# TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs

Emanuele D'Osualdo
Imperial College London
e.dosualdo@ic.ac.uk

Azadeh Farzan
University of Toronto
azadeh@cs.toronto.edu

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Julian Sutherland
Imperial College London
julian.sutherland10@ic.ac.uk



## Abstract

We introduce TaDA Live, a separation logic for reasoning compositionally about the termination of fine-grained concurrent programs. We illustrate the subtlety of our reasoning using a spin lock and a CLH lock, and prove soundness.

## 1 Introduction

Scalable reasoning for fine-grained concurrent programs interacting with shared memory is a fundamental, open research problem. Developers manage the complexity of concurrent software systems by designing software components that are *compositional*: that is, components that are both *local* and *modular*. With locality, a developer designs local subcomponents with *interfaces* that connect to the rest of the system. With modularity, a developer designs reusable subcomponents with *abstract* software interfaces that can hide the complexity of the subcomponents from the rest of the system. The challenge is to develop compositional reasoning of concurrent programs, which follows the intuitions of the developer in how to structure their software components, with precisely defined specifications of software interfaces. Specifications should not leak implementation details and should be expressed at the level of abstraction of the client.

We are beginning to obtain a good understanding of compositional reasoning about *safety properties* of concurrent programs: that is, if the program terminates and the input satisfies the precondition, then the program does not fault and the result satisfies the postcondition. A breakthrough came in the work of [2, 23], which introduced concurrent separation logic to provide a first step towards compositional reasoning about course-grained concurrent programs. This led to a flowering of work on modern concurrent separation logics for reasoning about fine-grained concurrent programs, introducing logical abstraction (the fiction of separation) in the CAP logic [8] and abstract atomicity (the fiction of atomicity) in the TaDA logic [4]. These ideas are also expressible in Iris [14] using higher-order reasoning and FCSL [21] using assertions about histories. It is now possible to provide compositional reasoning about safety properties of concurrent programs, with specifications that match the intuitive software interface of the developer and formally verified implementations and clients.

We have comparatively little understanding of compositional reasoning about *progress (liveness) properties* for fine-grained concurrent algorithms: that is, something good eventually happens. Examples of progress properties include termination, deadlock-freedom, or that every user request is eventually served. The intricacies of the design of concurrent programs often arise precisely from the need to make the program correct with respect to progress properties. Such properties therefore form an essential part of the software interface: in `java.util.concurrent`, the lock module has a flag that a developer can set to determine the progress behaviour of the chosen lock, with different locks being suitable for different clients. Such properties should be precisely stated in the specifications of concurrent programs.

In this paper, we focus on compositional reasoning about the termination of fine-grained concurrent programs. There has been some work on reasoning about synchronisation patterns of coarse-grained language primitives such as primitive locks and communication channels [1, 16]. This work side-steps a fundamental challenge for fine-grained concurrent programs, addressed in this paper, of how to handle the never-disabled implementation of, for example, a spin lock, with the abstract blocking behaviour of the lock module. Our aims better resonate with the work on history-based reasoning about fine-grained concurrent programs [10, 15], where specifications are described using abstract histories and interference is given by a rely/guarantee relation on such histories. This work provides a general, expressive framework in which to explore many forms of concurrent behaviour. However, with termination, the specifications are complex and the verification requires explicit manipulation of the histories. We have a different approach, to specify and verify programs by using specifications that that are based on abstract resource such as locks, rather than general histories. The work that is closest to ours is that of Lili [19], which is the only concurrent separation logic to prove directly termination results for fine-grained concurrency with blocking. However, the specifications for blocking behaviour are not atomic and, as a result, the verification does not scale well,

even for simple clients. We discuss these claims further in Section 6 and Appendix.

We introduce TaDA Live, a compositional separation logic for reasoning about the termination of fine-grained concurrent programs. Our first contribution is the introduction of TaDA Live total specifications, to express the liveness constraints on the environment under which an operation can guarantee termination. These specifications build on the partial safety specifications of TaDA [4]. The specifications are *expressive*, in that they capture all the necessary properties such as blocking and impedance. They are *concise*, in that they do not require information about histories to express the liveness constraints. They are *abstract*, in that they do not leak implementation details and they are expressed at the level of abstraction of the client. They are truly *atomic*, in that a linearizable operation will be represented as an atomic operation even when blocking. In summary, the TaDA Live total specifications are radically different from previous work; they are more readable, express atomicity, and support truly compositional reasoning.

Our second contribution is a proof system for the verification of implementations against their specifications. The proof system achieves compositionality by introducing a number of innovations. It uses a form of *liveness ghost state*, given by an obligation algebra, to express thread-local liveness constraints on arbitrary abstract resources. It uses a *layer system* to express dependencies on obligations that can be linked to custom (as opposed to primitive) synchronisation patterns. These dependencies can be hidden from the client to provide truly modular verification. Finally, it uses a *environment liveness condition* which characterises liveness properties of the abstract environment of a thread in a thread-local way. We prove our proof system is sound.

TaDA Live can verify all the terminating examples of [11, 13, 15, 18, 19, 24], including spin locks, ticket locks, MCS locks, spin and blocking counters, blocking queues/stacks and lock-coupling sets. Details can be found in the Appendix. We introduce the ideas of TaDA Live using two well-known, fine-grained implementations of a lock module: the spin lock and the Craig-Landin-Hagersten lock.

## 2 Motivation

We motivate TaDA Live specification and verification, using two implementations of a lock module which have the same safety specification, but different termination specifications.

***Two Lock Implementations.*** Consider the *spin lock* and the *CLH lock* given in Fig. 1. The implementations enable threads to compete for the acquisition of a lock at address `x` by running concurrent invocations of the `lock(x)` operation. Only one thread will succeed, leaving the others to wait until the `unlock(x)` operation is called by the winning thread.

We use a simple, fine-grained concurrent while language for manipulating shared state. The shared state comprises

Spin Lock

```
1 lock(x){
2   b := 0;
3   while(b = 0){
4     b := CAS(x,0,1);
5   }
6 }
7
8
9 unlock(x) {[x] := 0;}
```

CLH Lock

```
1 lock(x) {
2   c := alloc(1); [c] := 1;
3   p := FAS(x + 1, c);
4   v := [p];
5   while(v ≠ 0) { v := [p]; }
6   [x] := c;
7 }
8
9 unlock(x) {⟨h := [x]; [h] := 0;⟩}
```

**Figure 1.** Two fine-grained implementations of a lock.

heap cells which have addresses and store values (addresses, integers, booleans). The `[x]` notation denotes the value stored at the heap cell with address `x`. The primitive commands, such as assignment, lookup and mutation, are assumed to be primitive atomic and non-blocking: every primitive command, if given a CPU cycle, will terminate. Since reads and writes may race, the language is equipped with a *compare-and-swap* primitive command, **CAS**(x, $v_1$, $v_2$), which checks if the value stored at `x` is $v_1$: if so, it atomically stores $v_2$ at `x` and returns 1; otherwise it just returns 0. Similarly, the *fetch-and-set* primitive command, **FAS**(x, $v$), stores $v$ at `x` returning the value that was stored at `x` just before overwriting it.

The spin lock in Fig. 1 is standard. Its state comprises a heap cell at `x` which stores either 0 (unlocked) or 1 (locked). The Craig-Landin-Hagersten (CLH) lock in Fig. 1 serves threads competing for the lock in a FIFO order. It keeps a queue of requests, but only the head and tail pointers are stored in shared memory (in `x` and `x+1` respectively). The predecessor pointers are stored in each thread's local state (in `p`). The lock is acquired by a thread when the predecessor signals release of the lock by setting its node to 0. Unlocking a node corresponds to setting the head node value to 0.

***TaDA Safety Specification.*** We introduce the partial safety specifications of TaDA using our lock example. The spin lock and the CLH lock behave in the same way with respect to *safety*, and hence satisfy the same TaDA safety specification.

TaDA specifications combine the data abstraction introduced in the CAP logic [8], with time abstraction captured by *abstract atomicity*. Consider the concurrent trace in Figure 2: the local thread invokes the operation `lock(x)`; during this invocation, the environment continues to lock and unlock the lock. *Linearizability* [12], probably the best known technique for describing abstract atomicity, is a correctness condition for concurrent data structures that, when satisfied, enables us to find a sequential trace (no invocation overlaps) which is equivalent to the original concurrent trace. Typically, this is done by identifying so-called 'linearization points', illustrated by the bullet points in the concurrent trace, which are steps in the invocation of a operation where the abstract state change becomes visible to the client: with the spin lock, the linearization point is after the successful CAS; with the CLH lock, the linearization point is line 6. The linearization

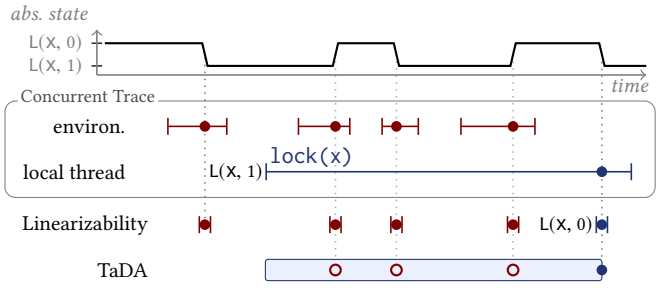



**Figure 2.** Linearizability versus TaDA.

points induce a sequential trace: the order of the operations coincides with the order of execution of the corresponding linearization points, as shown in the sequential trace in Fig. 2.

Since every concurrent trace can be seen equivalently as a sequential trace, linearizability enables the client of a data structure to use sequential specifications for operations, which are considered *atomic* as they are sequential *even in a concurrent environment*. Such sequential specifications cannot describe fully the behaviour of blocking operations like `lock`: there is a disconnect between the precondition at the time of invocation, and the required state at the time of the linearization point. A `lock` operation can be invoked when the lock is either locked or unlocked. The state can be changed by the environment arbitrarily many times before the linearization point. At the linearization point, the atomic step of acquiring the lock must start from the unlocked state. A sequential specification just describes the state of the lock before and after the linearization point.

To solve this problem, TaDA safety specifications explicitly describe what we call the *interference phase* of the execution of an operation. Consider the TaDA representation of the execution of the lock operation given in Fig. 2. The call is not collapsed to a single instant in time, but instead is represented by an interval of time from its invocation to the linearization point. During this interference phase, there can be abstract state changes by the environment, represented in the figure by ∘, but not by the local thread. The TaDA partial specification for the `lock` operation is the *atomic triple*:

$$\vdash \forall l \in \{0,1\}.\langle \mathsf{L}(\mathtt{x}, l)\rangle\ \texttt{lock(x)}\ \langle \mathsf{L}(\mathtt{x},1) \wedge l = 0\rangle \qquad (1)$$

where the lock assertion $\mathsf{L}(\mathtt{x}, l)$ describes a lock at address x with the abstract state $l \in \{0,1\}$. The *pseudo-quantification* $\forall l \in \{0,1\}$ together with the precondition $\mathsf{L}(x, l)$ form the *interference precondition*, stating that the environment must preserve the existence of the lock at x but may change the value of $l$; the implementation must be able to tolerate this interference by the environment. The triple (1) states that, if the environment satisfies the interference precondition and the operation terminates, then the operation does an abstract atomic step taking the lock from the unlocked to locked state. In the postcondition, $l$ is bound to the abstract state of the lock *just before the linearization point*. The $l = 0$ in the postcondition says that the lock *must have been available* at the time the operation atomically locked it. This is the place where the specification records mutual exclusion.

The `unlock` specification is $\vdash \langle \mathsf{L}(\mathtt{x},1)\rangle\ \texttt{unlock(x)}\ \langle \mathsf{L}(\mathtt{x},0)\rangle$, stating that the implementation can assume the environment is not able to change the abstract state during interference.

***TaDA Live Total Specification.*** We introduce the total specifications of TaDA Live, again using our lock example. This time, the spin lock and the CLH lock do not have the same TaDA Live specifications, because the two lock implementations terminate under different conditions.

**Example 2.1** (Distinguishing client)**.** The following program has different termination behaviour with the two locks:

```
lock(x);                 ||  while(!d){ // d false initially
[done] := true;          ||    lock(x); d := [done]; unlock(x);
unlock(x);               ||  }
```

With spin lock, this client program does not always terminate. The lock invocation of the left thread may be scheduled infinitely often, but always in a state where the lock is locked. As a result, `done` will never be set to `true` making the while loop spin forever. The spin lock has been *starved* by the other thread. With CLH lock, this program is guaranteed to terminate: a fair scheduler[1] will eventually allow the left thread to enqueue its node; from then on, the thread on the right can only acquire the lock at most once; after unlocking, the next `lock(x)` call of the right thread would enqueue it after the left thread, which is now the only unblocked thread. The CLH lock is *starvation free*.

Including the property of starvation-freedom in an atomic specification is not easy: it speaks of infinite traces and enabledness. We adopt a different approach, by capturing the difference in the behaviour of the two locks using different *liveness conditions* on how the environment may change the abstract state. Starvation freedom will be a logical consequence of the weaker liveness assumptions needed by CLH.

A blocking operation, such as `lock`, cannot make progress independently of the abstract state: for example, the abstract state 0, describing that the lock is unlocked, is *good* because the `lock(x)` operation can make progress; the abstract state 1, describing that the lock is locked, is *bad* because the `lock(x)` cannot make progress until the lock has been released. Consider the graph in Fig. 3(a), which illustrates the appropriate environment assumption necessary for the termination of a CLH lock. It describes the change of state of the lock over time, requiring that there are infinitely many good (unlocked) states. Assuming that every lock acquired is eventually released by the environment, it will eventually be the

[1]We assume that the scheduler is *weakly fair*: every active thread will be scheduled infinitely often. Note that non-blocking primitive commands will behave exactly the same under weak fairness and *strong fairness*: every infinitely-often enabled thread will be eventually scheduled. The primitive commands are, by definition, always enabled.

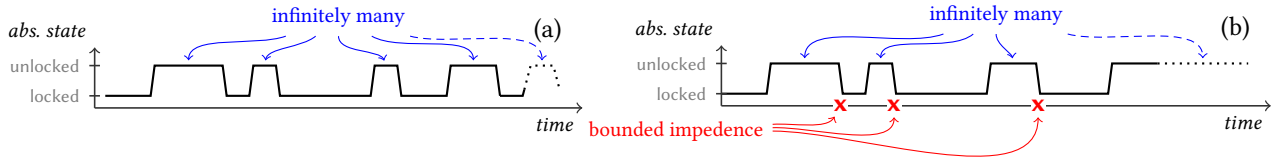



**Figure 3.** Fair blocking (a) and impeded blocking (b).

operation's turn to lock as it gets to the head of the queue. We refer to this kind of assumption on the environment's behaviour as *fair blocking*: every time the environment is in a bad state it will eventually transition to a good state.

We capture this fair-blocking assumption in TaDA Live specifications, by extending the TaDA atomic triples with an additional liveness condition on the environment. The TaDA Live total specification of the CLH lock is:

$$\vdash \mathbb{V}l \in \{0,1\} \twoheadrightarrow \{0\}.\langle \mathsf{L}(\mathtt{x}, l)\rangle\ \mathtt{lock(x)}\ \langle \mathsf{L}(\mathtt{x}, 1) \wedge l = 0\rangle \quad (2)$$

The pseudo-quantification $\mathbb{V}l \in \{0,1\} \twoheadrightarrow \{0\}$ now comprises a liveness condition: as well as stating that the environment can keep changing the lock, it also states that if the lock is in a bad state ($l = 1$) then the environment must always eventually change it to a good state ($l = 0$). The implementation needs to ensure termination under the assumption that the lock always eventually returns to the unlocked state. Note that the environment is allowed to change $l$ back to 1 arbitrarily many times, provided it always eventually sets it back to 0. With this assumption about the environment, the CLH `lock` operation will terminate with the desired behaviour. We prove this in Section 4.1.

The assumptions stated by this TaDA Live specification are not enough for spin lock to terminate. Intuitively, we need as an extra assumption that the number of times that the environment transitions from a good state (0) to a bad state (1) is bounded. While for the CLH lock even transitions from good to bad represent progress since the operation gets closer to the head of the queue, for spin lock these transitions represent delays. Consider the graph in Figure 3(b), which illustrates the appropriate environment assumption for the termination of the spin lock. As well as requiring infinitely many good states, it also requires the existence of boundedly many ×s in the graph, identifying the impeding transition steps from good to bad. We refer to this requirement that the transition steps are bounded as *bounded impedance*. Note that impedance also applies to non-blocking operations.

The bounded impedence condition, illustrated in Fig. 3(b), is represented in TaDA Live by adding an additional component to the lock assertion. The lock is now represented by the assertion $\mathsf{L}(x, l, \alpha)$, where the ordinal $\alpha$ provides the well-founded 'budget' for impedance. The TaDA Live total specification of spin lock is given by the atomic triple:

$$\forall \phi.\ \vdash \mathbb{V}l \in \{0,1\} \twoheadrightarrow \{0\}, \alpha.\ \langle \mathsf{L}(\mathtt{x}, l, \alpha) \wedge \phi(\alpha) < \alpha\rangle \mathtt{lock(x)} \langle \mathsf{L}(\mathtt{x}, 1, \phi(\alpha)) \wedge l = 0\rangle \quad (3)$$

The triple requires the lock to be always eventually unlocked as for CLH lock. In addition, it states that every `lock` operation consumes the budget $\alpha$ by a non-trivial amount. The initial value of the budget and the function $\phi$ from ordinals to ordinals is determined by the client, which must demonstrate that the budget is enough to make all its calls.

For the CLH lock, the TaDA Live specification of the unlock operation is the same as the TaDA specification. For spin lock, the TaDA Live specification needs to incorporate the ordinals: $\vdash \langle \mathsf{L}(\mathtt{x}, 1, \alpha)\rangle\ \mathtt{unlock(x)}\ \langle \mathsf{L}(\mathtt{x}, 0, \alpha)\rangle$. Note that the impedance budget $\alpha$ is not affected by the operation, since unlock does not cause impedance to other concurrent lock operations.

The representation of impedance by ordinals in the abstract state was first introduced to concurrent separation logics with Total TaDA [5]. Total TaDA only handles non-blocking operations. Here, we smoothly integrate ordinals into TaDA Live that fully supports blocking. It is crucial to note that fair blocking and bounded impedance cannot be enforced in any way by the lock module itself. They are the result of the specific pattern of usage of the operation by a client. These conditions must be represented in the specification as assumptions that the client needs to fulfil in order to use an operation.

***TaDA Live Verification.*** In Section 4.1, we verify the CLH lock and, in the Appendix, we verify the spin lock. Here, we sketch how to verify Example 2.1 (with a fair lock). Let $\mathbb{C}_\ell$ and $\mathbb{C}_r$ denote the left and right threads respectively. Our aim is to show that the program terminates, by proving $\vdash \{\mathsf{L}(\mathtt{x}, 0) * \mathtt{done} \mapsto \mathtt{false}\}\ \mathbb{C}_\ell \parallel \mathbb{C}_r\ \{\exists l.\, \mathsf{L}(\mathtt{x}, l) * \mathtt{done} \mapsto \mathtt{true}\}$. In TaDA Live, all our triples are total, including Hoare triples, which are special cases of our general triple presented in Section 3. The two threads share the cells `x` and `done`, so we wrap them in a region $\mathbf{c}_r(\mathtt{x}, \mathtt{done}, l, d)$ where $l$ and $d$ are the values at `x` and `done` respectively. Regions can be shared by different threads: they can only be updated by atomic operations; and they provide a protocol that threads must respect for them to interact with this portion of shared memory.

Using the Par rule (Figure 5), we can verify Hoare triples for $\mathbb{C}_\ell$ and $\mathbb{C}_r$ separately, then join them together using the separating conjunction $*$. For $\mathbb{C}_\ell$, the crucial step in proving termination is to show that the call to `lock(x)` will eventually return. This is true because the lock is always eventually unlocked, a *liveness invariant* that we need to formalise to prove that the environment during the lock invocation will satisfy the liveness precondition of the fair lock specification.

To do this, TaDA Live introduces special liveness ghost state given by an *obligation algebra*: a partial commutative monoid (PCM) with some additional structure. Intuitively, we declare an obligation $\kappa$ associated with region $r$ that will be held locally by the owner of the lock, and that will be fulfilled by unlocking the lock and putting it back in region $r$ awaiting the lock to be locked again. The *interpretation* of the region is: $I(\mathbf{c}_r(\mathtt{x}, \mathsf{done}, l, d)) \triangleq \mathsf{L}(\mathtt{x}, l) * \mathsf{done} \mapsto d * (\lceil \mathrm{U} \rceil_r \wedge l = 0) \vee (\lceil \mathrm{L} \rceil_r \wedge l = 1) * \cdots$. The original precondition has been wrapped up in the region $\mathbf{c}_r$ with additional obligation assertions: the obligation $\mathrm{U} = \mathrm{L} \bullet \kappa$ records that, when $l = 0$, the $\kappa$ is inside the region, absorbed by U; the obligation L records that, when $l = 1$, the $\kappa$ is outside the region, owned locally by the owner of the lock. Unlike safety ghost state, which in TaDA is specified using a PCM called a *guard algebra*, obligations have an implicit requirement that they are *live*, that is, they will be inside the region infinitely often. This requirement has two uses: (1) As a *liveness assumption*: when a local thread does not hold $\kappa$ and $l = 1$, we know that there must be a thread holding $\kappa$ somewhere in the environment; we can rely on the requirement being fulfilled by the environment at some point, with the effect of unlocking x. (2) As a *liveness obligation*: when a local thread obtains $\kappa$ locally by locking, we have to show that eventually $\kappa$ is placed back in the region, by unlocking. When using the specification of `lock` in the proof of $\mathbb{C}_\ell$, we are required to show that the liveness precondition $\forall l \in \{0, 1\} \twoheadrightarrow \{0\}$ is satisfied. This is done by proving the *environment liveness condition* with target states $T = (l = 0)$, which asks that from states violating $T$, the environment actions will make progress towards establishing $T$. Since $\mathbb{C}_\ell$ does not hold $\kappa$ when calling `lock(x)`, the liveness assumption of $\kappa$ is sufficient to show this, so the call will terminate.

For $\mathbb{C}_r$, an analogous argument justifies why the lock operations in the body of the loop will terminate. In addition, we need to argue why the loop will eventually be exited, which depends on the environment setting `done` to true. To apply the While rule of TaDA Live, we want to prove the environment liveness condition with target $T = (\mathtt{done} = \mathtt{true})$. From $T$, in at most one iteration, the loop terminates. From outside $T$, we need to show that there will be environment steps that make progress towards establishing $T$.

To do so, we again appeal to obligations: we introduce the obligation S representing the obligation to set `done` to true; we link it to the state of `done` using the obligation $\mathrm{D} = \mathrm{W} \bullet \mathrm{S}$, similarly to before; and we complete the interpretation of the region by $I(\mathbf{c}_r(\mathtt{x}, \mathsf{done}, l, d)) \triangleq \cdots * (\lceil \mathrm{D} \rceil_r \wedge d) \vee (\lceil \mathrm{W} \rceil_r \wedge \neg d)$. Since `done` is set to false initially, we obtain S locally. It is $\mathbb{C}_\ell$'s duty to set `done` to `true`, so the S obligation is given to $\mathbb{C}_\ell$ using the parallel rule. Thus, $\mathbb{C}_r$ will not be holding S, and this information provides a liveness assumption that eventually S will be put back into the region (absorbed by D) which can only happen if $d$ is true. A region is also associated with an *interference transition system* which specifies the valid transformations of the state of the region. In this case we set $\mathrm{S} : (l, d) \rightsquigarrow (l, \mathtt{true})$ and $\mathbf{0} : (l, d) \rightsquigarrow (l', d)$ which states that the holder of S can set $d$ to true and that anybody ($\mathbf{0}$ is the identity of the PCM) can change the state of the lock, while preserving the value of $d$. Both threads $\mathbb{C}_\ell$ and $\mathbb{C}_r$ respect these conditions. With this information we can infer that once $d$ becomes true, it will stay so forever. This is the last condition needed to prove the loop of $\mathbb{C}_r$ terminates: when we are in an unblocked state in $T$ we will make local progress; when we are in a blocked state the environment will make progress towards $T$; and we can only move from $T$ to not $T$ boundedly many times (in this case: not even once).

