf{atp}} h_{i+1}^{\mathsf{a}}, P_{i+1}^{\mathsf{a}}$
- *If this reduction step results from an application of* GTP-Red-Wait *to some signal* s*, then* $\mathsf{sig}(i) = s$ *holds and otherwise* $\mathsf{sig}(i) = \bot$*.*

In case the signal annotation sig *is clear from the context or not relevant, we omit it and write* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *instead of* $((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig})$*.*

We call $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})$ *an* annotated machine configuration*.*

Lemma 6 (Preservation of Finiteness). *Let* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *be an annotated reduction sequence with* $\mathsf{finite}_{\mathsf{ah}}(h_0^{\mathsf{a}})$ *and* $\mathsf{finite}_{\mathsf{lh}}(P_0^{\mathsf{a}}(\theta).\mathsf{heap})$ *for all* $\theta \in \mathsf{dom}(P_0^{\mathsf{a}})$*.*

Then, $\mathsf{finite}_{\mathsf{lh}}(P_i^{\mathsf{a}}(\theta).\mathsf{heap})$ *holds for all* $i \in \mathbb{N}$ *and all* $\theta \in \mathsf{dom}(P_i^{\mathsf{a}})$*.*

Proof. Proof by induction on i.

Lemma 7 (Preservation of Completeness). *Let* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *be an annotated reduction sequence with* $\mathsf{complete}_{\mathsf{lh}}(P_0^{\mathsf{a}}(\theta).\mathsf{heap})$ *for all* $\theta \in \mathsf{dom}(P_0^{\mathsf{a}})$*. Then,* $\mathsf{complete}_{\mathsf{lh}}(P_i^{\mathsf{a}}(\theta).\mathsf{heap})$ *holds for every* $i \in \mathbb{N}$ *and every* $\theta \in \mathsf{dom}(P_i^{\mathsf{a}})$*.*

Proof. Proof by induction on i.

Every thread of an annotated thread pool is annotated by a thread-local logical heap that expresses which resources are owned by this thread. In the following we define a function to extract the logical heap expressing which resources are owned by threads of a thread pool (i.e. the sum of all thread-local logical heaps).

Definition 30. *We define the function* $\mathsf{ownedResHeap}_{\mathsf{atp}} : \mathcal{TP}^{\mathsf{a}} \to \mathit{Heaps}^{\mathsf{log}}$ *mapping annotated thread pools to logical heaps as follows:*

$$P^{\mathsf{a}} \mapsto \sum_{\theta \in \mathsf{dom}(P^{\mathsf{a}})} P^{\mathsf{a}}(\theta).\mathsf{heap}$$

Annotated resources representing unlocked locks, i.e., $\mathrm{unlocked}_{\mathsf{aRes}}(m, a, H_a)$, contain a logical heap H_a that expresses which resources are protected by this lock. In the following, we define a function that extracts a logical heap from an annotated heap h^{a} expressing which resources are protected by unlocked locks in h^{a}.

Definition 31. *We define the function* $\mathsf{protectedResHeap}_{\mathsf{ah}} : \mathit{Heaps}^{\mathsf{annot}} \to \mathit{Heaps}^{\mathsf{log}}$ *mapping annotated heaps to logical heaps as follows:*

For any annotated heap h^{a} *let*

$$\mathit{LockInvs}(h^{\mathsf{a}}) := \{\!|H_P \in \mathit{Heaps}^{\mathsf{log}} \mid \exists m \in \mathcal{L}ocs \times \mathcal{L}evs.\ \exists P \in \mathcal{A}.\ \mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P) \in h^{\mathsf{a}}|\!\}$$

be the auxiliary set aggregating all logical heaps corresponding to lock invariants of unlocked locks stored in h^{a}*. We define* $\mathsf{protectedResHeap}_{\mathsf{ah}}$ *as*

$$h^{\mathsf{a}} \mapsto \sum_{H_P \in \mathit{LockInvs}(h^{\mathsf{a}})} H_P.$$

We consider a machine configuration $(h^{\mathsf{a}}, P^{\mathsf{a}})$ to be *consistent* if it fulfils three criteria: (i) Every thread-local logical heap is consistent, i.e., for all used thread IDs θ, $P^{\mathsf{a}}(\theta).\mathsf{heap}$ only stores full obligations, wait permission and iteration permission chunks. (ii) Every logical heap protected by an unlocked lock in h^{a} is consistent. (iii) h^{a} is compatible with the logical heap that represents (a) the resources owned by threads in P^{a} and (b) the resources protected by unlocked locks stored in h^{a}.