In our argument so far, there is the possibility of unsound circular reasoning. To observe this, replace $\mathbb{C}_r$ with $\mathbb{C}'_r \triangleq$ `lock(x);while(!d){d:=[done]};unlock(x)`. Although our argument still applies, the program can now deadlock. The problem is that, in proving $\mathbb{C}'_r$, we use the liveness assumption about S while holding $\kappa$; and in proving $\mathbb{C}_\ell$ we use the liveness assumption about $\kappa$ while holding S. We break this symmetry by introducing the notion of *layers*: the layer of an obligation $\mathrm{lay}(\mathrm{O})$ is an element of a well-founded set. The revised liveness requirement associated with an obligation is therefore: *an obligation is live if it has layer strictly lower than the layer of any obligation we may be holding locally.* A relation $\mathrm{lay}(\mathrm{O}_1) < \mathrm{lay}(\mathrm{O}_2)$ states that anybody can assume $\mathrm{O}_1$ live while trying to fulfill $\mathrm{O}_2$, but not viceversa. With layers, we could not prove $\mathbb{C}_\ell \parallel \mathbb{C}'_r$ because to be able to assume $\kappa$ live in $\mathbb{C}_\ell$ we would need $\mathrm{lay}(\kappa) < \mathrm{lay}(\mathrm{S})$ since S is held locally. The argument for $\mathbb{C}'_r$, however requires $\mathrm{lay}(\mathrm{S}) < \mathrm{lay}(\kappa)$, contradiction.

Note that we would not be able to show the termination of Example 2.1 using a spin lock. This is because we would not be able to use its `lock` specification, since we would not be able to put a bound on the number of calls we would make.

## 3 Technical Development

***Programs*** The syntax of programs is defined in Fig. 4. A program is a module $\mathbb{M}$ which imports a number of other modules and defines some exported functions. In our terminology, in a program (**`import`** $\mathbb{M}$ **`in`** $\mathbb{D}$), the $\mathbb{D}$ is a client of module $\mathbb{M}$. A closed program is simply a module exporting a single `main` function. Boolean expressions $\mathbb{B}$ and integer expressions $\mathbb{E}$ contain the usual logical, comparison and arithmetic operators. We have the usual syntax for commands $\mathbb{C}$. The language uses fine-grained concurrency: all the primitive commands are atomic and non-blocking. For simplicity, we do not model deallocation. It can be handled using standard techniques.

***Ghost state*** Following modern concurrent separation logics, TaDA Live uses a partial commutative monoid (PCM) to specify ghost state. A PCM is a tuple $(S, \bullet, \mathbf{0})$ where $S$ is a set and $\bullet : S \times S \rightharpoonup S$ is a partial binary operation over $S$.

$$\mathbb{C} ::= \texttt{skip} \mid \mathbb{C}_1;\mathbb{C}_2 \mid \mathbb{C}_1 \parallel \mathbb{C}_2 \mid \texttt{while}(\mathbb{B})\{\mathbb{C}\} \mid \texttt{if}(\mathbb{B})\{\mathbb{C}_1\}\texttt{else}\{\mathbb{C}_2\} \mid \mathtt{x} := \texttt{alloc}(\mathbb{E}) \mid \mathtt{x} := \mathbb{E} \mid \mathtt{x} := [\mathbb{E}] \mid [\mathbb{E}_1] := \mathbb{E}_2 \mid \texttt{return}\,\mathbb{E} \mid \mathtt{r} := \texttt{CAS}(\mathbb{E}_1,\mathbb{E}_2,\mathbb{E}_3) \mid \mathtt{r} := \texttt{FAS}(\mathbb{E}_1,\mathbb{E}_2) \mid \mathtt{r} := \mathtt{f}(\mathbb{E}_1,\ldots,\mathbb{E}_n)$$

$$\mathbb{M} ::= \texttt{import}\,\mathbb{M}_1,\ldots,\mathbb{M}_n\,\texttt{in}\,\mathbb{D} \qquad \mathbb{D} ::= \epsilon \mid \mathtt{f}(\vec{\mathtt{x}})\{\mathbb{C}\},\mathbb{D} \qquad \mathbb{E} ::= \mathtt{x} \mid n \mid \mathbb{E}_1{+}\mathbb{E}_2 \mid \ldots \qquad \mathbb{B} ::= \mathtt{true} \mid \mathtt{false} \mid \mathbb{E}_1 \le \mathbb{E}_2 \mid \ldots$$

$$P,Q,R ::= \mathsf{emp} \mid r \Mapsto d \mid \mathsf{A}(\vec{z}) \mid \mathbf{t}_r^{\lambda}(\vec{z},x) \mid \lfloor \mathsf{G}(\vec{z}) \rfloor_r \mid \lceil \mathsf{O}(\vec{z}) \rceil_r \mid x \mapsto y \mid P * Q \mid P \wedge Q \mid \ldots \qquad d ::= \Diamond \mid \blacklozenge \mid (\mathbb{E}_1,\mathbb{E}_2)$$

$$\lambda;\mathcal{A} \vdash P \trianglerighteq m \overset{\text{def}}{\iff} \forall r \in \mathsf{RId}, \mathsf{O} \in \mathsf{Oblig}_r \backslash \{\mathbf{0}\}.\, \lambda;\mathcal{A} \vdash (P \wedge (\lceil \mathsf{O} \rceil_r * \mathsf{True})) \Rightarrow \mathsf{lay}(\mathsf{O}) \ge m$$

$$\lambda;\mathcal{A} \vdash P \trianglerighteq_R \infty \overset{\text{def}}{\iff} \forall r \in R, \mathsf{O} \in \mathsf{Oblig}_r \backslash \{\mathbf{0}\}.\, \lambda;\mathcal{A} \vdash (P \wedge (\lceil \mathsf{O} \rceil_r * \mathsf{True})) \Rightarrow \mathsf{False}$$

**Figure 4.** Definitions of commands, modules, assertions and layer constraints.

Where the operation is defined, it obeys the usual commutative monoid axioms with identity $\mathbf{0}$ (in general $\mathbf{0}$ can be a set of identity elements). We associate with a PCM $(S, \bullet, \mathbf{0})$ a *resource order* $\sqsubseteq_S$ over $S$: for all $s_1, s_2 \in S$, $s_1 \sqsubseteq_S s_2$ iff $\exists s \in S.\, s_1 \bullet s = s_2$; and $\sqsubset_S$ is the related strict order. We sometimes require the existence of a set of $\sqsubseteq_S$-maximal elements in $S$, denoted $\mathbf{1}_S$. We drop the subscripts when the PCM is clear from the context. Two elements $s, s' \in S$ are *compatible*, written $x \# y$), if $s \bullet s'$ is defined. An element $s \in S \setminus \{\mathbf{0}\}$ is an *atom* if $\nexists s' \in S.\, \mathbf{0} \sqsubset s' \sqsubset s$.

In TaDA Live, we use two kinds of ghost state: a *guards* PCM for safety; and an *obligations* PCM for liveness.

***Layers*** Obligations have additional structure, which we call *layers*. A *layer structure* $(\mathcal{L}, \le, \bot)$ is a well-founded partial order with minimal element $\bot$. We will often conflate the support set $\mathcal{L}$ with the full lattice structure. Given a layer structure $\mathcal{L}$, an *obligation algebra over* $\mathcal{L}$ is a tuple $(\mathsf{Oblig}, \bullet, \mathbf{0}, \mathbf{1}, \mathsf{lay})$ where $(\mathsf{Oblig}, \bullet, \mathbf{0})$ is a PCM with maximal element $\mathbf{1}$ and $\mathsf{lay}\colon \mathsf{Oblig} \to \mathcal{L}$ maps every obligation to a layer, with the property that $\forall x, y \in \mathsf{Oblig}.\ \mathsf{lay}(x \bullet y) \le \mathsf{lay}(x)$. For two layers $m_1, m_2$, we write $m_1 \sqcap m_2$, and $m_1 \sqcup m_2$ for their greatest lower bound and lowest upper bound, respectively. In an obligation algebra, we will always assume that $x \bullet x$ is undefined for every obligation $x$.

***Assertions*** Figure 4 defines the assertions, where $r \in \mathsf{RId}$ ranges over region identifiers and $\vec{z}$ is a vector of expressions. We include in the language all the usual classical separation logics assertions. The atomicity tracking assertions $r \Mapsto d$ come from TaDA, and are used to identify and certify the linearisation points. As in TaDA, we have CAP-style abstract predicates $\mathsf{A}(\vec{z})$, region assertions $\mathbf{t}_r^{\lambda}(\vec{z}, x)$, and guard assertions $\lfloor \mathsf{G}(\vec{z}) \rfloor_r$. The notable addition is *obligation assertions* $\lceil \mathsf{O}(\vec{z}) \rceil_r$.

In Fig. 4 we define two conditions that constrain the obligations that can be held when an assertion $P$ holds. Intuitively, $\lambda;\mathcal{A} \vdash P \trianglerighteq m$ says that $m$ is a lower bound to every obligation that may be held when $P$ holds. When $\lambda;\mathcal{A} \vdash P \trianglerighteq_R \infty$ holds, we say $P$ is *R-obligation-free*. When $R = \mathsf{RId}$, we write $\lambda;\mathcal{A} \vdash P \trianglerighteq \infty$ and $P$ is said *obligation-free*.

***Abstract predicates and regions*** Each module's proof will have to define the abstract predicates it exposes to the client and the region definitions it uses internally. We associate to each abstract predicate name A an *abstract predicate definition* of the form $\mathsf{A}(\vec{x}) \triangleq P$ with $\lambda;\mathcal{A} \vdash P \trianglerighteq \infty$: internal obligations do not leak the abstraction boundaries.

We associate to each region type $\mathbf{t}$ a *region type definition* which is a tuple $(\lambda, \mathsf{Guard}_\mathbf{t}, \mathsf{Oblig}_\mathbf{t}, \mathsf{AState}_\mathbf{t}, I_\mathbf{t}, \mathcal{T}_\mathbf{t})$ where $\lambda \in \mathbb{N}$ is a region level, $\mathsf{Guard}_\mathbf{t}$ and $\mathsf{Oblig}_\mathbf{t}$ are a guard and an obligation algebra respectively; $I_\mathbf{t}$ is a region interpretation of the form $I_\mathbf{t}(\mathbf{t}_r^{\lambda}(\vec{z}, x)) = P$ where $r$ is a region-identifier-valued variable, $\vec{z}$ is a vector of logical variables and $x$ is an $\mathsf{AState}_\mathbf{t}$-valued variable, all free variables of the assertion $P$; we require $\lambda;\mathcal{A} \vdash P \trianglerighteq_{\mathsf{RId}\setminus\{r\}} \infty$, i.e. the region interpretation can only link that region's obligations and its state. The *guarded transition system* $\mathcal{T}_\mathbf{t}$: $\mathsf{Guard}_\mathbf{t} \to \mathcal{P}(\mathsf{AState}_\mathbf{t} \times \mathsf{AState}_\mathbf{t})$ is a monotonic (in the resource ordering on guards and $\subseteq$) function associating guards to nondeterministic state transformers. We write $\mathcal{T}_\mathbf{t}(g)^*$ for the reflexive transitive closure of $\mathcal{T}_\mathbf{t}(g)$ (seen as a relation).

Region levels are mainly a device to prevent circular reasoning arising from recursive region type definitions. In all our examples the region types will have level 1.

**Remark 1.** While the original TaDA logic is based on *intuitionistic* separation logics, TaDA Live is based on *classical* separation logic. This is simply motivated by the need of disallowing the elimination law $P * Q \Rightarrow P$, which is unsound with respect to the meaning of obligations: when $Q$ contains obligation assertions, the implication would allow to forget the requirement of fulfilling those obligations. Region assertions, however, retain some intuitionistic flavour, as they obey, for all $\vec{x}, \vec{y}$, the law: $(\mathbf{t}_r^{\lambda}(\vec{x}) * \mathbf{t}_r^{\lambda}(\vec{y})) \iff (\mathbf{t}_r^{\lambda}(\vec{x}) \wedge \vec{x} = \vec{y})$.

### 3.1 The proof system

The judgement for commands has the general form

$$m;\lambda;\mathcal{A} \vdash \forall x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \,|\, P_\mathsf{a}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_\mathsf{h}(x,y) \,|\, Q_\mathsf{a}(x,y)\rangle$$

where $m$ and $k$ are layers, $\lambda \in \mathbb{N}$ is a region level, $\mathcal{A}$ is an atomicity context (defined below), $P_\mathsf{h}$ and $Q_\mathsf{h}(x,y)$ are the "Hoare" pre/post-conditions, and $P_\mathsf{a}(x)$ and $Q_\mathsf{a}(x,y)$ are the "atomic" pre/post-conditions. We call $\forall x \in X \twoheadrightarrow_k X'$ the *pseudo-quantifier* which expresses both safety assumptions

($x \in X$) and liveness assumptions ($X \twoheadrightarrow_k X'$). From this general judgement, we derive a number of abbreviated forms. The quantifier $\exists y$ is placed so the scope of the quantifier spans both post-conditions. When $y$ is a function of $x$, there is no need for the quantification and we omit it. When $X = X'$ the pseudo-quantifier assumption becomes trivial (i.e. vacuously satisfied by any environment) and $k$ irrelevant so we just write $\mathbb{V}x \in X$. When there is no occurrence of $x$ in the atomic pre/post-conditions, $X$ can be arbitrarily set to be a singleton set $\{*\}$ and we omit the pseudo-quantifier altogether. Finally, when the atomic pre/post-conditions are both empty, the judgement has the meaning of a total Hoare triple in standard concurrent separation logics, and therefore we use the notation $m; \lambda; \mathcal{A} \vdash \{P\}\ \mathbb{C}\ \{Q\}$ which, given the short-hands we introduced, is equivalent to

$$m; \lambda; \mathcal{A} \vdash \mathbb{V}x \in \{*\} \twoheadrightarrow_\bot \{*\}.\langle P \,|\, \mathsf{emp}\rangle\ \mathbb{C}\ \exists y.\langle Q \,|\, \mathsf{emp}\rangle.$$

***Atomicity context*** The atomicity context $\mathcal{A}$ is a list of tuples of the form $(r, x, X \twoheadrightarrow_k X', Y(x))$ which record (i) which assumptions can be made about interference on region $r$ before the linearization point happens ($x \in X \twoheadrightarrow_k X'$), and (ii) which linearization point we should be proving happens (from $x \in X$ to $y \in Y(x)$).

***Rely/Guarantee*** The *rely* relation $\mathbf{R}_{\lambda;\mathcal{A}}$ is given by the region type definitions, and the level and context $\lambda, \mathcal{A}$. It formalises the environment interference as a relation between states. It allows the environment to perform any modification allowed by the interference transition system of a region and the atomicity context, provided the transition is guarded by some guard that is compatible with the guards held locally. An assertion $P$ is *stable* ($\lambda; \mathcal{A} \vDash P$ stable) if it is closed under rely: for all $s$ satisfying $P$, for all $s'$ with $s\ \mathbf{R}_{\lambda;\mathcal{A}}\ s'$, then $s'$ satisfies $P$. We write $\mathbf{R}^*_{\lambda;\mathcal{A}}$ for the transitive closure of $\mathbf{R}_{\lambda;\mathcal{A}}$.

***Rules*** Figure 5 shows the key rules of TaDA Live, explained below. The omitted rules can be found in the Appendix.

### 3.2 The While rule

The While rule proves termination of the only potentially non-terminating primitive construct of the language. The rule can be understood as follows. As in a standard Hoare proof, a *loop invariant* $P$ needs to hold initially and be re-established at the end of each iteration. $P$ is parametrised by an ordinal $\beta$ which will be equated by $P$ to some expression representing a *variant* for the loop. In a standard sequential proof, the variant is required to strictly decrease at each loop iteration. In a concurrent context, this condition would completely forbid the possibility of blocking, e.g. busy-waiting on a shared memory cell. We therefore require the definition of a *target* assertion $T$, which represents the "good" states for the while: states from which a loop iteration can make measurable progress. The body of the loop is required to terminate by strictly decreasing the variant only from states satisfying $T$, for the others it is sufficient to not increase the variant (the progress made so far should not be undone).

This is formalised in the last premise of the While rule. For an assertion $P$, we define $(1 \dot{\Rightarrow} P) = P$ and $(0 \dot{\Rightarrow} P) = \mathsf{emp}$. Therefore, the last premise can be seen as representing two premises in a single triple:

$$\forall \beta \le \beta_0.\ m; \lambda; \mathcal{A} \vdash \{P(\beta) \quad \wedge \mathbb{B}\}\ \mathbb{C}\ \{\exists \gamma.\, P(\gamma) \wedge \gamma \le \beta\}$$
$$\forall \beta \le \beta_0.\ m; \lambda; \mathcal{A} \vdash \{P(\beta) * T \wedge \mathbb{B}\}\ \mathbb{C}\ \{\exists \gamma.\, P(\gamma) \wedge \gamma < \beta\}$$

requiring the decrease of the invariant depending on having reached $T$ or not.

Then two conditions need to be checked: (i) *environment liveness:* that the environment will always eventually reach the target $T$ from states not satisfying $T$, and (ii) *bounded bad transitions:* that the number of times the environment can transition from a state satisfying $T$ to one not satisfying it, is bounded. Accordingly, the first premise in the rule is the *environment liveness condition.* To prove that the environment will make $T$ be reached in finite time, from any state not satisfying it, one needs to define an assertion $M(\alpha)$, called *environment progress measure*, with a free ordinal-valued logical variable $\alpha$. The environment liveness condition will check that the environment progress measure strictly decreases from any state not satisfying $T$, whenever some live transition is executed by the environment. Since $\alpha$ is well-founded, the measure cannot decrease forever and $T$ will eventually be reached. How the judgement $m; \lambda; \mathcal{A} \vdash \langle P(\beta) \,|\, L\rangle \xrightarrow{M}\!\!\!\!\twoheadrightarrow T$ can be proved will be explained in detail in Section 3.4.

The second premise (BoundBad) ensures that in the possible future from a state in the target $T$, if we reach a state not in the target ($\overline{T}$) the measure of progress $\alpha$ has strictly decreased. This condition effectively requires $M$ to also provide bounds on the number of times the environment can transition from "good" to "bad" states (cf. Fig. 3). The assertion $L$ is effectively framed from the while loop. This means it can be assumed to hold for the duration of the loop. This allows the proof of environment liveness to restrict the possible actions of the environment given the exclusive permissions that the current thread may be holding for the whole loop execution.

### 3.3 Other liveness-related rules

The Liveness check rule (LiveC) explains how to use a specification with a liveness assumption. When the liveness assumption can be proven to hold in the current proof context, the liveness requirement in the pseudo-quantifier can be dropped, thus declaring the operation unconditionally terminating in the current context.

The Parallel rule (Par) follows the standard pattern of concurrent separation logics: the precondition is separated in two preconditions one for each thread. Obligations that may be held locally at the beginning can be distributed to the two threads arbitrarily. Absence of deadlocks is enforced by the constraints on the layer $m$. Recall that the layer $m$

$$\frac{\forall \beta \le \beta_0.\ m;\lambda;\mathcal{A} \vdash \langle P(\beta) \mid L\rangle \xrightarrow{M} T \qquad \mathrm{BoundBad}^{\lambda}_{\mathcal{A}}(T,L,M) \qquad \mathrm{pv}(T,L,M)\cap \mathrm{mod}(\mathbb{C}) = \emptyset \\ \forall \beta \le \beta_0.\ \forall b \in \{0,1\}.\ m;\lambda;\mathcal{A} \vdash \{P(\beta) * (b \doteq T) \wedge \mathbb{B}\}\ \mathbb{C}\ \{\exists \gamma.\ P(\gamma) \wedge \gamma \le \beta * (b \doteq \gamma < \beta)\}}{m;\lambda;\mathcal{A} \vdash \{P(\beta_0) * L\}\ \texttt{while}(\mathbb{B})\{\mathbb{C}\}\ \{\exists \beta.\ P(\beta) * L \wedge \neg\mathbb{B} \wedge \beta_0 \ge \beta\}}\ \textsc{While}$$

where $\mathrm{BoundBad}^{\lambda}_{\mathcal{A}}(T,L,M) \triangleq \forall \alpha.\ \lambda;\mathcal{A} \vDash \mathbf{R}^{*}_{\lambda;\mathcal{A}}(L * M(\alpha) * T) \wedge \overline{T} \Rightarrow \left(\forall \alpha'.\ (M(\alpha') \Rightarrow \alpha' < \alpha) * \mathrm{True}\right)$

$$\frac{m \sqcap k;\lambda;\mathcal{A} \vdash \langle P_{\mathsf{h}} \mid L\rangle \xrightarrow{M} T \qquad \lambda;\mathcal{A} \vdash \forall x \in X.\ P_{\mathsf{a}}(x) * L * T \Rightarrow x \in X' \\ m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \mid P_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid Q_{\mathsf{a}}(x,y)\rangle}{m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X.\langle P_{\mathsf{h}} \mid P_{\mathsf{a}}(x) * L\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid Q_{\mathsf{a}}(x,y) * L\rangle}\ \textsc{LiveC}$$

$$\frac{\mathrm{fv}(R_{\mathsf{h}} * R_{\mathsf{a}}(x)) \cap \mathrm{mod}(\mathbb{C}) = \emptyset \qquad \lambda;\mathcal{A} \vDash \exists x \in X.\ R_{\mathsf{a}}(x)\ \mathrm{stable} \qquad \lambda;\mathcal{A} \vDash R_{\mathsf{h}}\ \mathrm{stable} \qquad \forall x \in X.\ \lambda;\mathcal{A} \vdash R_{\mathsf{h}} * R_{\mathsf{a}}(x) \rhd m \\ m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \mid P_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid Q_{\mathsf{a}}(x,y)\rangle}{m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} * R_{\mathsf{h}} \mid P_{\mathsf{a}}(x) * R_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) * R_{\mathsf{h}} \mid Q_{\mathsf{a}}(x,y) * R_{\mathsf{a}}(x)\rangle}\ \textsc{Frame}$$

$$\frac{\forall k \le m.\ k;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_j X'.\langle P_{\mathsf{h}}(k) \wedge k \le m \mid P_{\mathsf{a}}(k,x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(k,x,y) \mid Q_{\mathsf{a}}(k,x,y)\rangle}{\forall k \le m.\ m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_j X'.\langle P_{\mathsf{h}}(k) \mid P_{\mathsf{a}}(k,x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(k,x,y) \mid Q_{\mathsf{a}}(k,x,y)\rangle}\ \textsc{QL}$$

$$\frac{m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X''.\langle P_{\mathsf{h}} \mid P_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid Q_{\mathsf{a}}(x,y)\rangle \qquad X' \subseteq X'' \subseteq X}{m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \mid P_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid Q_{\mathsf{a}}(x,y)\rangle}\ \textsc{LiveW}$$

$$\frac{m;\lambda;\mathcal{A} \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \qquad \lambda;\mathcal{A} \vDash P_1\ \mathrm{stable} \qquad \lambda;\mathcal{A} \vdash Q_1 \rhd m \\ m;\lambda;\mathcal{A} \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\} \qquad \lambda;\mathcal{A} \vDash P_2\ \mathrm{stable} \qquad \lambda;\mathcal{A} \vdash Q_2 \rhd m}{m;\lambda;\mathcal{A} \vdash \{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

$$\frac{r \notin \mathcal{A} \qquad \mathcal{A}' = (r, x, X \twoheadrightarrow_k X', Y(x)), \mathcal{A} \qquad \{\,(x,y) \mid x \in X, y \in Y(x)\,\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^{*} \\ m;\lambda;\mathcal{A}' \vdash \{P_{\mathsf{h}} * \exists x \in X.\ \mathbf{t}^{\lambda}_{r}(x) * r \Mapsto \blacklozenge\}\ \mathbb{C}\ \{\exists x \in X, y \in Y(x), z.\ \mathbf{t}^{\lambda}_{r}(z) * Q_{\mathsf{h}}(x,y) * r \Mapsto (x,y)\}}{m;\lambda;\mathcal{A} \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \mid \mathbf{t}^{\lambda}_{r}(x) * \lfloor \mathrm{G} \rfloor_r\rangle\ \mathbb{C}\ \exists y.\langle Q_{\mathsf{h}}(x,y) \mid y \in Y(x) \wedge \mathbf{t}^{\lambda}_{r}(y) * \lfloor \mathrm{G} \rfloor_r\rangle}\ \textsc{MkAtom}$$

**Figure 5.** TaDA Live rules (selected)

$$\frac{m;\lambda;\mathcal{A} \vdash \langle P \mid L\rangle \xrightarrow{M} T : E(\alpha) \qquad \forall \alpha.\ \lambda;\mathcal{A} \vDash \left(((L * M(\alpha)) \wedge \overline{T}){\downarrow_\lambda} \twoheadrightarrow \mathrm{TotObl}\right) \Rightarrow E(\alpha) \\ \lambda;\mathcal{A} \vDash \exists \alpha.\ L * M(\alpha)\ \mathrm{stable} \qquad \lambda;\mathcal{A} \vdash P \Rightarrow P * \exists \alpha.\ M(\alpha) \qquad \mathrm{NonIncr}^{\lambda}_{\mathcal{A}}(T,L,M)}{m;\lambda;\mathcal{A} \vdash \langle P \mid L\rangle \xrightarrow{M} T}\ \textsc{EL}$$

$$\frac{\forall i,j,k \in \{1,2\}.\ m;\lambda;\mathcal{A} \vdash \left\langle P_i \,\middle|\, L_j\right\rangle \xrightarrow{M} T : E_k(\alpha)}{m;\lambda;\mathcal{A} \vdash \langle P_1 \vee P_2 \mid L_1 \vee L_2\rangle \xrightarrow{M} T : E_1(\alpha) \vee E_2(\alpha)}\ \textsc{ELCase} \qquad \frac{m;\lambda;\mathcal{A} \vdash \langle P \mid L\rangle \xrightarrow{M} T : E_1(\alpha)}{m;\lambda;\mathcal{A} \vdash \langle P \mid L\rangle \xrightarrow{M} T : E_1(\alpha) * E_2(\alpha)}\ \textsc{ELFrame}$$

$$\frac{\mathrm{PQLive}^{\lambda}_{\mathcal{A}}(m, P * L, k) \qquad \forall \alpha.\ E(\alpha) \Rightarrow \mathrm{PQProg}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, \alpha, f, T, L, S \cap X') \\ (r, x, X \twoheadrightarrow_k X', \_) \in \mathcal{A} \qquad S \subseteq \mathrm{AState}_r \qquad S \cap X' \neq \emptyset \qquad \forall \alpha.\ E(\alpha) \Rightarrow \mathrm{ASReach}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, L * M(\alpha), T, S)}{m;\lambda;\mathcal{A} \vdash \langle P \mid L * r \Mapsto \Diamond\rangle \xrightarrow{M} T : \exists x.\ \mathbf{t}^{\lambda'}_{r}(v,x) * E(\alpha)}\ \textsc{LiveA}$$

$$\frac{\mathrm{ObLive}^{\lambda}_{\mathcal{A}}(m, P * L, \mathrm{O}) \qquad \forall \alpha.\ E(\alpha) \Rightarrow \mathrm{ObProg}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, \alpha, T, M, L, \mathrm{O}) \qquad \mathrm{O}\ \mathrm{atom}}{m;\lambda;\mathcal{A} \vdash \langle P \mid L\rangle \xrightarrow{M} T : \exists x.\ \mathbf{t}^{\lambda'}_{r}(v,x) * \lceil \mathrm{O} \rceil_r * E(\alpha)}\ \textsc{LiveO}$$

$$\overline{T} \triangleq (T \Rightarrow \mathrm{False}) \qquad \mathrm{Reach}^{\lambda}_{\mathcal{A}}(T,I) \triangleq \left(\mathbf{R}_{\lambda;\mathcal{A}} \cap (\overline{T} \times \overline{T})\right)^{*}(I)$$

$$\mathrm{Inv}_{\lambda}(\mathbf{t}^{\lambda'}_{r}, v) \triangleq \begin{cases} \exists x.\ I(\mathbf{t}^{\lambda'}_{r}(v,x)) * \mathbf{t}^{\lambda'}_{r}(v,x) & \text{if } \lambda > \lambda' \\ \mathrm{emp} & \text{otherwise} \end{cases} \qquad \begin{aligned} \mathrm{ObLive}^{\lambda}_{\mathcal{A}}(m, Loc, \mathrm{O}) &\triangleq m > \mathrm{lay}(\mathrm{O}) \wedge \lambda;\mathcal{A} \vdash Loc \rhd \mathrm{lay}(\mathrm{O}) \\ \mathrm{PQLive}^{\lambda}_{\mathcal{A}}(m, Loc, k) &\triangleq m > k \wedge \lambda;\mathcal{A} \vdash Loc \rhd k \end{aligned}$$

$$\mathrm{NonIncr}^{\lambda}_{\mathcal{A}}(T,L,M) \triangleq \lambda;\mathcal{A} \vDash \forall \alpha.\ \left(\mathrm{Reach}^{\lambda}_{\mathcal{A}}(T, L * M(\alpha)) \Rightarrow \forall \alpha'.\ (M(\alpha') \Rightarrow \alpha' \le \alpha) * \mathrm{True}\right)$$

$$\mathrm{ASReach}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, I, T, S) \triangleq \lambda;\mathcal{A} \vDash \mathbf{R}^{*}_{\lambda;\mathcal{A}}(I) \Rightarrow (\exists s \in S.\ \mathbf{t}^{\lambda'}_{r}(v,s) * \mathrm{True})$$

$$\mathrm{ObProg}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, \alpha, T, M, L, \mathrm{O}) \triangleq \lambda;\mathcal{A} \vdash \exists \alpha'.\ \left((L * M(\alpha') * \alpha' \le \alpha * \mathrm{Inv}_{\lambda}(\mathbf{t}^{\lambda'}_{r}, v)) \wedge (\lceil \mathrm{O} \rceil_r * \mathrm{True})\right) \Rightarrow (\alpha' < \alpha \vee (T * \mathrm{True}))$$

$$\mathrm{PQProg}^{\lambda}_{\mathcal{A}}(\mathbf{t}^{\lambda'}_{r}, v, \alpha, T, M, L, G) \triangleq \forall s \in G.\ \lambda;\mathcal{A} \vdash \exists \alpha'.\ (L * M(\alpha') \wedge \alpha' \le \alpha * \mathbf{t}^{\lambda'}_{r}(v,s)) \Rightarrow (\alpha' < \alpha \vee (T * \mathrm{True}))$$

**Figure 6.** Environment Liveness rules

in the context of the triples is a strict upper-bound on the layers that can be assumed live in the proof of $\mathbb{C}_i$. At the end of its execution, $\mathbb{C}_2$ can only hold obligations of layer higher than $m$, i.e. only obligations on which $\mathbb{C}_1$ cannot rely. This ensures that $\mathbb{C}_1$ cannot keep waiting for obligations that are not live because $\mathbb{C}_2$, which terminated but waiting to join with $\mathbb{C}_1$, is blocking them indefinitely. Symmetric constraints are applied to $\mathbb{C}_1$.

The Frame rule restricts the layers of the frame: we can only ignore obligations we currently hold if we cannot rely on the liveness of any obligation that may depend on them. From the context of the triple in the premise, we know $\mathbb{C}$ can be proved terminating without assuming live any obligation of layer $m$ or higher, so we can safely frame obligations with layer $m$ or higher.

The Quantify layer rule (QL) is helpful when a layer is linked to some value in the local state, as signified by the dependency of the pre/post-conditions on the layer $k$. If one can prove the command terminating for every $k \leq m$ from the states where $k \leq m$ holds, without assuming liveness of layers $k$ or higher, then the command will terminate for every $k \leq m$ without assuming liveness of layers $m$ or higher.

The last rule dealing with liveness assumptions is the Liveness weakening rule (LiveW). It represents an immediate consequence of our choices for the meaning of specifications. Every environment satisfying the liveness assumption $\mathbb{W}x \in X \twoheadrightarrow_k X''$ will also satisfy the assumption $\mathbb{W}x \in X \twoheadrightarrow_k X'$.

### 3.4 The environment liveness condition

The judgement $m; \lambda; \mathcal{A} \vdash \langle P \,|\, L\rangle \xrightarrow{M}\!\!\!\!\!\rightarrow T$ is called the *environment liveness condition*, and is central in our system. It can be proven using a dedicated proof system (Fig. 6) that breaks down the proof in more manageable sub-proofs. The environment liveness judgement is proven using the EL rule. The lower premises check that $M$ is a meaningful measure of environment progress: it is meaningful to measure it at any point in time when $L$ holds, it can be measured at the beginning, and it will not be increased by any environment action (if $T$ stays false).

The top right premise requires the definition of an assertion $E(\alpha)$ which captures facts we know about the obligations that must be held by the environment. The requirement on $E$ can be read as follows. Consider an arbitrary point in the traces. By assumption we know $L$ is going to hold and we can measure $M(\alpha)$. Consider the states which do not satisfy the target assertion $T$, and open the relevant region interpretations. What we obtain are states where we see all the obligations that must be held locally or by the region. TotObl is the set of states where the total obligation $\mathbf{1}$ is held locally for every region. By using the separating implication $-\!\!*$ we obtain an assertion capturing which obligations must be held by the environment: since the composition of all the obligations of a region held by anyone always amounts to the total, the environment must hold the fraction of the total that is not held locally nor in the region interpretation.

Given such an assertion about the environment, we have to prove the sub-judgment $m; \lambda; \mathcal{A} \vdash \langle P \,|\, L\rangle \xrightarrow{M}\!\!\!\!\!\rightarrow T : E(\alpha)$, which requires to prove environment liveness for all $\alpha$, given the environment satisfies $E(\alpha)$. In general, we may want to invoke different liveness assumptions in different states. We therefore offer rules to do a case split on the states to be considered (ELCase, ELFrame).

Once we isolated all the distinct cases, for each we need to show that we can find a liveness assumption, in the form of pseudo-quantifiers in the proof context $\mathcal{A}$ or obligations, which can be assumed live (given their layers), and can be shown to make the environment progress measure strictly decrease when fulfilled. We provide two rules that check just this, one for obligations (LiveO) and one for pseudo-quantifiers (LiveA). The auxiliary definitions are in Fig. 6. PQLive and ObLive check that the layer of the assumption can be assumed live: it must be lower than the upper-bound imposed by $m$ and lower than the layer of any obligation we may be holding locally. ASReach asserts that $S$ is an over-approximation of the reachable abstract states of a region for which we have a pseudo-quantifier liveness assumption in the context. PQProg checks that by reaching any state that some pseudo-quantifier assumption promises to eventually reach ($S \cap X'$), the measure of environment progress is strictly decreased. ObProg checks that the same happens when some live obligation O is returned to the region.

Note how rule LiveO requires O to be an atom: we cannot assume a composite obligation $\mathrm{O}_1 \bullet \mathrm{O}_2$ will be fulfilled as a whole, even when each $\mathrm{O}_i$ is live.

## 4 Case studies and evaluation

With TaDA Live, we can prove all the terminating examples of [11, 13, 15, 18, 19, 24], including spin locks, ticket locks, MCS locks, spin and blocking counters, blocking queues/stacks, and lock-coupling sets. All our proofs are compositional. See Appendix for details. Here, we show that the CLH lock satisfies the specification of a fair lock. We also discuss limitations of the TaDA Live proof system.

### 4.1 Proof of CLH lock

We describe a TaDA Live proof of the CLH lock implementation. The full specification for a fair lock is:

$$\top \vdash \mathbb{W}l \in \{0,1\} \twoheadrightarrow_{\bot} \{0\}.\langle \mathsf{L}(s, \mathrm{x}, l)\rangle\ \mathtt{lock(x)}\ \langle \mathsf{L}(s, \mathrm{x}, 1) \wedge l = 0\rangle$$
$$\bot \vdash \langle \mathsf{L}(s, \mathrm{x}, 1)\rangle\ \mathtt{unlock(x)}\ \langle \mathsf{L}(s, \mathrm{x}, 0)\rangle$$

To verify the implementation, we define $\mathsf{L}(r, x, l) \triangleq \exists o \in \mathbb{N}.\ \mathbf{clh}_r(x, l, o) * \lfloor \mathrm{G} \rfloor_r$, where the **clh** region type is defined in Fig. 7. Let $x \in X, s, t \in X^*$. We write $x \oplus s$, $s \oplus x$, and $s \oplus t$ for prepend, append, and concatenation, respectively; $|s|$ is the length of $s$, and $s(i) = x$ states that the $i$-th element (from 0) in $s$ is $x$ and $i < |s|$; $\mathrm{fst}(s)$ and $\mathrm{last}(s)$ are the first and the last element of $s$, respectively. Figure 11 shows a proof outline for

$$I(\mathbf{clh}_r(x, l, o)) \triangleq \exists ns \in \mathrm{Addr}^+, t \in \mathbb{N}.\ x \mapsto \mathrm{fst}(ns), \mathrm{last}(ns) * \mathrm{fst}(ns) \mapsto l * \mathrm{ones}_1(ns) * \lfloor \mathrm{Q}(ns, o) \rfloor_r * \lceil \mathrm{O}(o, t) \rceil_r * \mathrm{True} \wedge t - o = |ns| - 1$$

**Interference:** $\mathrm{G} : (0, o) \rightsquigarrow (1, o+1) \quad \mathrm{G} : (1, o) \rightsquigarrow (0, o)$

**Guard algebra:** for any $p, c \in \mathrm{Addr}$, $ns \in \mathrm{Addr}^*$, $o, t \in \mathbb{N}$

$$\mathrm{Q}([p, c] \oplus ns, o) \bullet \mathrm{T}(p, c, o+1) = \mathrm{Q}(c \oplus ns, o+1)$$
$$\mathrm{Q}(ns, o) \bullet \mathrm{T}(p, c, t) \text{ defined} \iff ns(t - o - 1) = p \wedge ns(t - o) = c$$
$$\mathrm{Q}(ns \oplus p, o) = \mathrm{Q}(ns \oplus [p, c], o) \bullet \mathrm{T}(p, c, o + |ns| + 1)$$
$$\mathrm{G} \bullet \mathrm{G} \text{ undefined}$$

$$\mathrm{ones}_k(ns) \triangleq ns(k) \mapsto 1 * \cdots * ns(|ns| - 1) \mapsto 1$$

**Obligation algebra:** for any $o, o', t, t' \in \mathbb{N}$

$$\mathrm{O}(o, t) = \mathrm{O}(o, t+1) \bullet \mathrm{P}(t) \qquad \mathrm{O}(o, o) = \mathrm{O}(o', o')$$
$$\mathrm{O}(o+1, t) = \mathrm{O}(o, t) \bullet \mathrm{P}(o+1) \qquad \textbf{Total: } \mathrm{O}(0, 0)$$
$$\mathrm{O}(o, t) \bullet \mathrm{P}(t') \text{ is defined} \iff o \le t' < t$$
$$\mathcal{L} \triangleq \mathbb{N} \cup \{\top, \bot\} \quad \mathrm{lay}(\mathrm{O}(o, o')) = 0 \quad \mathrm{lay}(\mathrm{P}(t)) = t$$

**Figure 7.** Interpretation, interference, guard and obligation algebras of the **clh** region.

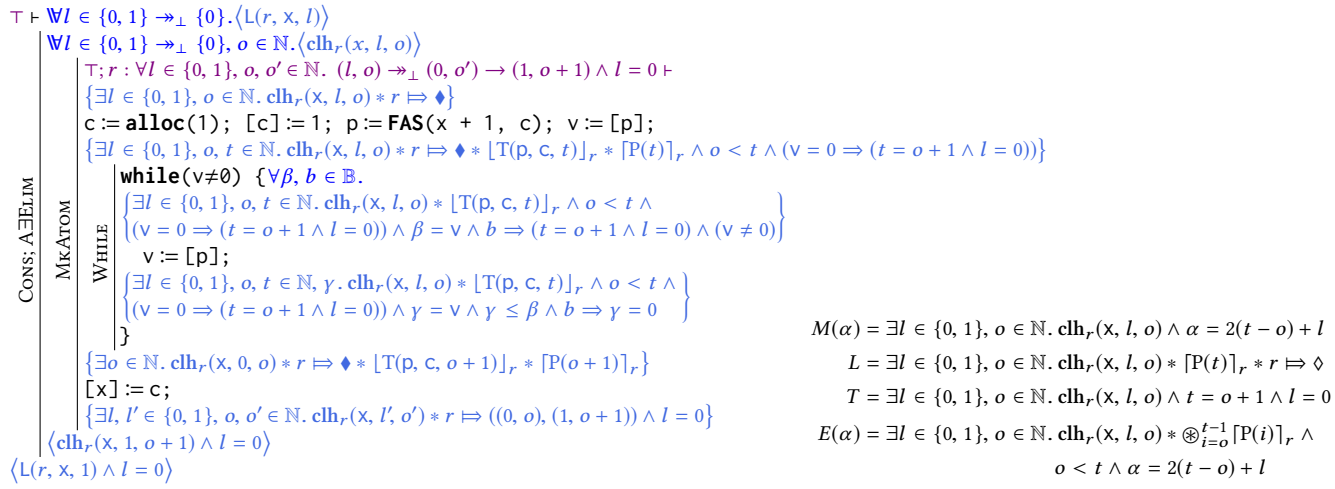



**Figure 8.** Proof outline of the CLH `lock` implementation, and parameters for the While rule application.

the `lock` operation. Most proof steps do not involve progress reasoning and are normal TaDA derivations (the Cons and A∃Elim rules are in Appendix). The guard/obligation algebra is designed so that by enqueueing a node $c$ in the virtual queue $ns$ one obtains the local ghost state $\mathrm{T}(p, c, t)$ and $\mathrm{P}(t)$ where $p$ is the predecessor, and $t$ is the position of $c$ in the queue. The ghost state $\mathrm{Q}(ns, o)$ keeps track of the position $o$ of the first still-to-be-served member of the queue, and $ns$, the queue from $o$. The information in the ghost state implies $o < t$ for the duration of the while. The obligation $\mathrm{P}(t)$ can only be put back in the region by setting the head to $c$. The key step in the proof is proving that the while terminates. In Fig. 12 we show the parameters of the While rule application. The target states are those where $t = o + 1$ and $l = 0$, (which implies that `p` points to 0). In those states we would make progress by reading 0 into `v` thus bringing down the variant from 1 to 0 (after which we must terminate). Once `p` is set to 0, it will stay 0: the BoundBad condition is trivially satisfied, since once we reach $T$ we cannot leave it.

The environment liveness condition is checked as follows. The $E(\alpha)$ assertion of Fig. 12 says that, given $t$ and $o$, we know that all the obligations $P(o) \ldots P(t-1)$ are in the environment. We examine two cases: $E(\alpha) \wedge l = 0$ and $E(\alpha) \wedge l = 1$. In the first case, we can invoke the presence of an obligation $P(i)$ with $o \le i < t$ in the environment. To apply LiveO we check that the obligation is live, which follows from $i < t$, and that the only way for the obligation to be put back in the region is if $i \le o$ which makes the environment progress measure $2(t - o) + l$ decrease.

In the second case, we can invoke the pseudo-quantifier assumption which promises eventually $l = 0$, which can be true only if $o$ strictly increases, making the environment progress measure decrease.

### 4.2 A client of a fair lock

We briefly explain how Example 2.1 can be proven when assuming it uses fair locks. The full proof can be found in the Appendix. The two threads share the cells `x` and `done`, so we wrap them in a region $\mathbf{c}_r(\mathtt{x}, \mathtt{done}, l, d)$ where $l$ and $d$ are the values at `x` and `done` respectively. We set guard and obligation algebra to the same PCM with axioms $\mathrm{U} = \mathrm{L} \bullet \mathrm{K}$, $\mathrm{D} = \mathrm{S} \bullet \mathrm{W}$, total $\mathrm{U} \bullet \mathrm{D}$, $\mathrm{lay}(\mathrm{U}) = \mathrm{lay}(\mathrm{D}) = 0$, $\mathrm{lay}(\mathrm{S}) = 2$, and $\mathrm{lay}(\mathrm{K}) = 1$. The interference is $\mathrm{S} : (l, d) \rightsquigarrow (l, \mathtt{true})$ and $\mathbf{0} : (l, d) \rightsquigarrow (l', d)$. The interpretation makes it so S and K are not in the region if `done` and the lock at `x` are `false` and locked, respectively: $I(\mathbf{c}_r(\mathtt{x}, \mathtt{done}, l, d)) \triangleq \mathsf{L}(x, l) * \mathtt{done} \mapsto d * (\lceil \mathrm{U} \rceil_r \wedge l = 0) \vee (\lceil \mathrm{L} \rceil_r \wedge l = 1) * (\lceil \mathrm{D} \rceil_r \wedge d) \vee (\lceil \mathrm{W} \rceil_r \wedge \neg d)$. Initially we have $\exists l.\ \mathbf{c}_r(\mathtt{x}, \mathtt{done}, l, \mathtt{false}) * \lceil \mathrm{S} \rceil_r$ and we apply the Par rule (with layer $m = 3$), giving $\lceil \mathrm{S} \rceil_r$ to the thread on the left. The lock acquisition on the left thread is proven using LiveC. The environment liveness condition is satisfied because if the lock is locked, the environment holds K, which

is live because lay(κ) < lay(s); κ can only be put back in the region by unlocking the lock, bringing us to the target state. Note how we are not required to check BoundBad so the client does not need to show the lock is not reaquired unboundedly many times. As a result of locking, we obtain κ. Then by setting done we put the s back in the region, and after unlocking x we also lose κ, leaving us with no obligations and thus satisfying the layer condition of the Par rule.

For proving the thread on the right, we apply the While rule. The target is states where done is true. If done is false, the environment holds s which is live because we do not hold any obligation (and $m = 3$ in the context). s is fulfilled by bringing us to the target state. Here it is important to prove BoundBad which holds trivially since the interference ensures that once done is true, it will remain so. The loop body is proven by the LiveC rule as in the left thread case. Note that inside the proof of the loop body, we do acquire κ, but since we fulfill it without relying on d before looping, there is no circularity of dependencies.

**Remark 2.** Consider Example 2.1 with a spin lock at x. We would not be able to use the `lock` specification, since we cannot put a bound on the number of calls we will make: we would not be able to pick an initial ordinal budget and spend a non trivial amount at every call.

### 4.3 Limitations

TaDA and TaDA Live do not support helping/speculation. Such patterns are challenging for the identification of the linearization point, which is entirely a safety property. Possible extensions to TaDA that could support such patterns are discussed in [3]. Such extensions are orthogonal to the termination argument. We therefore choose, in line with related literature, to explore termination in a simpler logic.

The While rule might require extension. Consider a loop invariant asserting the possession of obligation κ. We cannot distinguish, by only looking at the specification of the loop body, the case where κ is continuously held throughout the execution of the body, from the case where κ is fulfilled and then reacquired before the end of an iteration. The current While rule conservatively rules out the use of assumptions with layer higher than or equal to lay(κ); doing otherwise would be unsound in the case κ is held continuously. A solution would be to introduce an assertion live(κ), certifying that an obligation is fulfilled at some point in a block of code. This would allow the While rule to only forbid layers which may depend on obligations one holds in the loop invariant and for which it was not possible to prove live(κ).

A more interesting limitation comes from our approach to specifying impedence. For non-blocking programs, the ordinal-based approach is complete. It is not complete for blocking programs: consider $\mathbb{C}_2 \triangleq (\mathbb{C}_1 \parallel \texttt{[done]} := \texttt{true})$ where $\mathbb{C}_1$ is the program in Example 2.1 with a spin lock. Scheduler fairness guarantees $\mathbb{C}_2$ will terminate. The specification of spin lock, however, states that every call to `lock` needs to consume budget, forcing the client to provide an upper bound for the total number of calls to initialise the budget. Unfortunately, $\mathbb{C}_2$ will call `lock` an arbitrary unbounded number of times, determined only by the choices of the scheduler. It is, thus, not possible to provide the initial budget, and TaDA Live cannot prove that the program terminates. The impedance on the lock is only relevant when the client is unblocked (i.e. done is true) but the specifications do not allow for the distinction. To accommodate this behaviour, we could introduce $\alpha(\mathrm{S})$ to represent a prophecy of the number of steps it will take to fulfill *live* obligation s. This would solve the problem for $\mathbb{C}_2$, because $\alpha(\mathrm{s})+1$ (where s is fulfilled by setting done to true) would be the required budget. To the best of our knowledge, none of the logics that can prove total atomic specifications can handle this example.