Definition 32 (Consistency of Annotated Machine Configurations). *We call an annotated machine configuration* $(h^{\mathsf{a}}, P^{\mathsf{a}})$ consistent *and write* $\mathsf{consistent}_{\mathsf{conf}}(h^{\mathsf{a}}, P^{\mathsf{a}})$ *if all of the following hold:*

- $\mathsf{consistent}_{\mathsf{lh}}(P^{\mathsf{a}}(\theta).\mathsf{heap})$ *for all* $\theta \in \mathsf{dom}(P^{\mathsf{a}})$,
- $\forall m.\ \forall P.\ \forall H_P.\ \mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P) \in h^{\mathsf{a}} \to \mathsf{consistent}_{\mathsf{lh}}(H_P)$,
- $h^{\mathsf{a}}\ {}_{\mathsf{ah}}{\sim}_{\mathsf{lh}}\ \mathsf{ownedResHeap}_{\mathsf{atp}}(P^{\mathsf{a}}) + \mathsf{protectedResHeap}_{\mathsf{ah}}(h^{\mathsf{a}})$.

Lemma 8 (Preservation of Consistency). *Let* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i \in \mathbb{N}}$ *be an annotated reduction sequence with* $\mathsf{consistent}_{\mathsf{conf}}(h_0^{\mathsf{a}}, P_0^{\mathsf{a}})$*. Then,* $\mathsf{consistent}_{\mathsf{conf}}(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})$ *holds for every* $i \in \mathbb{N}$*.*

Proof. Proof by induction on i.

Definition 33 (Command Annotation). *We define the predicate* $\mathsf{annot}_{\mathsf{cmd}} \subset \mathit{Cmds} \times \mathit{Cmds}$ *such that* $\mathsf{annot}_{\mathsf{cmd}}(c', c)$ *holds iff* c' *results from* c *by removing all occurrences of* **consumeItPerm**.

Definition 34 (Thread Pool Annotation). *We define a predicate* $\mathsf{annot}_{\mathsf{tp}} \subset \mathcal{TP}^{\mathsf{a}} \times \mathcal{TP}$ *such that:*

$$\mathsf{annot}_{\mathsf{tp}}(P^{\mathsf{a}}, P) \iff \mathsf{dom}(P^{\mathsf{a}}) = \mathsf{dom}(P) \ \wedge \ \forall \theta \in \mathsf{dom}(P).\ \mathsf{annot}_{\mathsf{cmd}}(P^{\mathsf{a}}(\theta).\mathsf{cmd}, P(\theta))$$

D.2 Soundness Proof

In this subsection we present detailed versions of proof sketches presented in § 5.

Definition 35 (Program Order Graph).

Let $((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig})$ *be an annotated reduction sequence. Let* N^r *be the set of names referring to reduction rules defining the relations* $\rightsquigarrow_{\mathsf{real}}$, $\rightsquigarrow_{\mathsf{ghost}}$ *and* $\rightsquigarrow_{\mathsf{ast}}$. *We define the set of annotated reduction rule names* N^{a} *where* GTP-Red-Wait *is annotated by signals as*

$$N^{\mathsf{a}} \ := \ (N^r \setminus \{\text{GTP-Red-Wait}\}) \cup (\{\text{GTP-Red-Wait}\} \times \mathcal{S}).$$

We define the program order graph $G(((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig})) = (\mathbb{N}, E)$ *with root 0 where* $E \subset \mathbb{N} \times \Theta \times N^{\mathsf{a}} \times \mathbb{N}$.