## 5 Soundness

To simplify notation, we switch from judgements in infix form to the prefix form $m; \lambda; \mathcal{A} \vdash \mathbb{C} : \mathbb{S}$ where $\mathbb{S}$ is of the form $\forall x \in X \twoheadrightarrow_k X'. \langle P_\mathrm{h} \mid P_\mathrm{a}(x) \rangle\ \exists y. \langle Q_\mathrm{h}(x,y) \mid Q_\mathrm{a}(x,y) \rangle$.

The semantic of atomic triples in TaDA-Live are defined with respect to a trace semantic for programs $[\![\mathbb{C}]\!] \subseteq \mathsf{Trace}$. All the traces in $[\![\mathbb{C}]\!]$ are infinite and interleave steps from the command with arbitrary environmental steps. We say an infinite trace is *locally terminating* if, after some finite prefix, the trace only consists of environment steps. A trace is *fair* if every local active thread and the environment always perform a step in the future. To check wether a trace satisfies a specification $\mathbb{S}$, we first instrument it with ghost state obtaining an instrumented trace $it$. Then we check that $it$ satisfies the safety and liveness constraints of $\mathbb{S}$ as follows.

The safety constraints are from TaDA, and intuitively require that, starting from a state satisfying the precondition, under an environment that satisfies the rely, the operation will satisfy the guarantee and, once terminated, will produce a state satisfying the postcondition. The liveness constraints require that $\big(\mathit{fair}(st) \wedge \mathit{safe}(it) \wedge \neg \mathit{locterm}(it)\big) \Rightarrow \neg \mathit{envLive}^{m;k}_{\mathcal{A}}(it)$ holds. This condition states that, if a trace is fair and respects the pre/post-conditions and rely/guarantee safety constraints required by the specification, but is *not* locally terminating, then the environment must have violated the liveness assumptions of the specification. An environment *satisfies* the liveness assumptions, $\mathit{envLive}^{m;k}_{\mathcal{A}}$, if the environmental steps of a safe instrumented trace satisfy:

- Every atomic obligation that the environment holds is eventually returned to the shared state, unless an obligation of lower layer is held continuously by a local thread.
- The liveness requirement of every pseudoquantifier is eventually satisfied, including the current one, unless an obligation, of layer less than the $k$ of that pseudoquantifier, is continuously held by a local thread.

Using these traces accepted by a specification, we define the semantics of a triple $[\![\mathbb{S}]\!]^m_{\lambda;\mathcal{A}}$ to be the set of traces accepted by the triple. Thus, if a command can be shown to satisfy the specification, its behaviour is contained in $[\![\mathbb{S}]\!]^m_{\lambda;\mathcal{A}}$.

**Theorem 5.1.** *If* $m;\lambda;\mathcal{A} \vdash \mathbb{C} : \mathbb{S}$ *then* $[\![\mathbb{C}]\!] \subseteq [\![\mathbb{S}]\!]^m_{\lambda;\mathcal{A}}$.

## 6 Related Work

***Primitive Blocking*** There has been work on termination and deadlock-freedom of concurrent programs with primitive blocking constructs. Starting from the seminal work of [16], the idea of tracking dependencies between blocking actions and ensuring their acyclicity has been used to prove deadlock-freedom of shared-memory concurrent programs using primitive locks and (synchronous) channels [1, 17]. Similar techniques have been used in [11] to prove global deadlock-freedom (a *safety* property requiring that at least some thread can take a step), and [13] to prove termination. This entire line of work assumes the invocation of lock/channel primitives as the only source of blocking. As a consequence, this methodology provides no insight on the issue of understanding ad-hoc blocking: that is, blocking that arises by arbitrary busy waiting and shared memory interference. We focus on a language without primitive blocking; one of our fundamental contributions is how to represent abstract blocking in specifications, and how to link the never-disabled primitive operations with the abstract blocking in the specifications. Our solution uniformly handles programs that mix blocking primitives and ad-hoc synchronisation patterns. The notion of "obligations" found in [1, 11, 13, 17] is only superficially related to our obligations (see the Appendix).

***History-based methods*** The CertiKOS project [10, 15] developed mechanised techniques for the specification and verification of fine-grained low-level code with explicit support for abstract atomicity and progress verification. The approach is based on *histories*: the abstract state is a log of the abstract events of a trace; and the specification of an atomic operation inserts exactly one event in the log. Local reasoning is achieved by rely/guarantee through complex automata product constructions. The framework is very expressive, with the downside that specifications are more complex and difficult to read, and verification requires manipulation of abstract traces/interleavings.

Our work is similar in aim and scope, but our strategy is different: we try to specify/verify programs using the minimal machinery possible, keeping the specifications as close to the developer's intuition as we can. As a result, our specifications are more readable (compare our fair lock specifications with the 30-lines-long one of Fig. 7 in [15]), and our reasoning is simpler (the layered obligation system leads to a more intuitive proof compared to the proof of MCS locks in [15]).[2]

[2]The proof is a variation of the one for CLH, see Appendix.

***Higher-order logics*** Iris [14] has been extended to reason about termination in [24, 25]. The proof system's goal is to establish a non-contextual refinement between two programs, one acting as a specification and one as an implementation. There is no support for abstract atomicity in the presence of blocking: the authors prove a CLH lock correct by showing it refines a ticket lock [25, Appendix B.2]. A crucial shortcoming of using a non-contextual refinement is that the approach is not modular: the refinement concerns two closed programs and cannot be reused when the code is part of some larger program (like a module and its clients).

***Contextual refinement*** There has been work on extending linearizability, characterised as a contextual refinement, to support reasoning about progress properties, e.g. [9], but supporting atomic specifications only for non-blocking operations. Liang et al. [20] study the exact relationship between common progress properties of fine-grained operations and contextual refinement. We intend to study the contextual refinement induced by the trace semantics of TaDA Live specifications in future work.

***LiLi*** The work closest to ours is LiLi [18, 19]. LiLi was the first program logic to prove total specifications for linearizable concurrent objects with internal blocking [18]. LiLi is also a concurrent separation logic, but proves linearizability via contextual refinement: the specifications are expressed as atomic programs and the verification proves a refinement relation between these programs and the implementation. Recently, LiLi was extended to handle external blocking [19]. Although we share most of our goals with LiLi, our approach is radically different: (i) LiLi's specifications of blocking operations in [19] are not truly atomic; and (ii) TaDA Live's verification is more compositional.

*Specifications* To specify an abstractly atomic blocking operation, LiLi introduces a primitive-blocking specification construct: $\mathbf{await}(\mathbb{B})\{\mathbb{C}\}$ executes $\mathbb{C}$ atomically if scheduled in a state where $\mathbb{B}$ is true. As both fair- and spin-lock acquisition are specified using $\mathbf{await}(l = 0)\{l := 1\}$ the specifications include a flag to indicate whether the **await** should be considered starvation-free (SF) or only deadlock-free (DF). LiLi describes only how to verify an implementation against these specifications; client reasoning is not handled. The contextual refinement result, however, allows using the specifications of a module A to prove correctness of a module B built on top of A by taking the verified specifications of A's operations, which are in the form $\mathbf{await}(\mathbb{B})\{\mathbb{C}\}_{\mathsf{SF/DF}}$, and transforming them into *non-atomic* programs which are inlined in B in place of the calls to A's operations, before proving B correct. For example, a fair lock ($\mathbf{await}(l = 0)\{l := 1\}_{\mathsf{SF}}$) will be transformed into a program with a global queue of thread identifiers, and locking would correspond to two separate events: the request of the lock by enqueuing of the current thread id in the queue, and the lock acquisition

once the current thread is the head of the queue. In other words: LiLi's specifications are not truly atomic. The most evident disadvantage of this approach is proof duplication: starvation-freedom needs to be proven for the implementation, but is only indirectly represented through the operational behaviour of the wrapper. The client proof would need to translate back, with another starvation-freedom proof, the progress properties of the wrapper into logical facts that can be used in the client proof.

We believe that representing blocking with environment liveness assumptions (through $\twoheadrightarrow$) and bounded impedence assumptions (though ordinals) is the key to solving this issue. TaDA Live does not distinguish between client and library verification: it does both *uniformly* in a single system, demonstrating the generality and usefulness of its specifications.

*Verification* Our environment liveness condition was informed by LiLi's *definite progress* condition. In LiLi, progress rely/guarantee is expressed through *definite actions* of the form $P \rightsquigarrow Q$, intuitively requiring that when the system is in a state satisfying $P$, eventually a state satisfying $Q$ will be reached. Our obligations are similar in spirit to definite actions, but attain higher locality of the argument. First, in LiLi, cycles in the argument are avoided by having $P$ in a definite action unambiguously assign the responsibility of fulfilling the action to some thread. This is done by assigning unique thread ids and keeping a virtual queue of threads ordered by dependency in ghost state (maintained with ghost code). This mechanism requires the ghost state to represent a very global view of the state, and the progress reasoning will involve all threads at the same time. Our PCM-based obligations do not require ghost code, and allow dependencies to be represented as locally as possible: only threads that are *directly* dependent will be reasoned about together.

Second, typically, proofs require a number of inter-dependent definite actions. As LiLi does not have a layering structure, one cannot describe each action separately, and then describe their dependences. Instead, one needs to lump them together in a chain of definite actions precisely describing all possible interactions between these actions. This approach makes the proofs scale poorly as the number of inter-dependent actions increases. Our layered obligations solve this problem elegantly: the liveness invariant allows us to abstract away *how* the environment chooses to fulfill an obligation using lower layer obligations. See Appendix for details.

## 7 Conclusions and Future Work

We have introduced TaDA Live, a sound separation logic for reasoning compositionally about the termination of fine-grained concurrent programs. We have illustrated the subtlety of our reasoning using a spin logic and a CLH lock. We have given an in-depth evaluation of our work, proving many examples from the literature in the Appendix and also pointing out limitations. We believe our work provides a substantial contribution to the understanding of compositional reasoning about progress properties for fine-grained concurrent algorithms.

In future, we plan to study the notion of contextual refinement implied by our semantic triples. This would allow us to integrate our proof techniques in a refinement calculus, following the approach of TaDA Refine [22]. This should simplify the proof rules for TaDA Live. It should also give us mechanisms for extending our compositional reasoning beyond termination, towards general liveness for reactive systems. We would, eventually, like to provide a semi-automatic verification tool for establishing compositional progress properties for fine-grained concurrent programs. There are such tools for safety properties: CAPER [7] based on CAP [8]; and Starling [26] based on views [6]. Our plan is to move first to a refinement calculus, where the rules should be simpler, and then explore a verification tool.

## Acknowledgments

We would like to thank Hongjin Liang, Xinyu Feng, Shale Xiong and Petar Maksimovic, for the helpful discussions and feedback.

## References

[1] Pontus Boström and Peter Müller. 2015. Modular Verification of Finite Blocking in Non-terminating Programs. In *ECOOP*. 639–663.

[2] Stephen D. Brookes. 2004. A Semantics for Concurrent Separation Logic. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 16–34.

[3] Pedro da Rocha Pinto. 2016. *Reasoning with Time and Data Abstractions*. Ph.D. Dissertation. Imperial College London.

[4] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014–Object-Oriented Programming*. Springer Berlin Heidelberg, 207–231.

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 176–201.

[6] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*. ACM, 287–300.

[7] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 420–447.

[8] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP (Lecture Notes in Computer Science)*, Vol. 6183. Springer, 504–528.

[9] Alexey Gotsman and Hongseok Yang. 2011. Liveness-Preserving Atomicity Abstraction. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. 453–465.

[10] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. ACM, 646–661.

[11] Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 415–441.

[12] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[13] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. 2018. Modular Termination Verification of Single-Threaded and Multithreaded Programs. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 12:1–12:59.

[14] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

[15] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock – Layer by Layer. In *APLAS (Lecture Notes in Computer Science)*, Vol. 10695. Springer, 273–297.

[16] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 4137. Springer, 233–247.

[17] K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *ESOP (Lecture Notes in Computer Science)*, Vol. 6012. Springer, 407–426.

[18] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 385–399.

[19] Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. *PACMPL* 2, POPL (2018), 20:1–20:31.

[20] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. 2013. Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 8052. Springer, 227–241.

[21] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 290–310.

[22] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A Concurrent Specification of POSIX File Systems. In *ECOOP (LIPIcs)*, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 4:1–4:28.

[23] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 49–67.

[24] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 909–936.

[25] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. *CoRR* abs/1701.05888 (2017).

[26] Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. 2017. Starling: Lightweight Concurrency Verification with Views. In *CAV (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 544–569.

# Appendix

## A Further explanation of the methodology

Consider the implementation of a data structure D with one operation op which internally uses a CLH lock to protect access to some shared memory: `op(x){lock(x);...unlock(x);}`. We abstract away from the specifics of the data structure and assume it is described by some abstract state $a \in A$. By writing $\mathsf{D}(x, a)$ we assert the existence of this data structure at address $x$ with abstract state $a$. We assume the abstract state is completely unrelated to the state of the lock: the lock is used purely for synchronisation over some shared resource, and its state should not leak to the client. op is blocking because its implementation uses blocking operations. This blocking is however resolved entirely internally: there is no specific pattern of usage for op that the client needs to respect to avoid being blocked. We therefore call this kind of blocking "internal".

Assume $\vdash \mathbb{V}a \in A.\langle \mathsf{D}(x, a)\rangle$ `op(x)` $\langle \mathsf{D}(x, f(a))\rangle$. Note how the pseudo-quantifier is not imposing liveness constraints on the abstract state. This is equivalent to writing $\mathbb{V}a \in A \twoheadrightarrow A$. The crucial step in the verification of op against its total specification is the proof that the call to `lock(x)` will terminate. Intuitively, the lock acquisition does terminate because any concurrent access to the lock is via D's operation op, and op always eventually releases the lock. Since this protocol should not be leaked to the client through the specification, the liveness condition "the lock is always eventually unlocked" needs to be represented as a "liveness invariant" in the proof of correctness of the data structure.

To introduce how TaDA Live formalises these liveness invariants, we give a quick overview of the TaDA methodology, and then introduce the novelties of TaDA Live. TaDA puts a strong emphasis on abstraction boundaries for modules: the specification should state the behaviour at the level of abstract state manipulations, so that any concrete implementation can make very different choices for how the abstract state is represented and manipulated in memory, without a need to reprove the client correctness for a new implementation. A further observation of TaDA is that to a data abstraction corresponds a *time abstraction*: what appear to be many steps at the implementation level, may appear as a single atomic step on the abstract state. Atomic triples express this relationship directly: $\vdash \mathbb{V}x \in X.\langle P(x)\rangle\ \mathbb{C}\ \langle Q(x)\rangle$ says that, the execution $\mathbb{C}$ will be seen as an atomic update from $P(x)$ to $Q(x)$ to anybody not able to distinguish between states satisfying $P$ (up to changes of $x$).

TaDA is based on (concurrent) separation logics, enriched with concurrent abstract predicates [8]. In this framework, in addition to assertions about contents of heap cells and their separation, one can define so-called *abstract predicates*: predicates that, to the client appear as uninterpreted, and are defined by the implementation so that their axioms are guaranteed to hold by virtue of the implementation's code. This supports data abstraction by allowing equivalent implementations to give different definitions, to the same abstract predicates interface to the client. For example, a data-structure implementing a list could be specified using an abstract predicate $\mathsf{List}(\mathtt{x}, lst)$ representing the existence of a list at address x with the sequence of elements *lst* as an abstract state. Then the implementation can choose to realise the list using a linked list, a double linked list, etc. freely; to each of these implementation choices will correspond a different internal definition of the List predicate, linking the abstract sequence of elements *lst* to the contents of the (private) concrete heap cells maintained by the module.

To support concurrency, data structures need to be shared between threads. Using just separation, different threads can only access disjoint portions of the heap. To solve this problem, TaDA uses the concept of *region*: a portion of the heap that is declared shared explicitly. A region type $\mathbf{t}$ specifies two crucial pieces of information that enable coordinated sharing:

1. an *interpretation* $I_{\mathbf{t}}$, which expresses the link between some abstract state and its concrete memory implementation; this is akin to what is commonly known as an *object invariant*: it describes the states of the memory which can correspond to a valid abstract state. Any (atomic) manipulation of the region can rely

on the memory representing some valid abstract state, and in return needs to modify the region so that the memory still represents a valid state.
2. an *interference relation* $\mathcal{T}_{\mathbf{t}}$, which captures a region-local rely/guarantee relation, that is an over-approximation of the possible concurrent modifications to the abstract state.

For our hypothetical data structure D, for example, we introduce a *region type* $\mathbf{d}$ that expresses what it means for a portion of memory to represent that data structure in a specific implementation. A region assertion $\mathbf{d}_r(x, a)$ represents an instance of the region $\mathbf{d}$ at address $\mathtt{x}$ with abstract state $a$. For technical reasons, we associate to each instance of the region a unique identifier $r$. The abstract predicate D in this case is a simple wrapper $\mathsf{D}(x, a) \triangleq \mathbf{d}_r(x, a)$. The interpretation of the region will look like $I(\mathbf{d}_r(x, a)) \triangleq x \mapsto y, \ldots * \exists l.\ \big(\mathsf{L}(y, l) * (\lceil \mathtt{R} \rceil_r \iff l = 0)\big) * \cdots$ The definition states that $x$ points to some cells, the first of which has value $y$. At address $y$ there is a (fair) lock $\mathsf{L}(y, l)$ with some abstract state $l$. To explain the rest of the interpretation, we need to introduce how TaDA represents ghost state. In the case of D, for example, Since the state of the lock cannot be exposed in the abstract state of the region, some internal ghost state is required to represent the lock state. TaDA uses *guard algebras* as a general way to represent ghost state. A guard algebra is a set of *guards* equipped with a partial binary operation $\bullet$ which, when defined, satisfies the axioms of a commutative monoid (with identity $\mathbf{0}$), i.e. it forms a *partial commutative monoid* (PCM). Given a guard $\mathtt{G}$ and a region identifier $r$, the assertion $\lfloor \mathtt{G} \rfloor_r$ states that we currently "hold" the guard $\mathtt{G}$ belonging to region $r$. Using these assertions, the interpretation of a region can link guards to the memory contents of that region. TaDA guards, however, can only express safety information, i.e. information about the past of the computation. TaDA Live uses, in addition to guards, the concept of *obligation algebra* which is a PCM with some additional structure which we introduce gradually. They are linked to the memory contents of a region in the same way as guards. An obligation assertion $\lceil \mathtt{O} \rceil_r$ states that we hold an obligation $\mathtt{O}$ belonging to the region $r$. The interpretation of $\mathbf{d}$, for example, states that the regions invariant holds the obligation $\mathtt{R}$ (for release) when the state of the lock is 0. By design, obligations cannot be destroyed: the obligation algebra contains a special element $\mathbf{1}$, called the total, which is always equal to the composition of all the obligations belonging to a region. We also require all obligations to be self-incompatible: for every obligation $\mathtt{O}$, $\mathtt{O} \bullet \mathtt{O}$ is undefined. The obligation algebra we define for $\mathbf{d}$ is extremely simple: it consists of just two obligations, $\mathtt{R}$ (the total) and $\mathbf{0}$ (the identity). To perform an atomic update, such as the acquisition of the lock, on $\mathbf{d}_r(x, a)$, one can assume the interpretation holds, evaluate the effects of the update, and then check that the interpretation holds for some (potentially new) abstract state. When the lock is to be acquired, the state of the lock is 0 and thus the obligation $\mathtt{R}$ is held by the interpretation of the region. After the update, $l = 1$ so to check that the interpretation is re-established, one is forced to leave $\mathtt{R}$ in the local state obtaining $\mathbf{d}_r(x, a) * \lceil \mathtt{R} \rceil_r$ as a local assertion. The overall effect is that by acquiring the lock, we get hold of an obligation in the local state. The only way to get rid of it, is by performing another update to the region that will imply the obligation will be in the interpretation. In this case, unlocking the lock will set $l = 0$ and to re-establish the interpretation we need to "consume" $\lceil \mathtt{R} \rceil_r$.

Intuitively, what the interpretation asserts are facts that hold for the shared region; any obligation that is held locally cannot not be in the current interpretation of the region, and vice versa. TaDA Live exploits this mechanism to encode liveness information in obligations: the implicit assumption associated with them is that any thread holding an obligation locally will eventually put it back in the region.

We can now describe the TaDA Live reasoning principle for proving that `op` terminates. The specification of lock (to guarantee that `lock` does not block) requires us to prove that, in any future, the lock is eventually unlocked. At the invocation of `lock` the current thread does not hold the obligation $\mathtt{R}$: it is either in the region if $l = 0$ or in some other thread's local state if $l = 1$. We can then use the liveness assumption on obligations and conclude that if $l = 1$, then $\mathtt{R}$ is outside the region, so there will be a point in the future where the environment will put it back in the region. The only way to put $\mathtt{R}$ in the region is by setting $l = 0$, thus proving that eventually

the lock will be unlocked. After the `lock` operation succeeded, we find we hold $\mathtt{R}$ in our local state. To be able to establish the postcondition $\mathsf{D}(\mathtt{x}, f(a))$, which states that we hold no obligations, we will need to update the region again and put $\mathtt{R}$ back in the interpretation. This is realised by the call to `unlock` (the specification of which guarantees termination without the need to rely on liveness assumptions from the environment).

The idea of using ghost state to encode liveness assumptions is very powerful. First, it allows to define arbitrary "good states" that need to be always eventually reached. Second, because ghost state is linked uniformly to changes to the shared memory, what is considered "good" or "bad" is kept consistent across operations. Indeed if an operation `op1` needs a liveness assumption, there will be an obligation in the region associated with that assumption; any operation `op2` performing an update which may cause `op1` to block, will end up holding some obligation as a result, and will thus be aware that work needs to be done to fulfil the associated liveness assumption. We say an obligation is *live* if it can be assumed to be always returned to the region interpretation at some point in the future by the environment.

Note that the only way one can obtain an obligation locally is by performing some (non-ghost) state change: the thread performing such change will be aware of the responsibility of placing the obligation back in the region in some finite time. On the other hand, an obligation we hold locally may be fulfilled by another thread executing the necessary state change on our behalf. As a result of this interference we would see that our obligation may not be held locally any more if the state change happened.

Now suppose the data structure D uses two locks, one at `x` and one at `x+1`:

```
op1(x) { x1 := [x]; x2 := [x+1];
  lock(x1); lock(x2);
  // modify data
  unlock(x2); unlock(x1);
}
```

```
op2(x) { x1 := [x]; x2 := [x+1];
  lock(x2); lock(x1);
  // modify data
  unlock(x2); unlock(x1);
}
```

This code is the classic example of a potential deadlock: a concurrent execution of `op1` and `op2` could deadlock, with both calls never returning to their client. Let us see how TaDA Live detects the potential deadlock.

In the verification of `op1`, we need to show that the first call to `lock` terminates. We do this by using exactly the same argument as before, therefore introducing an obligation $\mathtt{R}_1$ representing a promise to unlock `x1`. Similarly, we introduce an obligation $\mathtt{R}_2$ for the lock at `x2`. The two obligations compose to the total: $\mathtt{R}_1 \bullet \mathtt{R}_2 = \mathbf{1}$. The region interpretation is adapted straightforwardly:

$$I(\mathbf{d}_r(x,a)) \triangleq x \mapsto x_1, x_2 \ldots * \exists l_1, l_2.\, \big(\mathsf{L}(x_1,l_1) * (\lceil \mathtt{R}_1 \rceil_r \iff l_1 = 0) * \mathsf{L}(x_2,l_2) * (\lceil \mathtt{R}_2 \rceil_r \iff l_2 = 0)\big) * \cdots \tag{4}$$

Now, exactly as the single lock case, we can justify why `lock(x1)` in `op1` terminates by invoking the liveness assumptions on $\mathtt{R}_1$.

The situation is different for `lock(x2)` however: we want to invoke liveness of $\mathtt{R}_2$ *while holding* $\mathtt{R}_1$ *locally*. It is not possible to do so soundly (i.e. not causing circularity in the argument) without knowing whether the environment may need $\mathtt{R}_2$ to be live, to maintain $\mathtt{R}_1$ live. To avoid circular reasoning and break the symmetry between obligations, we equip our obligation algebras with a *layer structure*: an arbitrary well-founded partial order $\mathcal{L}$ (with minimal element $\bot$) and a map lay(_) associating to each obligation a layer in $\mathcal{L}$. The (acyclic) strict order of $\mathcal{L}$ represents the possible dependencies between obligations. In our example, we set $\mathcal{L}_{\mathbf{d}} = \bot < k_2 < k_1 < \top$ and assign $\mathrm{lay}(\mathtt{R}_i) = k_i$ for $i = 1, 2$. Now we can refine the liveness assumption associated with obligations: *The environment of a thread $t$ is allowed to continuously hold an obligation of layer $k$ forever, only if the thread $t$ continuously holds an obligation of layer strictly lower than $k$ forever.* We can now, with the revised condition, prove that `lock(x2)` terminates in `op1`: during such call we hold $\mathtt{R}_1$ locally, which forbids us from assuming liveness of any obligation with layer greater or equal than $k_1$. We are however allowed to assume liveness of

obligations of lower layers, i.e. $k_2$. By using this liveness assumption we can prove that the lock at `x2` will always be eventually unlocked, satisfying the liveness condition of the specification of `lock`.

Now, consider `op2` locking the locks in the opposite order. By symmetry, the proof would require $\mathtt{R}_1$ to be live while $\mathtt{R}_2$ is continuously held locally. This is not possible because $\mathtt{R}_1$ is of higher layer than $\mathtt{R}_2$. Note that an operation `op3(x)` locking an unlocking `x1` and `x2` in a non-overlapping fashion, can be easily verified along side `op1` using the same obligation algebra and layer structure.

Reasoning with layers is quite intuitive. For our lock example, the proof outline follows precisely the developer's intuition and the layers of the obligations specify the intended synchronisation protocol very concisely. Note how there is nothing ad-hoc about obligations nor layers: they arise naturally as proof obligations in our system, and are always linked to non-ghost state changes. No a priori knowledge of blocking constructs/operations is encoded in them. We will see a general use of layers when considering the verification of CLH lock later.

***Using the liveness assumptions*** We now turn our attention to how TaDA Live can prove an implementation of a specification with non-trivial pseudo-quantifier liveness assumption, correct. We will illustrate the principle on spin lock. How can we show the spin lock implementation complies with its TaDA Live specification? The hard part of the reasoning is of course in the while loop. Once the loop is entered, there is no guarantee of termination unless we make full use of the liveness assumptions stated in the specification. The abstract state of the spin lock is a pair of the lock state $l \in \{0, 1\}$, and the impedance budget $\alpha$, an ordinal. Intuitively, we should reason about the loop by assuming the environment is composed of concurrent operation calls from the same module. To obtain a proof that is local to the current operation, we model interference as a *rely* relation, over-approximating the effects of the concurrent code on the abstract state. The rely in this case allows arbitrary changes to be made to $l$, but, because we need to assume bounded impedance, the transitions from $l = 0$ to $l = 1$ will consume some non-trivial amount of the budget $\alpha$. The loop invariant is asserting that there is a lock at `x`, in some abstract state $l, \alpha$, and either `b = 0`, in which case we have not performed the linearization point yet, or `b = 1`, in which case we have performed the linearization point from $0, \alpha$ to $1, \phi(\alpha)$. It is crucial that by bringing $l$ from 0 to 1, we show we consume some budget (through $\phi(\alpha) < \alpha$) because our effects on the resource (i.e. the *guarantee*) need to be compatible with our rely. Compatibility of the code with the rely is readily seen to be true: no change to the shared state is made while the **`CAS`** on line 4 is unsuccessful; line 4 can be only executed once, when the lock is unlocked: in that case we can strictly decrease $\alpha$by updating it to $\phi(\alpha)$, thus respecting the rely constraint on the budget use.

Now let us consider the while loop, given the above invariant. Certainly if it terminates we have the desired input/output behaviour. We need to argue why it does terminate. Traditionally this is done by defining a well-founded *variant* (i.e. a function from the state to an ordinal) which strictly decreases at each loop iteration. From well-foundedness we get that there are only finitely many decreases possible and the loop must terminate. In our case, this condition is too strong: the environment can take arbitrarily long to unlock the lock, and many iterations of the while loop will see no visible progress towards termination. We therefore refine the condition to support this blocking behaviour. We still define a variant which cannot be increased during iterations; however, we do not require a strict decrease at each iteration, but only in iterations from "unblocked" states. More precisely, we can define a *target condition* $T$ which represents states from which the loop can guarantee progress (i.e. a strict decrease in the variant). To ensure that the assumptions support the termination argument, one has to show that they imply that the target $T$ will be eventually reached in a stable manner. We call this the *environment liveness condition*.

In the spin lock example, $T$ is simply the states where $l = 0$: in such state, either line 4 succeeds, in which case we strictly decrease the budget $\alpha$, or we will be pre-empted by a lock acquisition by the environment, which also causes the budget to strictly decrease. Our variant can simply be (any upper-bound on) the budget $\alpha$.

For proving the environment liveness condition for $T$ we require a second variant, this time bounding the time the operation can wait for $T$ in the worst case. We call this variant, the *(environment) progress measure*. The progress measure is required to never increase, and strictly decrease as the result of performing any of the environment state changes, that can be assumed will eventually happen. From the spin lock specification, we know we can assume that from a state violating $T$ (i.e. $l = 1$) the environment will eventually make a transition to $T$. We define the progress measure for the spin lock loop to be $2\alpha + l$, which never increases (as ensured by the rely), and is strictly decreased when the environment fulfils its promise to bring $l$ from 1 to 0.

Now the termination argument is complete: at any point in time, $l$ can be 1 or 0; if the **CAS** on line 4 is scheduled the loop variant decreases from 1 to 0 and the loop will terminate. Otherwise the environment can pre-empt the operation and bring $l$ to 1; as a result $\alpha$ must strictly decrease, so we know this step can happen only until the budget is exhausted. From $l = 1$ the environment is assumed to bring $l = 0$ in the future. There can be arbitrary but finite delay until that happens. When that happens the progress measure strictly decreases. In the *worst case*, the progress measure reaches the value 0 in finite time, at which point $\alpha = 0$ thus forbidding any concurrent lock acquisition. The lock will stay stable in the unlocked state and, because of fairness of the scheduler, line 4 will eventually be scheduled and succeed.

***The layer system*** Since the spin lock does not present internal blocking, we do not need obligations. In general however, liveness assumptions about obligations and the pseudo-quantifier may need to be used in conjunction. For example, consider a module implementing an integer counter which (for simplicity) can *only be decremented* with a `dec` operation, from some initial value. We equip the counter with an operation that blocks until the counter is 0 or lower:

```
waitIfPos(x) { v := [x]; x1 := [x+1]; x2 := [x+2];
  lock(x1);
  while(v > 0) { lock(x2); v := [x]; unlock(x2); }
  unlock(x1); }
```

Here, for presentation purposes, we store an integer at `x` and two fair locks, at `x+1` and `x+2`. The `waitIfPos` operation presents both internal blocking (on the locks) and external blocking (on the value of the counter), giving rise to the specification:

$$\vdash \forall n \in \mathbb{Z} \twoheadrightarrow \mathbb{Z}^{\leq 0}.\langle \mathrm{C}(\mathtt{x}, n)\rangle\ \mathtt{waitIfPos(x)}\ \langle \mathrm{C}(\mathtt{x}, n) \wedge n \leq 0\rangle$$

The region **cnt** implementing the abstract predicate C will be analogous to Eq. (4) and we use $\mathtt{R}_1$ and $\mathtt{R}_2$ for the obligations associated with `x1` and `x2`. The layer structure $\mathcal{L}_{\mathbf{cnt}}$ would also be the same as the `op1` example, as the locks are acquired in a nested fashion.

Termination of the `lock(x1)` call follows directly from liveness of $\mathtt{R}_1$ as before. The termination argument for the while loop is however more involved: the loop body terminates by virtue of liveness of the obligation $\mathtt{R}_2$ which can be assumed thanks to the layers being in the desired order. The loop overall needs to use the pseudo-quantifier liveness assumption that guarantees that the counter will be eventually brought to a non-positive value. Since the value can only be decremented, once non-positive the value will remain non-positive. Without further constraints however we may allow two different implementations for `dec`:

```
dec1(x) { y := [x+1]; lock(y);v := [x];[x] := v-1; unlock(y) }
dec2(x) { y := [x+2]; lock(y);v := [x];[x] := v-1; unlock(y) }
```

Of the two, however, only `dec2` can be used to help `waitIfPos` to terminate! If only `dec1` is used by the client in the environment of `waitIfPos`, with the intention of eventually unblocking it, the client would deadlock. TaDA Live solves this problem by realising that the pseudo-quantifier assumption is nothing but a special form of obligation, an "external" obligation, specific to a single operation instead of being specified in the region, and never held by the operation itself. So, to harmonise it with the other internal obligations we assign to it a layer, by annotating

the pseudo quantifier: $\forall n \in \mathbb{Z} \twoheadrightarrow_{k_2} \mathbb{Z}^{\leq 0}$ represents the fact that the pseudo-quantifier cannot be considered "live" while we are holding locally obligations of layer lower or equal than $k_2$. To distinguish between `dec1` and `dec1`, we expose in the specifications, a (strict) upper-bound on the layers that are assumed live in the proof of the operation: we note such upper-bound before the turnstile symbol. All in all we obtain the specifications:

$$\top \vdash \forall n \in \mathbb{Z} \twoheadrightarrow_{k_2} \mathbb{Z}^{\leq 0}.\langle \mathrm{C}(\mathrm{x}, n)\rangle \ \texttt{waitIfPos(x)}\ \langle \mathrm{C}(\mathrm{x}, n) \wedge n \leq 0\rangle$$
$$\top \vdash \forall n \in \mathbb{Z}.\langle \mathrm{C}(\mathrm{x}, n)\rangle \ \texttt{dec1(x)}\ \langle \mathrm{C}(\mathrm{x}, n-1)\rangle$$
$$k_1 \vdash \forall n \in \mathbb{Z}.\langle \mathrm{C}(\mathrm{x}, n)\rangle \ \texttt{dec2(x)}\ \langle \mathrm{C}(\mathrm{x}, n-1)\rangle$$

The crucial difference between `dec1` and `dec2` is now expressed by the layer before the turnstile: `dec1` needs the liveness of $\mathrm{R}_1$ and therefore exposes the upper-bound $\top$ in its specification. We call *external*, the layers mentioned in the specifications. In our counter example, $\bot$ is the only non-external layer.

Let us conclude this example by briefly sketching how a client can use these specifications, and how the layers exposed in the specifications will be taken into account in the client's reasoning. Let us consider a client of a counter at `c`, with `done` initially false:

```
waitIfPos(c);          ||   while(!d){ // d false initially
[done] := true;        ||     dec2(c); d := [done]; }
```

Since TaDA Live proofs are thread-local, we need to represent the possible interference the two threads can have on each other through a shared region **client** with associated interpretation and interference relation. We give a simplified interpretation highlighting the interesting parts:

$$I(\mathbf{client}_r(c, n, done, d)) \triangleq c \mapsto n * done \mapsto d * (d \iff \lceil \mathrm{D} \rceil_r) * \lceil \mathrm{A}(\max(n, 0)) \rceil_r \tag{5}$$

with obligation algebra defined by $\mathbf{1} = \mathrm{A}(0) \bullet \mathrm{D}$, $\mathrm{lay}(\mathrm{A}(n)) = 0$, $\mathrm{lay}(\mathrm{Dec}(n)) = 1$, $\mathrm{A}(n) = \mathrm{Dec}(n) \bullet \mathrm{A}(n+1)$, $\mathrm{lay}(\mathrm{D}) = 2$, $\mathcal{L}_{\mathbf{client}} = \bot < 0 < 1 < 2$. The idea is that when the counter is initially created with some value $k > 0$ we will hold locally $k$ obligations $\lceil \mathrm{Dec}(0) \bullet \cdots \bullet \mathrm{Dec}(k-1) \rceil_r$ and when we set `done` to false we obtain locally $\lceil \mathrm{D} \rceil_r$. The Par rule allows us to split the state in two preconditions $P_1 * P_2$, one for each thread. We can let $P_1$ contain the D obligation and $P_2$ the Dec obligations.

Now, note that the layers of the client and the layers of the counter specifications are considered disjoint. When exposed to the client, the specifications of the counter look as following, for every strictly monotone partial layer function $\mu \colon \mathcal{L}_{\mathbf{cnt}} \rightharpoonup \mathcal{L}_{\mathbf{client}}$:

$$\mu(\top) \vdash \forall n \in \mathbb{Z} \twoheadrightarrow_{\mu(k_2)} \mathbb{Z}^{\leq 0}.\langle \mathrm{C}_\mu(\mathrm{x}, n)\rangle \ \texttt{waitIfPos(x)}\ \langle \mathrm{C}_\mu(\mathrm{x}, n) \wedge n \leq 0\rangle$$
$$\mu(\top) \vdash \forall n \in \mathbb{Z}.\langle \mathrm{C}_\mu(\mathrm{x}, n)\rangle \ \texttt{dec1(x)}\ \langle \mathrm{C}_\mu(\mathrm{x}, n-1)\rangle$$
$$\mu(k_1) \vdash \forall n \in \mathbb{Z}.\langle \mathrm{C}_\mu(\mathrm{x}, n)\rangle \ \texttt{dec1(x)}\ \langle \mathrm{C}_\mu(\mathrm{x}, n-1)\rangle$$

This allows the client to relate the externally visible layers of the counter module, with the layers needed in that specific client. Furthermore, $\mu$ is restricted to only be defined on external layers of the counter, thus allowing the counter to be reimplemented using a different internal layer structure, provided the relative ordering between the external layers is preserved. To keep this layer remapping consistent, a *tagged* version of the abstract predicate $\mathrm{C}_\mu$ is used in the specifications: it can be thought as a fresh name for that abstract predicate. This allows the client to create multiple instances of the same module, remapping the layers of which to different layers, and never confuse the layers of the two.

In our example we instantiate the specifications of the counter using $\mu$ with $\mu(\top) = 2$, $\mu(k_1) = 1$, and $\mu(k_2) = 0$. When using the specification of `waitIfPos(c)`, the client has the obligation to show that it only holds obligations of layer $\mu(\top) = 2$ or higher, which it does because D has layer 2. Additionally, it needs to show that, thanks to liveness of obligations of layer strictly lower than 2, the counter value will eventually be brought to zero. This is true because the Dec obligations (of layer 1) are not held locally, and they can only be put back in the region by being "absorbed" into A; thus at any point where the counter is strictly positive, we can be sure a Dec obligation

is outside the region; the only transitions that make it go back to the region are decrements of the counters. Each such decrement will bring us closer to our liveness target $\mathbb{Z}^{\leq 0}$, so we can define the environment progress variant to be simply $\max(n, 0)$.

For the thread on the right, the termination depends on the obligation D being eventually returned to the region, but this is only needed once the value of the counter is 0 or lower: before then, we are making progress. The call to `dec2` can be inferred to terminate because it promises to do so if only obligations of layer $\mu(k_1) = 1$ or higher are held locally, which is true because we only locally hold Dec obligations which have layer 1. If the client used `dec2` instead, there would be a contradiction: we would only be able to hold obligations of level $\mu(\top) = 2$ or higher. Once the counter is 0 or lower, all the Dec obligations are inside the region and we are free to assume D live, which, when eventually returned to the region, will bring `done` to be false and stably so, allowing us to terminate.

The parallel rule has an important side-condition: the post-conditions of the two threads cannot contain obligations of layers that the other thread is assuming live. That is: all the promises of a thread to the other need to be fulfilled before joining the parallel.

## B Rules imported from TaDA

These rules are imported from TaDA and do not manipulate in any way the liveness assumptions. Their main purpose is to offer a way to prove abstract atomicity. Their soundness is a straightforward adaptation of the proofs of soundness of the original TaDA rules.

The Make atomic rule takes a proof of an Hoare triple containing a witness that the linearization point has been executed (once) expressed as the assertions $r \Longmapsto \blacklozenge$ and $r \Longmapsto (x, y)$, and turns it into an atomic triple.

Make atomic rule

$$\frac{r \notin \mathcal{A} \qquad \mathcal{A}' = (r, x, X \twoheadrightarrow_k X', Y(x)), \mathcal{A} \qquad \{\,(x,y) \mid x \in X, y \in Y(x)\,\} \subseteq \mathcal{T}_{\mathfrak{t}}(\mathrm{G})^* \qquad m; \lambda; \mathcal{A}' \vdash \left\{P_{\mathrm{h}} * \exists x \in X.\, \mathbf{t}_r^{\lambda}(x) * r \Longmapsto \blacklozenge\right\} \mathbb{C} \left\{\exists x \in X, y \in Y(x), z.\, \mathbf{t}_r^{\lambda}(z) * Q_{\mathrm{h}}(x,y) * r \Longmapsto (x,y)\right\}}{m; \lambda; \mathcal{A} \vdash \forall x \in X \twoheadrightarrow_k X'. \left\langle P_{\mathrm{h}} \;\middle|\; \mathbf{t}_r^{\lambda}(x) * \lfloor \mathrm{G} \rfloor_r \right\rangle \mathbb{C} \exists y. \left\langle Q_{\mathrm{h}}(x,y) \;\middle|\; y \in Y(x) \wedge \mathbf{t}_r^{\lambda}(y) * \lfloor \mathrm{G} \rfloor_r \right\rangle} \textsc{MkAtom}$$

To perform an update to a region one can apply the Update region rule, which gives access to the interpretation of the region to the command, which needs to update the region atomically and then re-establish the region interpretation for some new abstract state $y$. The rule updates the atomicity tracking assertion $r \Longmapsto \_$ according to the performed update.

Update region rule

$$\frac{\mathcal{A}' = (r, x, X \twoheadrightarrow_k X', Z(x)), \mathcal{A} \qquad m; \lambda; \mathcal{A} \vdash \forall x \in X \twoheadrightarrow_k X'. \left\langle P_{\mathrm{h}} \;\middle|\; I(\mathbf{t}_r^{\lambda}(x)) * P_{\mathrm{a}}(x) \right\rangle \mathbb{C} \exists y. \left\langle Q_{\mathrm{h}}(x,y) \;\middle|\; \begin{array}{r} \exists z \in Z(x).\, I(\mathbf{t}_r^{\lambda}(z)) * Q_1(x,y) \\ \vee\, I(\mathbf{t}_r^{\lambda}(x)) * Q_2(x,y) \end{array} \right\rangle}{\begin{array}{l} m; \lambda{+}1; \mathcal{A}' \vdash \forall x \in X \twoheadrightarrow_k X'. \\ \qquad \left\langle P_{\mathrm{h}} \;\middle|\; \mathbf{t}_r^{\lambda}(x) * P_{\mathrm{a}}(x) * r \Longmapsto \blacklozenge \right\rangle \\ \qquad \mathbb{C} \\ \qquad \exists y. \left\langle Q_{\mathrm{h}}(x,y) \;\middle|\; \begin{array}{r} \exists z \in Z(x).\, \mathbf{t}_r^{\lambda}(z) * Q_1(x,y) * r \Longmapsto (x,z) \\ \vee\, \mathbf{t}_r^{\lambda}(x) * Q_2(x,y) * r \Longmapsto \blacklozenge \end{array} \right\rangle \end{array}} \textsc{UpdReg}$$

The open region rule can be used when the update is an abstract skip and not a linearization point.

Open region rule

$$\frac{m; \lambda; \mathcal{A} \vdash \forall x \in X \twoheadrightarrow_k X'. \left\langle P_{\mathrm{h}} \;\middle|\; I(\mathbf{t}_r^{\lambda}(x)) * P_{\mathrm{a}}(x) \right\rangle \mathbb{C} \exists y. \left\langle Q_{\mathrm{h}}(x,y) \;\middle|\; I(\mathbf{t}_r^{\lambda}(x)) * Q_{\mathrm{a}}(x,y) \right\rangle}{m; \lambda{+}1; \mathcal{A} \vdash \forall x \in X \twoheadrightarrow_k X'. \left\langle P_{\mathrm{h}} \;\middle|\; \mathbf{t}_r^{\lambda}(x) * P_{\mathrm{a}}(x) \right\rangle \mathbb{C} \exists y. \left\langle Q_{\mathrm{h}}(x,y) \;\middle|\; \mathbf{t}_r^{\lambda}(x) * Q_{\mathrm{a}}(x,y) \right\rangle} \textsc{OpReg}$$

An important rule is the Atomicity weakening rule which allows to weaken an atomic triple to a Hoare triple (when the atomic pre/post-conditions in the consequence are emp) provided the atomic components that are moved to the Hoare pre/post-conditions are stable. In the case of the precondition, this simply follows from the fact that $P$ does not depend on $x$.

Atomicity weakening rule

$$\frac{\forall x \in X, y.\ \lambda; \mathcal{A} \vDash Q(x,y) \text{ stable} \qquad m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \,|\, P * P_\mathsf{a}(x)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \,|\, Q(x,y) * Q_\mathsf{a}(x,y)\rangle}{m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} * P \,|\, P_\mathsf{a}(x)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) * Q(x,y) \,|\, Q_\mathsf{a}(x,y)\rangle}\ \textsc{AtomW}$$

Restrict atomic rule

$$\frac{\begin{array}{c} r \notin \mathcal{A} \qquad \forall x \in X.\ (x, f(x)) \in \mathcal{T}_\mathsf{t}(\mathrm{G})^* \\ m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \mid I(\mathsf{t}_r^\lambda(x)) * P_\mathsf{a}(x) * \lfloor \mathrm{G} \rfloor_r \rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \mid I(\mathsf{t}_r^\lambda(f(x))) * Q_\mathsf{a}(x,y)\rangle \end{array}}{m; \lambda{+}1; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \mid \mathsf{t}_r^\lambda(x) * P_\mathsf{a}(x) * \lfloor \mathrm{G} \rfloor_r \rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \mid \mathsf{t}_r^\lambda(f(x)) * Q_\mathsf{a}(x,y)\rangle}\ \textsc{RA}$$

Substitution rule

$$\frac{m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in f(X) \twoheadrightarrow_k f(X').\langle P_\mathsf{h} \,|\, P_\mathsf{a}(x)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \,|\, Q_\mathsf{a}(x,y)\rangle \qquad f : X \to Z}{m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \,|\, P_\mathsf{a}(f(x))\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(f(x),y) \,|\, Q_\mathsf{a}(f(x),y)\rangle}\ \textsc{Sub}$$

Consequence rule

$$\frac{\begin{array}{c} \lambda; \mathcal{A}; \vDash_\Delta P_\mathsf{h} \Rrightarrow P'_\mathsf{h} \qquad \forall x \in X.\ \lambda; \mathcal{A}; \vDash_\Delta P_\mathsf{a}(x) \Rrightarrow P'_\mathsf{a}(x) \\ m; \lambda; \mathcal{A} \vdash_\Delta \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P'_\mathsf{h} \,|\, P'_\mathsf{a}(x)\rangle \,\mathbb{C}\, \exists y.\langle Q'_\mathsf{h}(x,y) \,|\, Q'_\mathsf{a}(x,y)\rangle \\ \forall x \in X, y.\ \lambda; \mathcal{A}; \vDash_\Delta Q'_\mathsf{h}(x,y) \Rrightarrow Q_\mathsf{h}(x,y) \qquad \forall x \in X, y.\ \lambda; \mathcal{A}; \vDash_\Delta Q'_\mathsf{a}(x,y) \Rrightarrow Q_\mathsf{a}(x,y) \end{array}}{m; \lambda; \mathcal{A} \vdash_\Delta \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \,|\, P_\mathsf{a}(x)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \,|\, Q_\mathsf{a}(x,y)\rangle}\ \textsc{Cons}$$

Atomic $\exists$ Elimination rule

$$\frac{m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X', z \in Z.\langle P_\mathsf{h} \,|\, P_\mathsf{a}(x,z)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \,|\, Q_\mathsf{a}(x,y,z)\rangle}{m; \lambda; \mathcal{A} \vdash \mathbb{W} x \in X \twoheadrightarrow_k X'.\langle P_\mathsf{h} \,|\, \exists z \in Z.\, P_\mathsf{a}(x,z)\rangle \,\mathbb{C}\, \exists y.\langle Q_\mathsf{h}(x,y) \,|\, \exists z \in Z.\, Q_\mathsf{a}(x,y,z)\rangle}\ \textsc{A}\exists\textsc{Elim}$$

$\exists$ Elimination rule

$$\frac{m; \lambda; \mathcal{A} \vdash \{P\} \,\mathbb{C}\, \{Q\}}{m; \lambda; \mathcal{A} \vdash \{\exists x.\, P\} \,\mathbb{C}\, \{\exists x.\, Q\}}\ \exists\textsc{Elim}$$

The rules for the primitive commands are exactly the same as TaDA's, we write the one we add for the **FAS** primitive for illustration:

Fetch-and-set rule

$$\frac{}{m; \lambda; \mathcal{A} \vdash_\Delta \mathbb{W} v \in V.\langle \mathbb{E}_1 \mapsto v\rangle\ \mathsf{r} := \mathbf{FAS}(\mathbb{E}_1, \mathbb{E}_2)\ \langle \mathbb{E}_1 \mapsto \mathbb{E}_2 \wedge \mathsf{r} = v\rangle}\ \text{FAS}$$

## C The modularity rules

Our proofs attain modularity by means of abstraction, following the CAP approach. The judgement for a module $\vdash \mathbb{M} : (\Sigma, \mathrm{ASp}, \mathrm{Ax})$ declares the implementation consistent with the specification visible to the client: a set of "exported" abstract predicate names $\Sigma$, a set of function specifications (in the form of atomic triples) ASp, and a set of axioms Ax of the form $\forall x.P(x) \Rightarrow Q(x)$ (these can be useful for expressive abstract predicate abstractions, see CAP for examples). The Module rule can be used to prove the judgement:

Module rule

$$\frac{\forall i \in I.\, \vdash \mathbb{M}_i : (\Sigma_i, \mathsf{ASp}_i, \mathsf{Ax}_i) \qquad \Sigma^{\mathsf{im}} = \cup_{i\in I}\Sigma_i \qquad (\mathsf{ASp}_i, \mathsf{Ax}_i)_{i\in I}, \mathsf{ADefs}(\Sigma, \Sigma^{\mathsf{im}}) \vdash \mathbb{D} : \mathsf{ASp} \qquad \forall A \in \mathsf{Ax}.\ (\mathsf{Ax}_i)_{i\in I}, \mathsf{ADefs}(\Sigma, \Sigma^{\mathsf{im}}) \vDash A}{\vdash \textbf{import } (\mathbb{M}_i)_{i\in I} \textbf{ in } \mathbb{D} : (\Sigma, \mathsf{ASp}, \mathsf{Ax})^{\mathcal{L}(\mathsf{ASp})}}\ \textsc{Mod}$$

To apply the rule, one has to give a set of abstract predicate and region definitions $\mathsf{ADefs}(\Sigma, \Sigma^{\mathsf{im}})$ where $\Sigma$ are the abstract predicate names with definitions in the set, and these definitions can make use of the abstract predicates imported from the imported modules $\Sigma^{\mathsf{im}}$. Then, with knowledge of the imported modules specifications and of the internal regions and abstract predicates definitions, each function defined by the module is proven correct with respect to its specification. The last premise simply requires a semantic check that proves the exported axioms are implied by the imported and internal definitions.