A node $a \in \mathbb{N}$ *corresponds to the sequence's* a^{th} *reduction step, i.e., to the step* $h_a^{\mathsf{a}}, P_a^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h_{a+1}^{\mathsf{a}}, P_{a+1}^{\mathsf{a}}$ *for some* $\theta \in \mathsf{dom}(P_a^{\mathsf{a}})$. *An edge from node* a *to node* b *expresses that the* b^{th} *reduction step continues the control flow of step* a. *For any* $\ell \in \mathbb{N}$, *let* θ_ℓ *denote the ID of the thread reduced in step* ℓ. *Furthermore, let* n_ℓ^{a} *denote the name of the reduction rule applied in the* ℓ^{th} *step, in the following sense:*

- *If* $h_\ell^{\mathsf{a}}, P_\ell^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h_{\ell+1}^{\mathsf{a}}, P_{\ell+1}^{\mathsf{a}}$ *results from an application of* RTP-Red-Lift *in combination with single-thread reduction rule* n^{st}, *then* $n_\ell^{\mathsf{a}} = n^{\mathsf{st}}$.
- *If* $h_\ell^{\mathsf{a}}, P_\ell^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h_{\ell+1}^{\mathsf{a}}, P_{\ell+1}^{\mathsf{a}}$ *results from an application of* GTP-Red-Wait, *then* $n_\ell^{\mathsf{a}} = (\text{GTP-Red-Wait}, \mathsf{sig}(\ell))$.
- *Otherwise,* n^{a} *denotes the applied (real or ghost) thread pool reduction rule.*

An edge $(a, \theta, n^{\mathsf{a}}, b) \in \mathbb{N} \times \Theta \times N^{\mathsf{a}} \times \mathbb{N}$ *is contained in* E *if* $n^{\mathsf{a}} = n_a^{\mathsf{a}}$ *and one of the following conditions applies:*

- $\theta = \theta_a = \theta_b$ *and* $b = \min(\{k > a \mid h_k^{\mathsf{a}}, P_k^{\mathsf{a}} \overset{\theta_a}{\rightsquigarrow}_{\mathsf{atp}} h_{k+1}^{\mathsf{a}}, P_{k+1}^{\mathsf{a}}\})$.
  In this case, the edge expresses that step b *marks the first time that thread* θ_a *is rescheduled for reduction (after step* a*).*
- $\mathsf{dom}(P_{a+1}^{\mathsf{a}}) \setminus \mathsf{dom}(P_a^{\mathsf{a}}) = \{\theta\}$ *and*
  $b = \min\{k \in \mathbb{N} \mid h_k^{\mathsf{a}}, P_k^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h_{k+1}^{\mathsf{a}}, P_{k+1}^{\mathsf{a}}\}$.
  In this case, θ *identifies the thread forked in step* a. *The edge expresses that step* b *marks the first reduction of the forked thread.*

In case the choice of reduction sequence $((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig})$ *is clear from the context, we write* G *instead of* $G(((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}, \mathsf{sig}))$.

Observation 1. *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ be an annotated reduction sequence with $|\mathsf{dom}(P_0^{\mathsf{a}})| = 1$. The sequence's program order graph $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$ is a binary tree.*

For any reduction sequence $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$, the paths in its program order graph $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$ represent the sequence's control flow paths. Hence, we are going to use program order graphs to analyse reduction sequences' control flows.

We refer to a program order graph's edges by the kind of reduction step they represent. For instance, we call edges of the form $(a, \theta, \text{ST-RED-WHILE}, b)$ *loop edges* because they represent a loop backjump and we call edges of the form $(a, \theta, (\text{GTP-RED-WAIT}, s), b)$ *wait edges*. Any wait edge of this form represents an application of GTP-RED-WAIT to signal s.

In the following, we prove that any path in a program order graph that does not involve a loop edge is finite. This follows from the fact that the size of the command reduced along this path decreases with each non-ghost non-loop step.

Lemma 9. *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ be a fair annotated reduction sequence. Let $p = (V, E)$ be a path in $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$. Let $L = \{e \in E \mid \pi_3(e) = \text{AST-RED-WHILE}\}$ be the set of loop edges contained in p. Then, p is infinite if and only if L is infinite.*

Proof. If L is infinite, p is obviously infinite as well. So, suppose L is finite.

For any command, we consider its size to be the number of nodes contained in its abstract syntax tree. By structural induction over the set of commands, it follows that the size of a command $c = P^{\mathsf{a}}(\theta).\mathsf{cmd}$ decreases in every non-ghost reduction step $h^{\mathsf{a}}, P^{\mathsf{a}} \overset{\theta}{\rightsquigarrow}_{\mathsf{atp}} h^{\mathsf{a}\prime}, P^{\mathsf{a}\prime}$ that is not an application of RTP-RED-LIFT in combination with AST-RED-WHILE.