The exported specifications are closed, in the consequence of the rule, with respect to transformations from the exported layers $\mathcal{L}(\mathsf{ASp})$, i.e. the internal layers that are mentioned in any of the function specifications. This closure is defined as follows. Given a layer structure $\mathcal{L}$, the set of layer transformations $\mathsf{LTr}^{\mathcal{L}}$ is the set of strictly monotonic maps $\mu\colon \mathcal{L} \to \mathcal{L}'$, for some layer structure $\mathcal{L}'$. Strictly monotonic means that for all $m_1, m_2 \in \mathcal{L}$ if $m_1 < m_2$ then $\mu(m_1) < \mu(m_2)$.

Given a specification $(\Sigma, \mathsf{ASp}, \mathsf{Ax})$, let $\mathcal{L}$ be the set of layers occurring in any abstract specification in ASp (a sub-lattice of the layer structure in $\mathsf{ADefs}(\Sigma, \Sigma^{\mathsf{im}})$). Then $(\Sigma, \mathsf{ASp}, \mathsf{Ax})^{\mathcal{L}}$ is the specification $(\Sigma', \mathsf{ASp}', \mathsf{Ax}')$ where

$$\Sigma' \triangleq \{\mathsf{A}_\mu \mid \mathsf{A} \in \Sigma, \mu \in \mathsf{LTr}^{\mathcal{L}}\} \qquad \mathsf{ASp}' \triangleq \{S\theta_\mu \mid S \in \mathsf{ASp}, \mu \in \mathsf{LTr}^{\mathcal{L}}\}$$

$$\theta_\mu\colon \Sigma \to \Sigma' \quad \theta_\mu(\mathsf{A}) \triangleq \mathsf{A}_\mu \qquad \mathsf{Ax}' \triangleq \{A\theta_\mu \mid A \in \mathsf{Ax}, \mu \in \mathsf{LTr}^{\mathcal{L}}\}$$

The intuitive meaning of this construction is: if a module was proven correct with respect to some specification, the only client-relevant information about the layers occurring in the specification is their relative ordering; in other words, the specifications do not leak the internal layer structure. We express this idea by allowing arbitrary transformations of these layers into any client-specific layer structure $\mathcal{L}'$ as long as the relative orderings are preserved. The substitution $\theta_\mu$ is a syntactic renaming of the abstract predicates recording the choice of layer map in the name. The abstract predicate renaming is necessary to ensure that the client must fix the choice of the layer map per instance of the module. In fact, once the choice of $\mu$ is made when creating an instance of the module, only the abstract specifications and axioms tagged with the same map $\mu$ can be put to use for that instance.

The implementation rule defines how to prove the intermediate judgement $\Delta \vdash \mathbb{D} : \mathsf{ASp}$: for each function $\mathtt{f}$ with a specification in ASp we fetch the corresponding implementation from $\mathbb{D}$ and prove it satisfies the specification.

Implementation rule

$$\frac{\text{for each } (m \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \,|\, P_{\mathsf{a}}(x)\rangle\ \mathtt{f}(\vec{\mathtt{z}})\ \exists y \in Y.\langle Q_{\mathsf{h}}(x,y) \,|\, Q_{\mathsf{a}}(x,y)\rangle) \in \mathsf{ASp} \qquad \mathtt{f}(\vec{\mathtt{x}})\{\mathbb{C}\} \in \mathbb{D} \qquad m, \lambda, \epsilon \vdash_\Delta \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \,|\, P_{\mathsf{a}}(x)\rangle\ \mathbb{C}\ \exists y \in Y.\langle Q_{\mathsf{h}}(x,y) \,|\, Q_{\mathsf{a}}(x,y)\rangle}{\Delta \vdash \mathbb{D} : \mathsf{ASp}}\ \textsc{Imp}$$

The atomicity context can be set to be empty ($\epsilon$) as the function will not be able to see any higher level assumption other than the ones stated in the specification.

The function call rule simply requires the called function (which will belong to some imported module) to have a specification which supports the pre/post-conditions given the substitution of the formal parameters.

Function Call rule

$$\frac{m \vdash \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}}(\vec{\mathtt{z}}) \,|\, P_{\mathsf{a}}(\vec{\mathtt{z}}, x)\rangle\ \mathtt{f}(\vec{\mathtt{z}})\ \exists y.\langle Q_{\mathsf{h}}(\vec{\mathtt{z}}, x, y, \mathsf{ret}) \,|\, Q_{\mathsf{a}}(\vec{\mathtt{z}}, x, y)\rangle \in \Delta \qquad \vec{\mathtt{z}}, \mathtt{r} \notin \mathsf{fv}(\vec{\mathbb{E}})}{m; \lambda; \mathcal{A} \vdash_\Delta \mathbb{W}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}}(\vec{\mathbb{E}}) \,|\, P_{\mathsf{a}}(\vec{\mathbb{E}}, x)\rangle\ \mathtt{r} := \mathtt{f}(\vec{\mathbb{E}})\ \exists y.\langle Q_{\mathsf{h}}(\vec{\mathbb{E}}, x, y, \mathtt{r}) \,|\, Q_{\mathsf{a}}(\vec{\mathbb{E}}, x, y)\rangle}\ \textsc{Fun}$$

# D Primitive Atomic blocks

It is often useful in formal proofs to model a primitive not included in the formalised programming language or some ghost code, as an *atomic block* $\langle \mathbb{C} \rangle$. The semantics of an atomic block is to execute $\mathbb{C}$ sequentially and atomically, i.e. in a single step.

To handle atomic blocks in the logics, we would need to formulate a judgment for proving a command run sequentially in isolation would produce some state change, and then lift the state change in an atomic triple specification. Since sequential proofs in separation logics are well-understood, we state the rule semi-formally, by using the notation $\vdash_{\mathsf{SL}} [P]\ \mathbb{C}\ [Q]$ to indicate a judgment proven in standard total separation logics. In particular, $\mathbb{C}$ has to terminate in all cases if run from a state satisfying $P$.

The primitive atomic rule lifts the input/output behaviour of a command to an atomic triple for the primitive block executing that command.

Primitive atomic rule

$$\frac{\forall x \in X. \vdash_{\mathsf{SL}} \left[\lfloor P(x) \rfloor^{\Delta}_{\lambda}\right]\ \mathbb{C}\ \left[\lfloor Q(x) \rfloor^{\Delta}_{\lambda}\right]}{m; \lambda; \mathcal{A} \vdash_{\Delta} \mathbb{V} x \in X. \langle P(x) \rangle\ \langle \mathbb{C} \rangle\ \langle Q(x) \rangle}\ \textsc{PrAt}$$

The reification $\lfloor \cdot \rfloor^{\Delta}_{\lambda}$ translates the assertions in assertions about the heap, removing all the abstractions/ghost state. With this rule, for example, we can prove

$$\vdash \mathbb{V} l \in \{0,1\}, v \in \mathbb{N}.\ \langle \mathtt{x}+2 \mapsto l * \mathtt{x}+1 \mapsto v \rangle\ \langle \texttt{[x+2]} := \texttt{0;}\ \mathtt{v} := \texttt{[x+1];}\ \texttt{[x+1]} := \mathtt{v+1} \rangle\ \langle \mathtt{x}+2 \mapsto 0 * \mathtt{x}+1 \mapsto v+1 * \mathtt{v} = v \rangle$$

# E Examples

## E.1 Spin lock

Recall the code for the spin `lock` operation.

```
1 lock(x){
2   b := 0;
3   while(b = 0){
4     b := CAS(x,0,1);
5   }
6 }
```

```
1 unlock(x) {
2   [x] := 0;
3 }
```

The full specifications for spin lock:

$$\top \vdash \mathbb{V} l \in \{0,1\} \twoheadrightarrow_{\bot} \{0\}, \alpha. \langle \mathsf{L}(s, \mathtt{x}, l, \alpha) \wedge \alpha > \phi(\alpha) \rangle\ \mathtt{lock(x)}\ \langle \mathsf{L}(s, \mathtt{x}, 1, \phi(\alpha)) \wedge l = 0 \rangle$$
$$\bot \vdash \langle \mathsf{L}(s, \mathtt{x}, 1, \alpha) \rangle\ \mathtt{unlock(x)}\ \langle \mathsf{L}(s, \mathtt{x}, 0, \alpha) \rangle$$

We give the following abstract predicate and the region definitions:

$$\mathsf{L}(r, x, l, \alpha) \triangleq \mathbf{slock}_r(x, l, \alpha) \qquad \forall \alpha, \beta < \alpha.\ \mathbf{0} : (0, \alpha) \rightsquigarrow (1, \beta)$$
$$I(\mathbf{slock}_r(x, l, \alpha)) \triangleq x \mapsto l \qquad \forall \alpha.\ \mathbf{0} : (1, \alpha) \rightsquigarrow (0, \alpha)$$

The proof outline for the `lock` implementation can be found in figure 9.

The parameters for the while rule application are

$$L \triangleq \mathsf{emp} \qquad M(\alpha) \triangleq \exists l \in \{0,1\}, \delta. \mathbf{slock}_r(\mathtt{x}, l, \delta) \wedge \alpha = 2\delta + l \qquad T \triangleq \exists \delta. \mathbf{slock}_r(\mathtt{x}, 0, \delta)$$

The proof outline for the `unlock` implementation can be found in figure 10

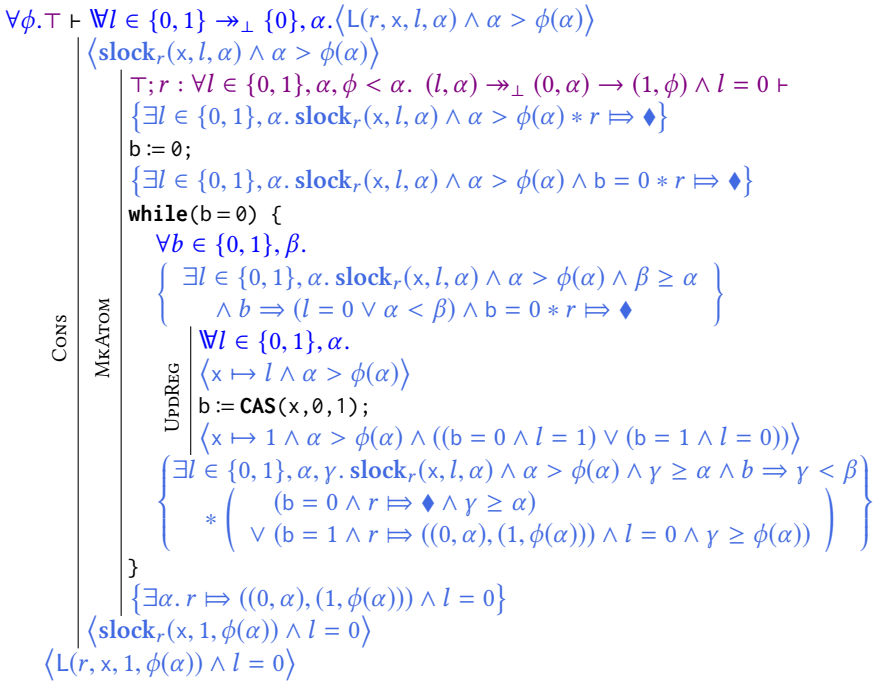



**Figure 9.** Spin lock `lock` operation proof outline.

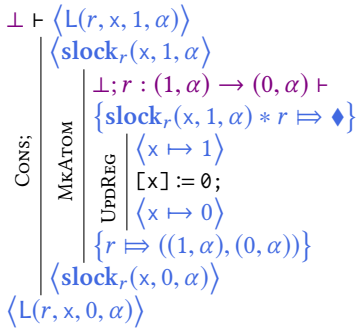



**Figure 10.** Spin lock `unlock` operation proof outline.

## E.2 CLH lock

```
1  lock(x) {
2    c := alloc(1); [c] := 1;
3    p := FAS(x + 1, c);
4    v := [p];
5    while(v ≠ 0) {
6      v := [p];
7    }
8    [x] := c;
9  }
```

```
1  unlock(x) {
2    ⟨h := [x]; [h] := 0;⟩
3  }
```

$$\top \vdash \mathbb{V}l \in \{0,1\} \twoheadrightarrow_{\perp} \{0\}.\langle \mathsf{L}(s,\mathsf{x},l)\rangle\ \mathsf{lock(x)}\ \langle \mathsf{L}(s,\mathsf{x},1) \wedge l = 0\rangle$$
$$\perp \vdash \langle \mathsf{L}(s,\mathsf{x},1)\rangle\ \mathsf{unlock(x)}\ \langle \mathsf{L}(s,\mathsf{x},0)\rangle$$

$$\mathsf{L}(r,x,l) \triangleq \exists o \in \mathbb{N}.\ \mathbf{clh}_r(x,l,o) * \lfloor \mathsf{G} \rfloor_r$$
$$\mathrm{ones}_k(ns) \triangleq ns(k) \mapsto 1 * \cdots * ns(|ns|-1) \mapsto 1$$

$$I(\mathbf{clh}_r(x,l,o)) \triangleq \exists ns \in \mathsf{Addr}^+, t \in \mathbb{N}.\ x \mapsto \mathrm{fst}(ns), \mathrm{last}(ns) * \mathrm{fst}(ns) \mapsto l * \mathrm{ones}_1(ns) * \lfloor \mathsf{Q}(ns,o) \rfloor_r * \lceil \mathsf{O}(o,t) \rceil_r * \mathsf{True} \wedge t - o = |ns| - 1$$

**Interference:** $\mathsf{G} : (0,o) \rightsquigarrow (1,o+1) \quad \mathsf{G} : (1,o) \rightsquigarrow (0,o)$

**Guard algebra:** for any $p, c \in \mathsf{Addr}, ns \in \mathsf{Addr}^*, o, t \in \mathbb{N}$

$$\mathsf{Q}([p,c] \oplus ns, o) \bullet \mathsf{T}(p,c,o+1) = \mathsf{Q}(c \oplus ns, o+1)$$
$$\mathsf{Q}(ns,o) \bullet \mathsf{T}(p,c,t) \text{ defined} \iff ns(t-o-1) = p \wedge ns(t-o) = c$$
$$\mathsf{Q}(ns \oplus p, o) = \mathsf{Q}(ns \oplus [p,c], o) \bullet \mathsf{T}(p,c,o+|ns|+1)$$
$$\mathsf{G} \bullet \mathsf{G} \text{ undefined}$$

**Obligation algebra:** for any $o, o', t, t' \in \mathbb{N}$

$$\mathsf{O}(o,t) = \mathsf{O}(o,t+1) \bullet \mathsf{P}(t) \quad \mathsf{O}(o,o) = \mathsf{O}(o',o')$$
$$\mathsf{O}(o+1,t) = \mathsf{O}(o,t) \bullet \mathsf{P}(o+1) \quad \textbf{Total: } \mathsf{O}(0,0)$$
$$\mathsf{O}(o,t) \bullet \mathsf{P}(t') \text{ is defined} \iff o \le t' < t$$
$$\mathcal{L} \triangleq \mathbb{N} \cup \{\top, \perp\} \quad \mathrm{lay}(\mathsf{O}(o,o')) = 0 \quad \mathrm{lay}(\mathsf{P}(t)) = t$$

Let $\mathcal{A} = r : \forall l \in \{0,1\}, o, o' \in \mathbb{N}.\ (l,o) \twoheadrightarrow_{\perp} (0,o') \rightarrow (1,o+1) \wedge l = 0$.

The parameters for the while rule application are:

$$P(\beta) = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) * r \Longmapsto \blacklozenge * \lfloor \mathsf{T(p,c,}t\mathsf{)} \rfloor_r \wedge o < t \wedge (\mathsf{v} = 0 \Longrightarrow (t = o+1 \wedge l = 0)) \wedge \beta = \mathsf{v}$$
$$M(\alpha) = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) \wedge \alpha = 2(t-o) + l$$
$$L = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) * \lceil \mathsf{P}(t) \rceil_r * r \Longmapsto \Diamond$$
$$L' = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) * \lceil \mathsf{P}(t) \rceil_r$$
$$T = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) \wedge t = o+1 \wedge l = 0$$
$$E(\alpha) = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},l,o) * \circledast_{i=o}^{t-1} \lceil \mathsf{P}(i) \rceil_r \wedge o < t \wedge \alpha = 2(t-o) + l$$
$$E'(\alpha) = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},0,o) \wedge o < t \wedge \alpha = 2 * (t-o)$$
$$E''(\alpha) = \exists l \in \{0,1\}, o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},1,o) \wedge o < t \wedge \alpha = 2 * (t-o) + 1$$

To prove the environment liveness condition $\top; \mathcal{A} \vdash \langle P \mid L \rangle \xrightarrow{M} T : E(\alpha)$ we distinguish two cases by using the ELCase on $(E(\alpha) \wedge l = 0) \vee (E(\alpha) \wedge l = 1)$.

$$\dfrac{\dfrac{}{m;\lambda;\mathcal{A} \vdash \langle P \mid L \rangle \xrightarrow{M} T : \exists o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},0,o) * \lceil \mathsf{P}(o) \rceil_r * E'(\alpha)}\ \textsc{LiveO}}{m;\lambda;\mathcal{A} \vdash \langle P \mid L \rangle \xrightarrow{M} T : \exists o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},0,o) * \circledast_{i=o}^{t-1} \lceil \mathsf{P}(i) \rceil_r * E'(\alpha)}\ \textsc{ElFrame} \qquad \dfrac{}{m;\lambda;\mathcal{A} \vdash \langle P \,|\, L' * r \Longmapsto \Diamond \rangle \xrightarrow{M} T : \exists o \in \mathbb{N}.\ \mathbf{clh}_r(\mathsf{x},1,o) * E''(\alpha)}\ \textsc{LiveA}$$
$$m;\lambda;\mathcal{A} \vdash \langle P \mid L \rangle \xrightarrow{M} T : E(\alpha)\ \textsc{ElCase}$$

For $E(\alpha) \wedge l = 0$ we apply the LiveO on the obligation $\mathsf{P}(o)$, which is live because $o < t$. Then $\alpha$ decreases:

$$\lambda; \mathcal{A} \vdash \exists \alpha'. \left( \left( \begin{array}{c} \exists o' \in \mathbb{N}, l' \in \{0,1\}.\ \mathbf{clh}_r(\mathsf{x},l',o') * \lceil \mathsf{P}(t) \rceil_r * r \Longmapsto \blacklozenge \wedge \\ o < t \wedge \alpha' = 2 * (t-o') + l' \wedge \alpha' \le 2 * (t-o) \wedge \mathrm{Inv}_\lambda(\mathbf{t}_r^{\lambda'}, v) \end{array} \right) \wedge (\lceil \mathsf{P}(o) \rceil_r * \mathsf{True}) \right)$$

implies $o' > o$, and therefore $\alpha' < \alpha \vee (T * \mathsf{True})$.

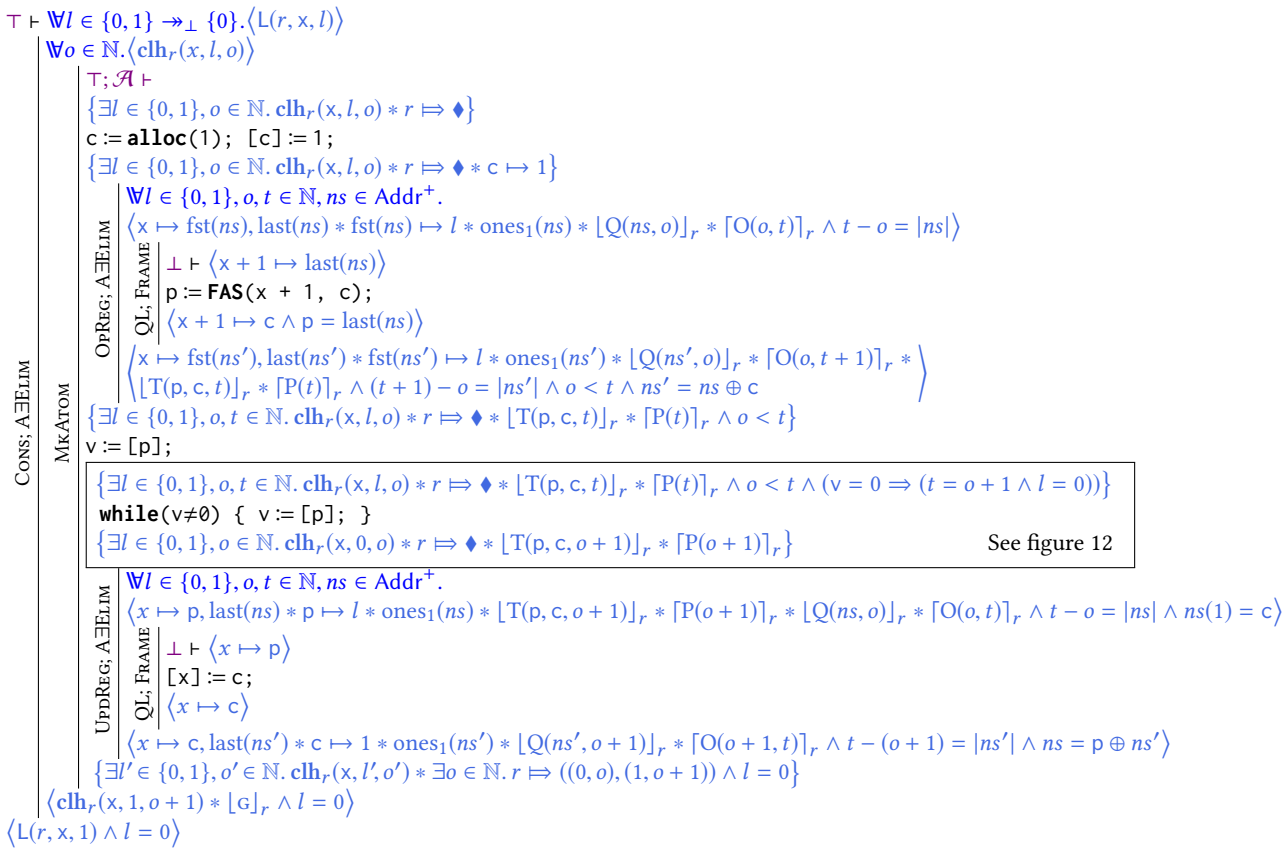



**Figure 11.** CLH lock `lock` operation proof outline.

For $E(\alpha) \wedge l = 1$ we apply the LiveA on the live assumption $(l, \_) \twoheadrightarrow_\perp (0, \_)$ in $\mathcal{A}$. Then $\alpha$ can be shown to decrease if the set $S = \{(0, o') \mid o' \geq o\}$ is reached:

$$0; \mathcal{A} \vdash \exists \alpha'. \left( \begin{array}{l} \exists o' \in \mathbb{N}, l' \in \{0,1\}.\ \mathbf{clh}_r(\mathsf{x}, l', o') * \lceil \mathsf{P}(t) \rceil_r * r \Rrightarrow \blacklozenge \wedge \\ o < t \wedge \alpha' = 2 * (t - o') \wedge \alpha' \leq 2 * (t - o) + 1 \wedge o' \geq o \end{array} \right)$$

which implies $\alpha' < \alpha \vee (T * \mathsf{True})$.

Unlike the original CLH lock implementation, the implementation of the `unlock` operation that we are using uses atomic brackets. This is because the original CLH lock implementation requires a notion of "ownership" that is not encoded in the basic TaDA-live lock specification. This notion of ownership is required to stabilize the head of the queue of cells waiting on the lock, so that it is not changed before its value is updated. The implementation without atomic brackets can be proved correct with respect to a specification that does encode a notion of ownership, for example:

$$\begin{array}{l} \top \vdash \forall l \in \{0,1\} \twoheadrightarrow_\perp \{0\}.\langle \mathsf{L}(\mathsf{x}, l) \rangle\ \mathtt{lock(x)}\ \langle \mathsf{L}(\mathsf{x}, 1) * \mathsf{Own}(\mathsf{x}) \rangle \\ \perp \vdash \langle \mathsf{L}(\mathsf{x}, 1) * \mathsf{Own}(\mathsf{x}) \rangle\ \mathtt{unlock(x)}\ \langle \mathsf{L}(\mathsf{x}, 0) \rangle \end{array}$$

with the axiom: $\mathsf{Own}(x) * \mathsf{L}(x, l)$ is defined. $\Leftrightarrow\ l = 1$.

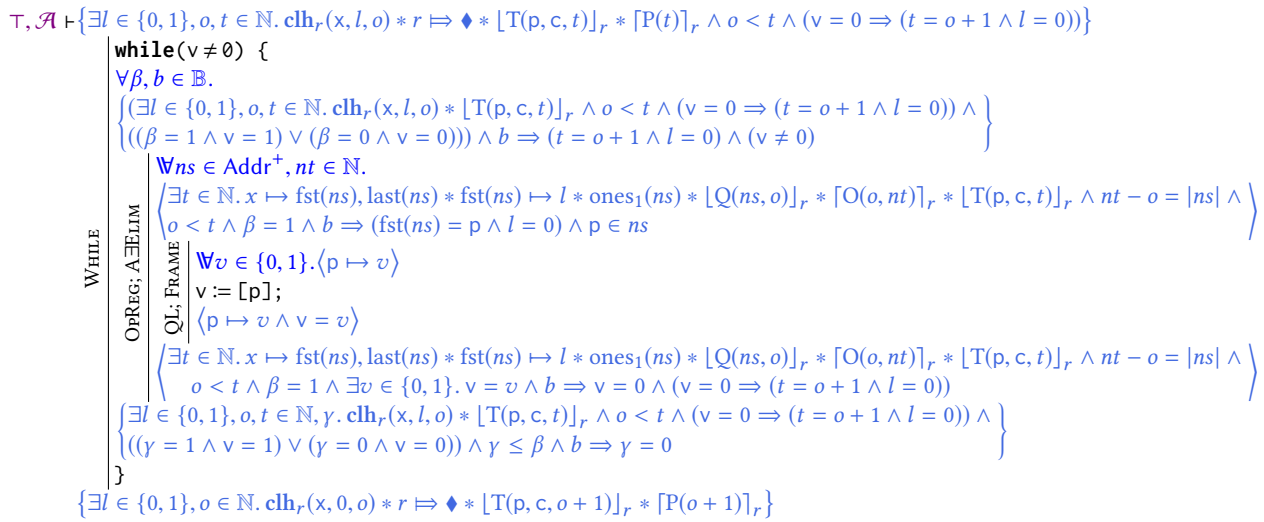



**Figure 12.** Details of CLH lock `lock` operation while loop.

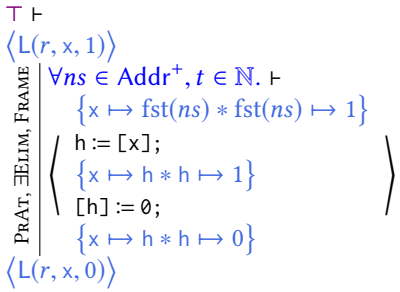



**Figure 13.** CLH lock `unlock` operation proof outline.

### E.3 Ticket lock

Consider a ticket lock implementation:

```
1  lock(x){
2   ⟨i := [x]; [x] := i+1⟩;
3    o := [x+1];
4    while(o ≠ i){ o := [x+1]; }
5    [x+2] := 1;
6  }
7
8  unlock(x){
9   ⟨[x+2] := 0; v := [x+1]; [x+1] := v+1⟩
10 }
```

$$I(\mathbf{tlock}_r(x,l,o)) \triangleq \exists t \in \mathbb{N}.\, x \mapsto o,t,l * \circledast_{i=0}^{o-1} \lceil \mathrm{T}(i) \rceil_r * (l \;\Leftrightarrow\; (\lceil \mathrm{T}(o) \rceil_r \wedge o < t)) * \lceil \mathrm{A}(t) \rceil_r \wedge o \leq t$$
$$\mathbf{0} : \forall o \in \mathbb{N}.\, (0,o) \rightsquigarrow (1,o) \qquad \mathbf{0} : \forall o \in \mathbb{N}.\, (1,o) \rightsquigarrow (0,o+1)$$

**Figure 14.** Interpretation, interference, guard and obligation algebras of the **tlock** region.

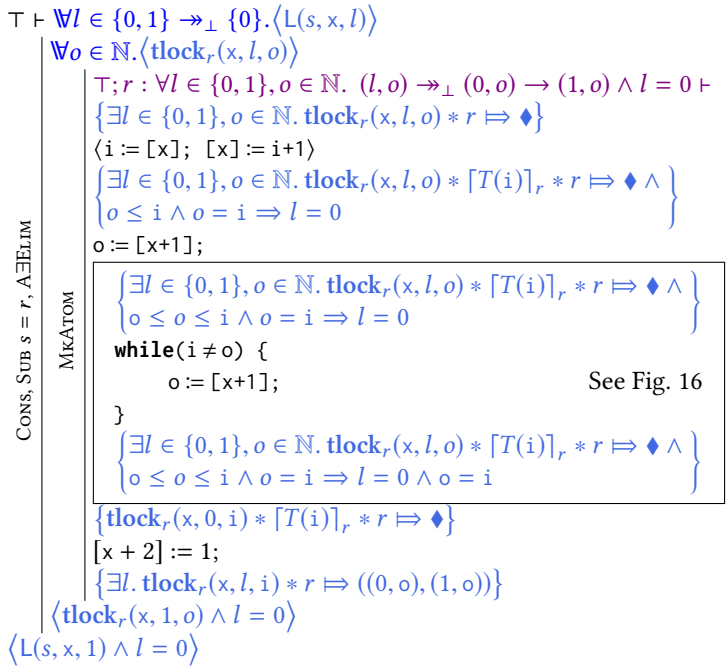



**Figure 15.** Ticket lock `lock` operation proof outline.

The reader may be familiar with ticket lock as an example of a data structure with linearization points that are not within the operation invocation; the ticket lock exhibits *helping*. Since TaDA does not support helping yet, we side-step the issue by injecting line 5 which effectively provides an artificial linearization point internal to the operation using ghost code. Adding support for helping to TaDA is an issue orthogonal to the goals of this paper.

We also note that, to the best of our knowledge, all other logics that can prove starvation freedom of ticket lock also do not support helping and apply similar workarounds using ghost state.

A ticket lock is starvation-free, as we will prove by showing it satisfies the same specifications of CLH lock.

To verify the implementation, we define the L abstract predicate as $\mathsf{L}(r,x,l) \triangleq \exists o \in \mathbb{N}.\, \mathbf{tlock}_r(x,l,o)$, where the **tlock** region is defined in Fig. 14. The obligation algebra represents tickets with an obligation $\mathrm{T}(i)$, and the available tickets from $i$ with $\mathrm{A}(i)$. We have: $\mathbf{1} = \mathrm{A}(0)$, $\forall i,j \in \mathbb{N}.\, \mathrm{A}(i) \bullet \mathrm{A}(j)$ is undefined, and $\forall i \in \mathbb{N}.\, \mathrm{A}(i) \Rightarrow \mathrm{T}(i) \bullet \mathrm{A}(i+1)$, with $\mathrm{lay}(\mathrm{A}(i)) = 0$ and $\mathrm{lay}(\mathrm{T}(i)) = i$. The proof of the while statement is shown in Fig. 16. To verify the environment liveness condition, we can set the target $T \triangleq \exists l \in \{0,1\}, o \in \mathbb{N}.\, \mathbf{tlock}_r(\mathrm{x},l,o) \wedge \mathrm{i} = o$, $L = \lceil \mathrm{T}(\mathrm{i}) \rceil_r * r \Rrightarrow \Diamond$ because we hold $\mathrm{T}(\mathrm{i})$ throughout the while and not use it, and we have not performed our linearization point yet (hence $r \Rrightarrow \Diamond$). Finally, the environment progress measure $M(\alpha) \triangleq \exists l \in \{0,1\}, o \in \mathbb{N}.\, \mathbf{tlock}_r(\mathrm{x},l,o) \wedge o \leq \mathrm{i} \wedge \alpha = (2(\mathrm{i}-o)-l) \wedge (\mathrm{i} = o \Rightarrow l = 0)$. From these parameters we can extract:

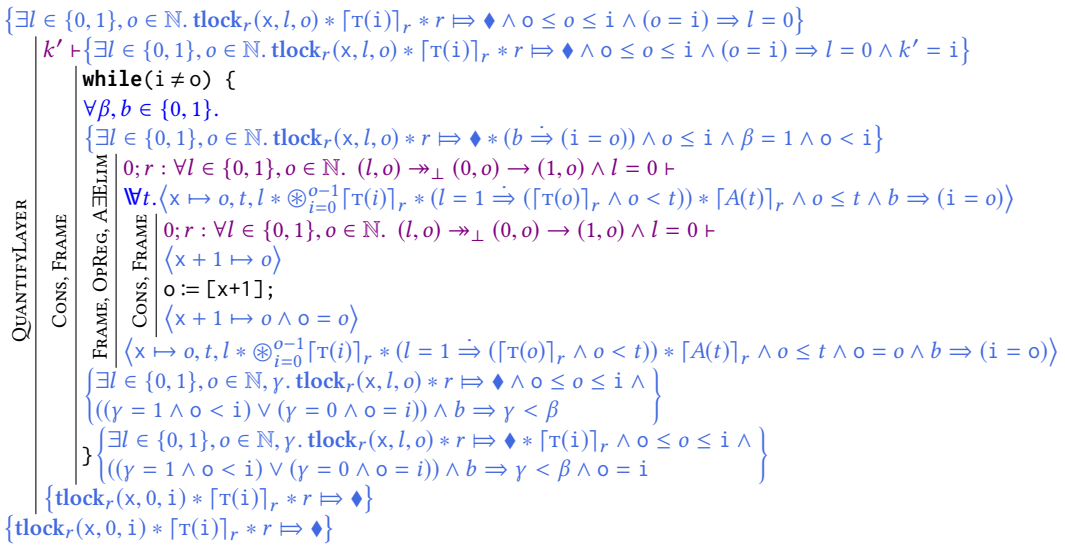



**Figure 16.** Proof outline of the ticket lock operation.

$E(\alpha) \triangleq \exists l \in \{0,1\}, o \in \mathbb{N}.\, \mathbf{tlock}_r(\mathtt{x}, l, o) \wedge o < \mathtt{i} \wedge \alpha = (2(\mathtt{i} - o) - l) * \circledast_{j=o+1}^{\mathtt{i}} \lceil \mathrm{T}(j) \rceil_r * (l = 0 \dot{\Rightarrow} \lceil \mathrm{T}(o) \rceil_r)$. This expresses which obligations are not held locally nor in the region during the while loop and when $T$ does not hold. Now we can use the the Liveness case analysis rule to break $E(\alpha)$ in two cases: either $l = 0$ or $l = 1$. When $l = 0$, we can infer the obligation $\mathrm{T}(o)$ is in the environment, in a state where we know $o < \mathtt{i}$; the layers we can use, because we are holding an obligation of layer $\mathtt{i}$, are the ones strictly smaller than $\mathtt{i}$ so the obligation $\mathrm{T}(o)$, of layer $o$, can be assumed live. The only transitions that can place it back in the region is the one that sets $l = 1$, in which case the measure decreases. For the case where $l = 1$, we can use the pseudo-quantifier liveness assumption in the atomicity context and infer that $l$ will be brought to 0 eventually, but by incrementing $o$, thus making the measure decrease.

### E.4 A blocking impeded counter

Consider a data structure representing an integer, the concurrent access of which is protected with spin locks:

```
1  incr(x){                    8  read(x){
2    y := [x];                 9    y := [x];
3    lock(y);                  10   lock(y);
4    v := [x+1];               11   v := [x+1];
5    [x+1] := v+1;             12   unlock(y);
6    unlock(y);                13   return v;
7  }                           14 }
```

The operations are specified as follows:

$$\forall \phi.\ \bot \vdash \mathbb{W} n \in \mathbb{N}, \alpha. \left\langle \mathrm{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n) \right\rangle \ \mathtt{incr(x)}\ \left\langle \mathrm{C}(s, \mathtt{x}, n+1, \phi(\alpha, n)) \right\rangle$$

$$\forall \phi.\ \bot \vdash \mathbb{W} n \in \mathbb{N}, \alpha. \left\langle \mathrm{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n) \right\rangle \ \mathtt{read(x)}\ \left\langle \mathrm{C}(s, \mathtt{x}, n, \phi(\alpha, n)) \wedge \mathrm{ret} = n \right\rangle$$

The specifications express the fact that both operations can impede each other; this is a direct consequence of using a spin lock instead of a ticket lock, and the specifications reflect this termination property.

We give the following abstract predicate and the region definitions:

$$\mathsf{C}((r, sl, la), x, n, \alpha) \triangleq \mathbf{cnt}_r((sl, la), x, n, \alpha)$$

$$\forall n, m \in \mathbb{N}, \alpha, \beta < \alpha.\ \mathbf{0}\ :\ (n, \alpha) \rightsquigarrow (m, \beta)$$

$$I(\mathbf{cnt}_r((lr, y), x, n, \alpha)) \triangleq \exists l \in \{0, 1\}, m \in \mathbb{N}, \beta.\ x \mapsto y, m * \mathsf{L}(lr, y, l, \beta) * \begin{pmatrix} (l = 0 \wedge m = n \wedge \alpha = \beta \wedge \lceil \textsc{r} \rceil_r * \lfloor \textsc{u} \rfloor_r) \\ \vee\ (l = 1 \wedge \lfloor \textsc{l}(n, \alpha, m, \beta) \rfloor_r) \end{pmatrix}$$

The obligation algebra consists of a single obligation $\mathbf{1} = \textsc{r}$, with $\mathrm{lay}(\textsc{r}) = 0 > \bot$. In the interpretation, we use an untagged abstract predicate $\mathsf{L}$ which is an abbreviation for the same predicate tagged with the identity on layers: the external layer $\bot$ of the lock is mapped to the counter's layer $\bot$.

The guard algebra is defined so that the ghost state can keep track of the global counter and the local contributions of the thread to the value of the counter, until the linearisation point is performed, at which point the "official value" of the counter will be the global value plus the local contribution.

$$\forall n, m \in \mathbb{N}, \alpha, \beta.\ \textsc{u} = \textsc{l}(n, \alpha, m, \beta) \bullet \textsc{k}(n, \alpha, m, \beta)$$

$$\forall n, n', m, m', \alpha, \alpha', \beta, \beta'.\ (n \neq n' \vee m \neq m' \vee \alpha \neq \alpha' \vee \beta \neq \beta') \Rightarrow \textsc{l}(n, \alpha, m, \beta) \bullet \textsc{l}(n', \alpha', m', \beta') \text{ is undefined.}$$

$$\forall n, m, \alpha, \beta.\ \textsc{u} \bullet \textsc{k}(n, \alpha, m, \beta) \text{ is undefined.}$$

This follows a common pattern established in the TaDA methodology.

The proof outline is as follows.

$\forall\phi.\ \vdash\ \mathbb{W}n \in \mathbb{N}, \alpha.$
$\langle \mathrm{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n)\rangle$
Cons; substitute $s = (r, sl, la)$
MakeAtomic
$\langle \mathbf{cnt}_r((sl, la), \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n)\rangle$
$0;\ r : \forall n \in \mathbb{N}, \alpha, \beta < \alpha.(n, \alpha) \longrightarrow (n+1, \beta) \vdash$
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, la), \mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \alpha > \phi(\alpha, n)\}$
`y := [x];`
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \alpha > \phi(\alpha, n)\}$
Frame
$0;\ r : \forall n \in \mathbb{N}, \alpha, \beta < \alpha.(n, \alpha) \longrightarrow (n+1, \beta) \vdash$ See below
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n)\}$
`lock(y);`
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n, \phi(\alpha, n))\rfloor_r * \lceil \mathrm{R}\rceil_r \wedge \alpha > \phi(\alpha, n)\}$
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n, \phi(\alpha, n))\rfloor_r * \lceil \mathrm{R}\rceil_r * r \Rrightarrow \blacklozenge \wedge \alpha > \phi(\alpha, n)\}$
Frame
$\bot;\ r : \forall n \in \mathbb{N}, \alpha, \beta < \alpha.(n, \alpha) \longrightarrow (n+1, \beta) \vdash$
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n, \phi(\alpha, n))\rfloor_r\}$
`v := [x+1];`
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n, \phi(\alpha, n))\rfloor_r \wedge \mathtt{v} = n\}$
`[x+1] := v+1;`
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n+1, \phi(\alpha, n))\rfloor_r\}$
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n+1, \phi(\alpha, n))\rfloor_r * \lceil U\rceil_r * r \Rrightarrow \blacklozenge \wedge \alpha > \phi(\alpha, n)\}$
`unlock(y);`
$\{r \Rrightarrow ((n, \alpha), (n+1, \phi(\alpha, n)))\}$
$\langle \mathbf{cnt}_r((sl, la), \mathtt{x}, n+1, \phi(\alpha, n))\rangle$
$\langle \mathrm{C}(s, \mathtt{x}, n+1, \phi(\alpha, n))\rangle$

We prove the `lock(y)` call separately as follows:

$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) \wedge \alpha > \phi(\alpha, n)\}$
Ex, AtomW, OpReg, A∃El, Frame
$0;\ \forall n \in \mathbb{N}, \alpha, \beta < \alpha.(n, \alpha) \longrightarrow (n+1, \beta) \vdash$
$\mathbb{W}m \in \mathbb{N}, l \in \{0, 1\}, \beta.$
$\left\langle \mathrm{L}(sl, \mathtt{y}, l, \beta) * \begin{pmatrix} (l = 0 \wedge m = n \wedge \alpha = \beta \wedge \lceil \mathrm{R}\rceil_r * \lfloor \mathrm{U}\rfloor_r) \vee \\ (l = 1 \wedge \lfloor \mathrm{L}(n, \alpha, m, \beta)\rfloor_r) \end{pmatrix} \wedge \alpha > \phi(\alpha, n)\right\rangle$
Cons
$\left\langle \mathrm{L}(sl, \mathtt{y}, l, \beta) * \begin{pmatrix} (l = 0 \wedge m = n \wedge \alpha = \beta \wedge \lceil \mathrm{R}\rceil_r * \lfloor \mathrm{U}\rfloor_r) \vee \\ (l = 1 \wedge \lfloor \mathrm{L}(n, \alpha, m, \beta)\rfloor_r) \end{pmatrix} \wedge \alpha > \phi(\alpha, n)\right\rangle$
LiveC, Frame
$0;\ \forall n \in \mathbb{N}, \alpha, \beta < \alpha.\ (n, \alpha) \longrightarrow (n+1, \beta) \vdash$
$\mathbb{W}l \in \{0, 1\} \twoheadrightarrow_\bot \{0\}, \alpha.$
$\langle \mathrm{L}(sl, \mathtt{y}, l, \alpha) \wedge \alpha > \phi(\alpha, n)\rangle$
`lock(y);`
$\langle \mathrm{L}(sl, \mathtt{y}, 1, \phi(\alpha, n)) \wedge l = 0\rangle$
$\langle \mathrm{L}(sl, \mathtt{y}, 1, \phi(\alpha, n)) * ((l = 0 \wedge m = n \wedge \alpha = \beta \wedge \lceil \mathrm{R}\rceil_r * \lfloor \mathrm{U}\rfloor_r) \vee (l = 1 \wedge \lfloor \mathrm{L}(n, \alpha, m, \beta)\rfloor_r)) \wedge l = 0\rangle$
$\langle \mathrm{L}(sl, \mathtt{y}, 1, \phi(\alpha, n)) * \lfloor \mathrm{L}(n, \alpha, m, \phi(\alpha, n))\rfloor_r * \lfloor \mathrm{K}(n, \alpha, m, \phi(\alpha, n))\rfloor_r * \lceil \mathrm{R}\rceil_r\rangle$
$\{\exists n \in \mathbb{N}, \alpha.\ \mathbf{cnt}_r((sl, \mathtt{y}), \mathtt{x}, n, \alpha) * \lfloor \mathrm{K}(n, \alpha, n, \phi(\alpha, n))\rfloor_r * \lceil \mathrm{R}\rceil_r \wedge \alpha > \phi(\alpha, n)\}$

The parameters for the application of the LiveC rule are as follows:

$$L \triangleq \mathsf{emp}$$
$$M(\alpha) \triangleq \exists l \in \{0,1\}, \delta.\, \mathsf{L}(sl, \mathsf{y}, l, \delta) \wedge \alpha = 2\delta + l$$
$$T \triangleq \exists \delta.\, \mathsf{L}(sl, \mathsf{y}, 1, \delta)$$

### E.5 Lock-coupling list set

```
add(x, e) {
  p := locate(x, e);
  c := p.next;
  v := c.val;

  if(v ≠ e) {
    n := alloc(3);
    nl := newLock();
    n.lock := nl;
    n.val := e;
    n.next := c;
    p.next := n;
    unlock(nl);
  }

  pl := p.lock;
  cl := c.lock;
  unlock(cl);
  unlock(pl);
}
```

```
rem(x, e) {
  p := locate(x, e);
  c := p.next;
  v := c.val;

  if(v = e) {
    n := c.next;
    p.next = n;
  }

  pl := p.lock;
  cl := c.lock;
  unlock(pl);
  unlock(cl);
}
```

```
locate(x, e) {
  hl := [x];
  lock(hl);

  p := [x + 1];
  pl := p.lock;
  lock(pl);
  c := p.next;
  cl := c.lock;
  lock(cl);
  v := c.value;
  unlock(hl);

  while((v < e) && (v ≠ -1)) {
    pl := p.lock;
    c' := c.next;
    cl' := c'.lock;
    lock(cl');
    v := c'.val;
    unlock(pl);
    p := c;
    c := c';
  }

  return p;
}
```

***Abbreviations*** We use the following abbreviations in the code, for readability:

$$\mathtt{x.lock} = [\mathtt{x}] \qquad \mathtt{x.val} = [\mathtt{x}+1] \qquad \mathtt{x.next} = [\mathtt{x}+2]$$

In the proof, when an obligation O belongs both to the obligation and guard algebras, we write $[\mathrm{O}]_r$ as an abbreviation for $\lfloor \mathrm{O} \rfloor_r * \lceil \mathrm{O} \rceil_r$.