Since L is finite, there exists a node x such that the suffix $p_{\geq x}$ starting at node x does not contain any loop edges. By fairness of $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$, every non-empty suffix of $p_{\geq x}$ contains an edge corresponding to a non-ghost reduction step. For any edge $e = (i, \theta, n, j)$ consider the command $c_e = P_i^{\mathsf{a}}(\theta).\mathsf{cmd}$ reduced in this edge. The size of these commands decreases along $p_{\geq x}$. So, $p_{\geq x}$ must be finite and thus p must be finite as well.

Corollary 1. *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ be a fair annotated reduction sequence. Let $p = (V, E)$ be a path in $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$. Let*

$$C = \{e \in E \mid \pi_3(e) = \text{AST-RED-CONSUMEITPERM}\}$$

be the set of consume edges contained in p. Then, p is infinite if and only if C is infinite.

Proof. Follows from Lemma 9 by the fact that the set $\{e \in E \mid \pi_3(e) = \text{AST-RED-WHILE}\}$ is infinite if and only if C is infinite.

Definition 36. *Let $G = (V, E)$ be a subgraph of some program order graph. We define the function $\mathsf{waitEdges}_G : \mathcal{S} \to \mathcal{P}(E)$ mapping any signal s to the set of wait edges in G concerning s as:*

$$\mathsf{waitEdges}_G(s) \;:=\; \{(a, \theta, (\text{GTP-RED-WAIT}, s'), b) \in E \mid s' = s\}.$$

Furthermore, we define the set $\mathcal{S}_G \subset \mathcal{S}$ of signals being waited for in G and its subset $S_G^\infty \subseteq \mathcal{S}_G$ of signals waited-for infinitely often in G as follows:

$$\begin{aligned} \mathcal{S}_G &:= \{s \in \mathcal{S} \mid \mathsf{waitEdges}_G(s) \neq \emptyset\}, \\ S_G^\infty &:= \{s^\infty \in \mathcal{S}_G \mid \mathsf{waitEdges}_G(s^\infty) \textit{ infinite}\}. \end{aligned}$$

Definition 37. *Let $(h_i^\mathsf{a}, P_i^\mathsf{a})_{i\in\mathbb{N}}$ be a fair annotated reduction sequence and let $G = (V, E)$ be a subgraph of the sequence's program order graph. We define the function $\mathsf{itperms}_G : E \to \mathit{Bags}_\mathsf{fin}(\Lambda)$ mapping any edge e to the (potentially empty) finite bag of iteration permissions derived in the reduction step corresponding to e as follows:*

Let $(i, \theta, n, j) \in E$ be an edge.

- *If $n = (\text{GTP-Red-Wait}, s)$ for some signal $s \in \mathcal{S}$, then the i^{th} reduction step spawns a single iteration permission of degree δ, i.e.,*
  $P_{i+1}^\mathsf{a} = P_i^\mathsf{a}[\theta := (P_i^\mathsf{a}(\theta).\mathsf{heap} + \{\mathrm{itperm}_{\mathsf{lRes}}(\delta)\},\ P_i^\mathsf{a}(\theta).\mathsf{cmd})]$.
  In this case, we define

$$\mathsf{itperms}_G((i, \theta, (\text{GTP-Red-Wait}, s), j)) := \{\!\{\delta\}\!\}.$$

- *If $n = \text{GTP-Red-WeakItPerm}$, then the i^{th} reduction step consumes an iteration permission of degree δ and produces N permissions of a lower degree δ', i.e., $P_i^\mathsf{a}(\theta).\mathsf{heap} = H + \{\mathsf{itperm}(\delta)\}$ for some heap H and $P_{i+1}^\mathsf{a} = P_i^\mathsf{a}[\theta := (H',\ P_i^\mathsf{a}(\theta).\mathsf{cmd})]$ for*

$$H' = H + N \cdot \{\mathrm{itperm}_{\mathsf{lRes}}(\delta')\}.$$

  In this case, we define

$$\mathsf{itperms}_G((i, \theta, \text{GTP-Red-WeakItPerm}, j)) := \{\!\{\underbrace{\delta', \ldots, \delta'}_{N \textit{ times}}\}\!\}.$$

- *Otherwise, we define*

$$\mathsf{itperms}_G((i, \theta, n, j)) := \emptyset.$$

Definition 38 (Partial Order on Finite Bags).

Let X be a set and $<_X \subset X \times X$ a partial order on X. We define the partial order $\prec_X \subset \mathit{Bags}_\mathsf{fin}(X) \times \mathit{Bags}_\mathsf{fin}(X)$ on finite bags over X as the Dershowitz-Manna ordering [6] induced by $<_X$:

$$\begin{aligned} A \prec_X B \iff \exists C, D \in \mathit{Bags}_\mathsf{fin}(X).\ & \emptyset \neq C \subseteq B \\ & \wedge\ A = (B \setminus C) \uplus D \\ & \wedge\ \forall d \in D.\ \exists c \in C.\ d <_X c. \end{aligned}$$

We define $\preceq_X \subset \mathit{Bags}_\mathsf{fin}(X) \times \mathit{Bags}_\mathsf{fin}(X)$ such that

$$A \preceq_X B \iff A = B \ \vee\ A \prec_X B$$

holds.

Corollary 2. *The partial order* $\prec_\Lambda \subset \mathit{Bags}_{\mathsf{fin}}(\Lambda) \times \mathit{Bags}_{\mathsf{fin}}(\Lambda)$ *is well-founded.*

Proof. Follows from [6] and and well-foundedness of $<_\Delta$.

We view paths in a program order graph as single-branched subgraphs. This allows us to apply above definitions on graphs to paths. In particular, this allows us to refer to the capacity of a signal s on a path p by referring to sigCap_p.

For the following definition, remember that a bag $B \in \mathit{Bags}(X)$ is a function $B : X \to \mathbb{N}$ while a logical heap $H \in \mathit{Heaps}^{\mathsf{log}}$ is a function $H : \mathcal{R}^{\mathsf{log}} \to \mathbb{Q}_{\geq 0}$.

Definition 39. *We define the functions* $\mathsf{itPerms}_{\mathsf{lh}} : \mathit{Heaps}^{\mathsf{log}} \to \mathit{Bags}(\Lambda)$ *and* $\mathsf{waitPerms}_{\mathsf{lh}} : \mathit{Heaps}^{\mathsf{log}} \to \mathit{Bags}(\Omega)$ *mapping logical heaps to bags of iteration and wait permissions, respectively, as follows:*

$$\begin{array}{lcl} \mathsf{itPerms}_{\mathsf{lh}}(H)(\delta) & := & \lfloor H(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) \rfloor \\ \mathsf{waitPerms}_{\mathsf{lh}}(H)(id,\delta) & := & \lfloor H(\mathrm{wperm}_{\mathsf{lRes}}(id,\delta)) \rfloor \end{array}$$

Note that for consistent logical heaps $(h^{\mathsf{a}}, P^{\mathsf{a}})$ the above flooring is without any affect.

Lemma 10. *Let* $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}}$ *be an annotated reduction sequence such that every initial thread-local heap contains only finitely many permissions, i.e., such that for every* $\theta \in \mathsf{dom}(P_0^{\mathsf{a}})$ *the set*

$$\begin{array}{c} \{\delta \in \Lambda \mid P_0^{\mathsf{a}}(\theta).\mathsf{heap}(\mathrm{itperm}_{\mathsf{lRes}}(\delta)) > 0\} \\ \cup \\ \{(id,\delta) \in \Omega \mid P_0^{\mathsf{a}}(\theta).\mathsf{heap}(\mathrm{wperm}_{\mathsf{lRes}}(id,\delta)) > 0\} \end{array}$$

is finite. Then, $\mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}(\theta).\mathsf{heap})$ *and* $\mathsf{waitPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}(\theta).\mathsf{heap})$ *are finite for every choice of* $i \in \mathbb{N}$ *and* $\theta \in \mathsf{dom}(P_i^{\mathsf{a}})$.

Proof. Proof by induction on i.

Lemma 3. *Let* $G((h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i\in\mathbb{N}})$ *be a program order graph and let* $p = (V, E)$ *be a path in* G *with* $S_p^\infty = \emptyset$. *For every* $\theta \in \mathsf{dom}(P_0^{\mathsf{a}})$ *let* $P_0^{\mathsf{a}}(\theta).\mathsf{heap}$ *be finite and complete. Then,* p *is finite.*

Proof. Assume p is infinite. We prove a contradiction by assigning a finite capacity to every node along the path. Consider the function $\mathsf{nodeCap} : V \to \mathit{Bags}_{\mathsf{fin}}(\Lambda)$ defined as

$$\mathsf{nodeCap}(i) := \mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap}) \uplus \biguplus_{\substack{id \in \{id' \mid (id',\_) \in \mathsf{waitPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})\} \\ L \in \mathcal{L}evs}} \mathsf{sigCap}_p((id,L),i).$$

For every $i \in V$, the capacity of node i, i.e., $\mathsf{nodeCap}(i)$, is the union of two finite iteration permission bags: (i) To the left $\mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ captures all iteration permissions held by θ_i in step i. (ii) To the right $\biguplus \mathsf{sigCap}_p((id,L),i)$ captures all iteration permissions that will be created along the suffix of p that

starts at node i by waiting for signals for which thread θ_i already holds a wait permission (id, δ) in step i.

Note that for every $i \in V$, the bag of iteration permissions returned by $\mathsf{nodeCap}(i)$ is indeed finite. All initial thread-local heaps contain only finitely many permissions. By Lemma 10 $\mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ and $\mathsf{waitPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap})$ are finite. Since signal IDs are unique, for every fixed choice of i and id, there is at most one level L, for which $\mathsf{sigCap}_p((id, L), i) \neq \emptyset$. By assumption, along p all signals are waited for only finitely often, i.e., $S_p^\infty = \emptyset$. Hence, also the big union $\biguplus \mathsf{sigCap}_p((id, L), i)$ is defined and finite.

Consider the sequence $(\mathsf{nodeCap}(i))_{i \in V}$. Since every element is a finite bag of iteration permissions, we can order it by $\prec_\Lambda$. We are going to prove a contradiction by proving that the sequence is an infinitely descending chain.

Consider any edge $(i, \theta, n, j) \in E$. There are only three cases in which $\mathsf{nodeCap}(i) \neq \mathsf{nodeCap}(j)$ holds.

- $n =$ GTP-Red-WaitPerm:
  In this case, there are degrees δ, δ' with $\delta' <_\Delta \delta$, a signal s and $N \in \mathbb{N}$ for which we get

$$\mathsf{nodeCap}(j) = (\mathsf{nodeCap}(i) \setminus \{\!\{\delta\}\!\}) \uplus \{\!\{\underbrace{\delta', \ldots, \delta'}_{N \text{ times}}\}\!\}.$$

  That is, $\mathsf{nodeCap}(j) \prec_\Lambda \mathsf{nodeCap}(i)$.
- $n =$ GTP-Red-WeakItPerm: Same as above.
- $n =$ AST-Red-ConsumeItPerm:
  In this case, we know that $\mathsf{nodeCap}(j) = \mathsf{nodeCap}(i) \setminus \{\!\{\delta\}\!\} \prec_\Lambda \mathsf{nodeCap}(i)$ holds for some δ.

(Note that in case of $n =$ GTP-Red-Wait, we have $\mathsf{nodeCap}(i) = \mathsf{nodeCap}(j)$ since

$$\begin{array}{rcl} \mathsf{itPerms}_{\mathsf{lh}}(P_j^{\mathsf{a}}.\mathsf{heap}) & = & \mathsf{itPerms}_{\mathsf{lh}}(P_i^{\mathsf{a}}.\mathsf{heap}) \uplus \{\!\{\delta\}\!\}, \\ \biguplus \mathsf{sigCap}_p((id, L), j) & = & \left(\biguplus \mathsf{sigCap}_p((id, L), i)\right) \setminus \{\!\{\delta\}\!\} \end{array}$$

holds for some δ.) So, $\mathsf{nodeCap}$ is monotonically decreasing.

By assumption p is infinite. According to Corollary 1 this implies that the path contains infinitely many consume edges, i.e., edges with a labelling $n =$ AST-Red-ConsumeItPerm. Hence, the sequence $(\mathsf{nodeCap}(i))_{i \in V}$ forms an infinitely descending chain. However, according to Corollary 2, $\prec_\Lambda$ is well-founded. A contradiction.

Lemma 4. *Let $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i \in \mathbb{N}}$ be a fair annotated reduction sequence with $P_0^{\mathsf{a}} = \{(\theta_0, (H_0, c))\}$, $\mathsf{finite}_{\mathsf{ah}}(h_0^{\mathsf{a}})$, $\mathsf{complete}_{\mathsf{lh}}(H_0)$, $\mathsf{finite}_{\mathsf{lh}}(H_0)$ and $\mathsf{consistent}_{\mathsf{conf}}(h_0^{\mathsf{a}}, P_0^{\mathsf{a}})$. Let H_0 contain no signal or wait permission chunks. Further, let h_0^{a} contain no chunks $\mathrm{unlocked}_{\mathsf{aRes}}(m, P, H_P)$ where H_P contains any signal chunks. Let G be the program order graph of $(h_i^{\mathsf{a}}, P_i^{\mathsf{a}})_{i \in \mathbb{N}}$. Then, $S_G^\infty = \emptyset$.*

Proof. Suppose $S_G^\infty \neq \emptyset$. Since $\mathcal{L}evs$ is well-founded, the same holds for the set $\{\mathsf{lev}(s) \mid s \in S^\infty\}$. Hence, there is some $s_{\min} \in S^\infty$ for which no $z \in S^\infty$ with $\mathsf{lev}(z) <_\mathsf{L} \mathsf{lev}(s_{\min})$ exists.

Since neither the initial logical heap H_0 nor any unlocked lock invariant stored in h_0^a does contain any signals, $s_{\min}$ must be created during the reduction sequence. The reduction step creating signal $s_{\min}$ is an application of GTP-Red-NewSignal, which simultaneously creates an obligation to set $s_{\min}$. By preservation of completeness, Lemma 7, every thread-local logical heap $P_i^\mathsf{a}(\theta).\mathsf{heap}$ annotating some thread θ in some step i is complete. According to reduction rule GTP-Red-Wait, every wait edge $(a, \theta, (\text{GTP-Red-Wait}, s_{\min}), b)$ implies together with completeness that in step a (i) thread θ does not hold any obligation for $s_{\min}$ (i.e. $P_a^\mathsf{a}(\theta).\mathsf{heap}(\mathrm{obs}_{\mathsf{lRes}}(O)) = 1$ for some bag of obligations O with $s_{\min} \notin O$) and (ii) $s_{\min}$ has not been set, yet (i.e. $\mathrm{signal}_{\mathsf{aRes}}(s_{\min}, \mathsf{False}) \in h_a^\mathsf{a}$). Hence, in step a another thread $\theta_{\mathrm{ob}} \neq \theta$ must hold the obligation for $s_{\min}$ (i.e. $P_a^\mathsf{a}(\theta_{\mathrm{ob}}).\mathsf{heap}(\mathrm{obs}_{\mathsf{lRes}}(O)) = 1$ for some bag of obligations O with $s_{\min} \in O$). Since there are infinitely many wait edges concerning $s_{\min}$ in G, the signal is never set.

By fairness, for every wait edge as above, there must be a non-ghost reduction step $h_k^\mathsf{a}, P_k^\mathsf{a} \overset{\theta_{\mathrm{ob}}}{\rightsquigarrow}_{\mathsf{atp}} h_{k+1}^\mathsf{a}, P_{k+1}^\mathsf{a}$ of the thread θ_{ob} holding the obligation for $s_{\min}$ with $k \geq a$. Hence, there exists an infinite path p_{ob} in G where each edge $(e, \theta_{\mathrm{ob}}, n, f) \in \mathsf{edges}(p_{\mathrm{ob}})$ concerns some thread θ_{ob} holding the obligation for $s_{\min}$. (Note that this thread ID does not have to be constant along the path, since the obligation can be passed on during fork steps.)

The path p_{ob} does not contain wait edges $(e, \theta_{\mathrm{ob}}, (\text{GTP-Red-Wait}, s^\infty), f)$ for any $s^\infty \in S^\infty$, since reduction rule GTP-Red-Wait would (together with completeness of $P_e^\mathsf{a}(\theta_{\mathrm{ob}}).\mathsf{heap}$) require s^∞ to be of a lower level than all held obligations. This restriction implies $\mathsf{lev}(s^\infty) <_\mathsf{L} \mathsf{lev}(s_{\min})$ and would hence contradict the minimality of $s_{\min}$. That is, $S_{p_{\mathrm{ob}}}^\infty = \emptyset$.

By preservation of finiteness, Lemma 6, we get that every logical heap associated with the root of p_{ob} is finite. This allows us to apply Lemma 3, by which we get that p_{ob} is finite. A contradiction.