$$\top \vdash \left\{ \exists S.\, \mathbf{lcset}_r(hl, \mathtt{x}, S) \right\} \mathtt{locate(x, e)} \left\{ \begin{array}{c} \exists l, \bar{l}, v, v', h, lay, \overline{lay}, S.\, \mathbf{lcset}_r(hl, \mathtt{x}, S) * \\ \lfloor \mathrm{K}(\mathtt{ret}, l, v, h, lay+1) \rfloor_r * \lceil \textsc{Unlck}(v, lay+1) \rceil_r * \\ \lfloor \mathrm{K}(h, \bar{l}, \overline{v}, \_, \overline{lay}) \rfloor_r * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}) \rceil_r * \\ \lfloor \textsc{Free}(lay) \rfloor_r \wedge lay > \overline{lay} \wedge v < e \leq v' \end{array} \right\}$$

$$\top \vdash \mathbb{V} S \in \mathcal{P}(\mathbb{N}).\, \langle \mathrm{LCSet}(s, \mathtt{x}, S) \rangle\ \mathtt{add(x, e)}\ \langle \mathrm{LCSet}(s, \mathtt{x}, S \cup \{\mathtt{e}\}) \rangle$$
$$\top \vdash \mathbb{V} S \in \mathcal{P}(\mathbb{N}).\, \langle \mathrm{LCSet}(s, \mathtt{x}, S) \rangle\ \mathtt{remove(x, e)}\ \langle \mathrm{LCSet}(s, \mathtt{x}, S \setminus \{\mathtt{e}\}) \rangle$$

$$\mathsf{LCSet}((r, r', hl), x, S) \triangleq \mathbf{lcset}_r(r', hl, x, S) * \lfloor \mathrm{G} \rfloor_r$$
$$\mathrm{G} \ : \ \forall e.\, S \rightsquigarrow S \cup \{e\}$$
$$\mathrm{G} \ : \ \forall e.\, S \rightsquigarrow S \setminus \{e\}$$

$$I(\mathbf{lcset}_r(r', hl, x, S)) \triangleq \exists S' \in \mathcal{P}(\mathbb{N} \times \{0,1\}).\, \mathbf{lcseth}_{r'}(hl, x, S') * \lfloor \mathrm{G'} \rfloor_{r'} \wedge S = \{v \mid (v, \_) \in S'\}$$

$$\mathrm{G'} \ : \ \forall e.\, S \rightsquigarrow S \cup \{(e,0)\} \qquad (e, \_) \notin S$$
$$\mathrm{G'} \ : \ \forall e.\, S \cup \{(e,1)\} \rightsquigarrow S$$
$$\mathrm{G'} \ : \ \forall e.\, S \cup \{(e,0)\} \rightsquigarrow S \cup \{(e,1)\}$$
$$\mathrm{G'} \ : \ \forall e.\, S \cup \{(e,1)\} \rightsquigarrow S \cup \{(e,0)\}$$

$$I(\mathbf{lcseth}_r(hl, x, S')) \triangleq \exists s, l, h, ls, lay, clay.\, x \mapsto hl, h * list(h, ls, clay) * \mathsf{TLock}(s, hl, l) *$$
$$((l = 0 \wedge clay = lay \wedge \lceil \mathrm{HU} \rceil_r) \vee (l = 1 \wedge lay = clay + 1 \wedge \lceil \mathrm{HL}(h) \rceil_r)) * [\textsc{Cont}(S', lay)]_r \wedge$$
$$(\forall v \in \mathbb{N}, l \in \{0,1\}.\, (v,l) \in S' \iff v \in vals(ls)) \wedge ord(ls)$$

$$list(h, ls, lay) = \exists sl, cl, l, h', lay'.\, h \mapsto cl, -1, h' * list'(h', ls, lay') * \mathsf{TLock}(sl, cl, l) *$$
$$((l = 0 \wedge \lfloor \mathrm{U}(h, cl, -1, lay) \rfloor_r * \lceil \textsc{Unlck}(-1, lay) \rceil_)) \vee (l = 1 \wedge \lfloor \mathrm{L}(h, cl, -1, h', lay) \rfloor_r)) \wedge lay > lay'$$
$$list'(h, [], 0) \triangleq \exists sl, cl, l.\, h \mapsto cl, \infty, \mathtt{null} * \mathsf{TLock}(sl, cl, l) *$$
$$((l = 0 \wedge \lfloor \mathrm{U}(h, cl, \infty, 0) \rfloor_r * \lceil \textsc{Unlck}(\infty, 0) \rceil_r) \vee (l = 1 \wedge \lfloor \mathrm{L}(h, cl, \infty, \mathtt{null}, 0) \rfloor_r))$$
$$list'(h, v : ls, lay) \triangleq \exists s, cl, l, h', lay'.\, h \mapsto l, v, h' * list(h', ls) * \mathsf{TLock}(s, cl, l) *$$
$$((l = 0 \wedge \lfloor \mathrm{U}(h, cl, v, lay) \rfloor_r * \lceil \textsc{Unlck}(v, lay) \rceil_r) \vee (l = 1 \wedge \lfloor \mathrm{L}(h, cl, v, h', lay) \rfloor_r))$$
$$ord([]) = \top$$
$$ord([v]) = \top$$
$$ord(v : v' : ls) = v < v' \wedge ord(v' : ls)$$

guard algebra:

$$\textsc{Cont}(S, lay) = \textsc{Free}(lay + 1) \bullet \textsc{Cont}(S, lay + 1)$$
$$\textsc{Cont}(S, lay) \bullet \textsc{Free}(lay') \text{ is defined.} \iff lay \geq lay'$$
$$\textsc{Free}(lay) \bullet \textsc{Free}(lay) \text{ is undefined.}$$
$$\textsc{Free}(lay) \bullet \mathrm{L}(h, l, v, lay) = \mathrm{L}(h, l, v, lay + 1) \bullet \textsc{Free}(lay') \qquad lay' < lay$$
$$\textsc{Cont}(S, lay') * \textsc{Free}(lay) = \textsc{Cont}(S \uplus \{(v, 0)\}, lay') \bullet \mathrm{U}(h, l, v, lay)$$
$$\textsc{Cont}(S, lay') \bullet \mathrm{K}(h, l, v, lay) \text{ is defined.} \iff (v, 1) \in S$$
$$\mathrm{U}(h, l, v, lay) = \mathrm{U}(h, l, v, lay) \bullet \mathrm{W}(h, l, v)$$
$$\mathrm{L}(h, l, v, lay) = \mathrm{L}(h, l, v, lay) \bullet \mathrm{W}(h, l, v)$$
$$\textsc{Cont}(S) \bullet \mathrm{W}(h, l, v) \iff (v, \_) \in S$$
$$\textsc{Cont}(S \uplus \{(v, 0)\}, lay') \bullet \mathrm{U}(h, l, v, lay) = \textsc{Cont}(S \uplus \{(v, 1)\}, lay') \bullet \mathrm{L}(h, l, v, h', lay) \bullet \mathrm{K}(h, l, v, h', lay)$$
$$\mathrm{K}(h, l, v, lay) \bullet \mathrm{L}(h, l, v, lay) \bullet \textsc{Free}(lay) \bullet \textsc{Cont}(S \uplus \{(v, 1)\}, lay') = \textsc{Cont}(S, lay')$$
$$\mathrm{L}(h, l, v, h', lay) \bullet \mathrm{K}(\overline{h}, \overline{l}, v, \overline{h'}, \overline{lay}) \text{ is defined.} \iff h = \overline{h} \wedge l = \overline{l} \wedge h' = \overline{h'} \wedge lay = \overline{lay}$$
$$\mathrm{U}(h, l, v, lay) \bullet \mathrm{K}(h', l', v, lay') \text{ is undefined.}$$
$$\mathrm{U}(h, l, v, \_) \bullet \mathrm{W}(h', l', v') \text{ is defined.} \iff (h = h' \Rightarrow (l = l' \wedge v = v'))$$
$$\mathrm{L}(h, l, v, \_) \bullet \mathrm{W}(h', l', v') \text{ is defined.} \iff (h = h' \Rightarrow (l = l' \wedge v = v'))$$

Obligation algebra:

$$\mathrm{HU} = \mathrm{HL}(h) \bullet \mathrm{HK}(h)$$
$$\mathrm{HL}(h) \bullet \mathrm{HK}(h') \text{ is defined.} \iff h = h'$$
$$\mathrm{HU} \bullet \mathrm{HK}(h) \text{ is undefined.}$$
$$\textsc{Cont}(S, lay) = \textsc{Cont}(S, lay')$$
$$\textsc{Cont}(S, lay) = \textsc{Cont}(S \uplus \{(v, 0)\}, lay) \bullet \textsc{Unlck}(v, lay')$$
$$\textsc{Cont}(S \uplus \{(v, 0)\}) = \textsc{Cont}(S \uplus \{(v, 1)\})$$

$$\mathrm{lay}(\textsc{Unlck}(v, lay)) = lay$$
$$\mathrm{lay}(\mathrm{HK}(\_)) = \top$$

Total: $\textsc{Cont}(\emptyset, \_)$


⊤ ⊢
∀S ∈ 𝒫(ℕ).
⟨LCSet(s, x, S)⟩
⟨lcset_r(r′, hl, x, S) ∗ ⌊G⌋_r⟩
⊤; r : ∀S ∈ 𝒫(ℕ). S → S ∪ {e} ⊢
{∃S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦}
p := locate(x, e);
{∃la, l̄a, v, v̄, h, lay, l̄ay, S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦ ∗ ⌊K(p, la, v, h, lay + 1)⌋_r′ ∗ ⌈Unlck(v, lay + 1)⌉_r′ ∗
⌊K(h, l̄a, v̄, _, l̄ay)⌋_r′ ∗ ⌈Unlck(v̄, l̄ay)⌉_r′ ∗ ⌊Free(lay)⌋_r′ ∧ v < e ≤ v̄ ∧ lay > l̄ay}
⊥; r : ∀S ∈ 𝒫(ℕ). S → S ∪ {e} ⊢
{∃la, l̄a, v, v̄, h, lay, l̄ay, S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦ ∗ ⌊K(p, la, v, h, lay + 1)⌋_r′ ∗
⌊K(h, l̄a, v̄, _, l̄ay)⌋_r′ ∗ ⌊Free(lay)⌋_r′ ∧ v < e ≤ v̄ ∧ lay > l̄ay}
c := p.next; v := c.val;
{∃la, l̄a, v, v̄, lay, l̄ay, S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦ ∗ ⌊K(p, la, v, c, lay + 1)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∗
⌊Free(lay)⌋_r′ ∧ v < e ≤ v̄ ∧ lay > l̄ay ∧ v ≠ e ⇒ e < v̄}
if (v ≠ e) {
{∃la, l̄a, v, v̄, lay, l̄ay, S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦ ∗ ⌊K(p, la, v, c, lay + 1)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∗
⌊Free(lay)⌋_r′ ∧ v < e < v̄ ∧ lay > l̄ay}
n := alloc(3); nl := newLock(); n.lock := nl; n.val := e; n.next := c;
{∃la, l̄a, v, v̄, lay, l̄ay, S. lcset_r(r′, hl, x, S) ∗ r ⇛ ♦ ∗
n ↦ nl, e, c ∗ L(s, nl, 0) ∗ ⌊K(p, la, v, c, lay + 1)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∗
⌊Free(lay)⌋_r′ ∧ v < e < v̄ ∧ lay > l̄ay}
p.next := n;
{∃la, l̄a, lay, l̄ay, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗ ⌈Unlck(e, lay)⌉_r′ ∗
⌊K(p, la, v, n, lay + 1)⌋_r′ ∗ ⌊K(n, nl, e, c, lay)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∧ lay > l̄ay}
unlock(nl);
{∃la, l̄a, lay, l̄ay, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗
⌊K(p, la, v, n, lay)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∧ lay > l̄ay}
}
{∃la, l̄a, lay, l̄ay, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗
⌊K(p, la, v, _1, lay)⌋_r′ ∗ ⌊K(c, l̄a, v̄, _, l̄ay)⌋_r′ ∧ lay > l̄ay}
pl := p.lock; cl := c.lock;
{∃lay, l̄ay, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗
⌊K(p, pl, v, _1, lay)⌋_r′ ∗ ⌊K(c, cl, v̄, _, l̄ay)⌋_r′ ∧ lay > l̄ay}
{∃lay, l̄ay, v, v̄, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗
⌊K(p, pl, v, _, lay)⌋_r′ ∗ ⌈Unlck(v, lay)⌉_r′ ∗ ⌊K(c, cl, v̄, _, l̄ay)⌋_r′ ∗ ⌈Unlck(v̄, l̄ay)⌉_r′ ∧ lay > l̄ay}
unlock(cl);
{∃lay, v, S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e}) ∗ ⌊K(p, pl, v, _, lay)⌋_r′ ∗ ⌈Unlck(v, lay)⌉_r}
unlock(pl);
{∃S, S′. lcset_r(r′, hl, x, S′) ∗ r ⇛ (S, S ∪ {e})}
⟨lcset_r(r′, hl, x, S) ∗ ⌊G⌋_r⟩
⟨LCSet(s, x, S ∪ {e})⟩
Cons; Sub s = (r, r′, hl)
Make Atomic
∃Elim; QL; Cons; Frame

$$\left\{\begin{array}{l}\exists lay, \overline{lay}, v, \overline{v}, S, S'.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S') * r \Mapsto (S, S \cup \{\mathtt{e}\}) * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, v, \_, lay)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay)\rceil_{r'} * \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, \overline{v}, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'} \wedge lay > \overline{lay}\end{array}\right\}$$

QL; Cons; Frame

$\overline{lay}; r : \forall S \in \mathcal{P}(\mathbb{N}).\ S \to S \cup \{\mathtt{e}\} \vdash$

$$\left\{\begin{array}{l}\exists \overline{v}, S'.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S') * \\ \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, \overline{v}, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'}\end{array}\right\}$$

`unlock(cl);`

$$\left\{\exists \overline{v}, S'.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S')\right\}$$

$$\left\{\exists lay, v, S, S'.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S') * r \Mapsto (S, S \cup \{\mathtt{e}\}) * \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, v, \_, lay)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay)\rceil_{r'}\right\}$$

$\{\exists S.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S)\}$
`hl := [x];`
$\{\exists S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S)\}$
`lock(hl);`
$\{\exists h, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(h) \rceil_{r'}\}$
`p := [x + 1];`
$\{\exists S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \mathrm{W}(\mathtt{p}, pl, -1) \rfloor_{r'}\}$
`pl := p.lock;`
$\{\exists S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \mathrm{W}(\mathtt{p}, \mathtt{pl}, -1) \rfloor_{r'}\}$
`lock(pl);`
$\{\exists lay, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \_, lay) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay) \rceil_{r'} * \lfloor \textsc{Free}(lay + 1) \rfloor_{r'}\}$
`c := p.next;`
$\{\exists cl, lay, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay) \rceil_{r'} * \lfloor \mathrm{W}(\mathtt{c}, cl, \_) \rfloor_{r'} * \lfloor \textsc{Free}(lay + 1) \rfloor_{r'}\}$
`cl := c.lock;`
$\{\exists lay, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay) \rceil_{r'} * \lfloor \mathrm{W}(\mathtt{c}, \mathtt{cl}, \_) \rfloor_{r'} * \lfloor \textsc{Free}(lay + 1) \rfloor_{r'}\}$
`lock(cl);`
$\left\{\begin{array}{l}\exists v, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \textsc{Free}(lay) \rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay + 1) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay + 1) \rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, v, \_, \overline{lay}) \rfloor_{r'} * \lceil \textsc{Unlck}(v, \overline{lay}) \rceil_{r'} \wedge lay > \overline{lay}\end{array}\right\}$
`v := c.value;`
$\left\{\begin{array}{l}\exists v, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lceil \mathrm{HK}(\mathtt{p}) \rceil_{r'} * \lfloor \textsc{Free}(lay) \rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay + 1) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay + 1) \rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, v, \_, \overline{lay}) \rfloor_{r'} * \lceil \textsc{Unlck}(v, \overline{lay}) \rceil_{r'} \wedge \\ lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge v = \infty) \vee (\mathtt{v} \geq 0 \wedge \mathtt{v} = v))\end{array}\right\}$
`unlock(hl);`
$\left\{\begin{array}{l}\exists v, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(lay) \rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay + 1) \rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay + 1) \rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, v, \_, \overline{lay}) \rfloor_{r'} * \lceil \textsc{Unlck}(v, \overline{lay}) \rceil_{r'} \wedge \\ lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge v = \infty) \vee (\mathtt{v} \geq 0 \wedge \mathtt{v} = v))\end{array}\right\}$
`while ((v < e) && (v ≠ −1)) {`
  `pl := p.lock;`
  `c' := c.next;`
  `cl' := c'.lock;`
  `lock(cl');`
  `v := c'.val;`
  `unlock(pl);`
  `p := c;`
  `c := c';`
`}`
$\left\{\begin{array}{l}\exists l, \bar{l}, v, \overline{v}, h, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S) * \\ \lfloor \mathrm{K}(\mathtt{p}, l, v, h, lay + 1) \rfloor_{r'} * \lceil \textsc{Unlck}(v, lay + 1) \rceil_{r'} * \lfloor \mathrm{K}(h, \bar{l}, \overline{v}, \_, \overline{lay}) \rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}) \rceil_{r'} * \\ \lfloor \textsc{Free}(lay) \rfloor_{r'} \wedge lay > \overline{lay} \wedge v < e \leq \overline{v}\end{array}\right\}$
`return p;`
$\left\{\begin{array}{l}\exists l, \bar{l}, v, \overline{v}, h, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S) * \\ \lfloor \mathrm{K}(\mathtt{ret}, l, v, h, lay + 1) \rfloor_{r'} * \lceil \textsc{Unlck}(v, lay + 1) \rceil_{r'} * \lfloor \mathrm{K}(h, \bar{l}, \overline{v}, \_, \overline{lay}) \rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}) \rceil_{r'} * \\ \lfloor \textsc{Free}(lay) \rfloor_{r'} \wedge lay > \overline{lay} \wedge v < e \leq \overline{v}\end{array}\right\}$

$$\left\{\begin{array}{l}\exists v, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(lay)\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, -1, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(-1, lay+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, \mathtt{cl}, v, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(v, \overline{lay})\rceil_{r'} \wedge \\ lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge v = \infty) \vee (\mathtt{v} \geq 0 \wedge \mathtt{v} = v))\end{array}\right\}$$

`while ((v < e) && (v ≠ −1)) {`

$\forall \beta.$

$$\left\{\begin{array}{l}\exists pl, cl, v, \overline{v}, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(lay)\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, pl, v, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'} \wedge v < \mathtt{e} \wedge \mathtt{v} < \mathtt{e} \\ \beta \geq lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge \overline{v} = \infty) \vee (\mathtt{v} \geq 0 \wedge \overline{v} = \mathtt{v}))\end{array}\right\}$$

`pl := p.lock;`

`c' := c.next;`

`cl' := c'.lock;`

$$\left\{\begin{array}{l}\exists cl, v, \overline{v}, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(lay)\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, v, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \mathtt{c'}, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'} * \lfloor \mathrm{W}(\mathtt{c'}, \mathtt{cl'}, \_)\rfloor_{r'} \wedge \\ \beta \geq lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge \overline{v} = \infty) \vee (\mathtt{v} \geq 0 \wedge \overline{v} = \mathtt{v})) \wedge v < \mathtt{e} \wedge \mathtt{v} < \mathtt{e}\end{array}\right\}$$

`lock(cl');`

$$\left\{\begin{array}{l}\exists cl, cl', v, v', \overline{v}, lay, lay', \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(\overline{lay})\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, v, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \mathtt{c'}, \overline{lay}+1)\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c'}, \mathtt{cl'}, v', \_, lay')\rfloor_{r'} * \lceil \textsc{Unlck}(v', lay')\rceil_{r'} * \\ \beta \geq lay > \overline{lay} > lay' \wedge ((\mathtt{v} = -1 \wedge \overline{v} = \infty) \vee (\mathtt{v} \geq 0 \wedge \overline{v} = \mathtt{v})) \wedge v < \mathtt{e} \wedge \mathtt{v} < \mathtt{e}\end{array}\right\}$$

`v := c'.val;`

$$\left\{\begin{array}{l}\exists cl, cl', v, v', \overline{v}, lay, lay', \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(\overline{lay})\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, \mathtt{pl}, v, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \mathtt{c'}, \overline{lay}+1)\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}+1)\rceil_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c'}, \mathtt{cl'}, v', \_, lay')\rfloor_{r'} * \lceil \textsc{Unlck}(v', lay')\rceil_{r'} * \\ \beta \geq lay > \overline{lay} > lay' \wedge ((\mathtt{v} = -1 \wedge v' = \infty) \vee (\mathtt{v} \geq 0 \wedge v' = \mathtt{v})) \wedge v < \mathtt{e}\end{array}\right\}$$

`unlock(pl);`

$$\left\{\begin{array}{l}\exists cl, cl', v, v', \overline{v}, lay, lay', \overline{lay}, S.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(\overline{lay})\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \mathtt{c'}, \overline{lay}+1)\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay}+1)\rceil_{r'} * \lfloor \mathrm{K}(\mathtt{c'}, \mathtt{cl'}, v', \_, lay')\rfloor_{r'} * \lceil \textsc{Unlck}(v', lay')\rceil_{r'} * \\ \beta \geq lay > \overline{lay} > lay' \wedge ((\mathtt{v} = -1 \wedge v' = \infty) \vee (\mathtt{v} \geq 0 \wedge v' = \mathtt{v})) \wedge v < \mathtt{e}\end{array}\right\}$$

`p := c;`

`c := c';`

$$\left\{\begin{array}{l}\exists pl, cl, v, \overline{v}, lay, \overline{lay}, S, \gamma.\ \mathbf{lcset}_r(r', \mathtt{hl}, \mathtt{x}, S) * \lfloor \textsc{Free}(lay)\rfloor_{r'} * \\ \lfloor \mathrm{K}(\mathtt{p}, pl, v, \mathtt{c}, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \lfloor \mathrm{K}(\mathtt{c}, cl, \overline{v}, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'} * \\ \gamma \geq lay > \overline{lay} \wedge ((\mathtt{v} = -1 \wedge \overline{v} = \infty) \vee (\mathtt{v} \geq 0 \wedge \overline{v} = \mathtt{v})) \wedge v < \mathtt{e} \wedge \beta > \gamma\end{array}\right\}$$

`}`

$$\left\{\begin{array}{l}\exists l, \bar{l}, v, \overline{v}, h, lay, \overline{lay}, S.\ \mathbf{lcset}_r(r', hl, \mathtt{x}, S) * \\ \lfloor \mathrm{K}(\mathtt{p}, l, v, h, lay+1)\rfloor_{r'} * \lceil \textsc{Unlck}(v, lay+1)\rceil_{r'} * \lfloor \mathrm{K}(h, \bar{l}, \overline{v}, \_, \overline{lay})\rfloor_{r'} * \lceil \textsc{Unlck}(\overline{v}, \overline{lay})\rceil_{r'} * \\ \lfloor \textsc{Free}(lay)\rfloor_{r'} \wedge lay > \overline{lay} \wedge v < e \leq v'\end{array}\right\}$$

### E.6 Queue Counter Client from [11]

In [11, Fig. 9] the authors present a program (reproduced below using our syntax) that cannot be verified using the naive restriction on their obligations/credits system. Since the restriction reads similar to our condition on layers and live obligations, we show that our system can handle this example very straightforwardly. This demonstrates that the relation between our obligations/layers and their obligations/credits is superficial.

The program consists of two threads communicating via an unbounded channel:

$$\begin{array}{c} \mathsf{x} := \mathtt{newQueue}(); \\ \mathtt{send}(\mathsf{x}, 0); \\ \mathsf{v} := \mathtt{receive}(\mathsf{x}); \\ \mathtt{send}(\mathsf{x}, \mathsf{v}+1); \end{array} \Bigg\| \begin{array}{c} \\ \\ \mathsf{v} := \mathtt{receive}(\mathsf{x}); \\ \mathtt{send}(\mathsf{x}, \mathsf{v}+1); \end{array}$$

We give specification to the channel primitives as follows:

$$\bot \vdash \{\mathsf{emp}\}\ \mathtt{newQueue}()\ \{\mathrm{Q}(\mathrm{ret}, \epsilon)\}$$

$$\top \vdash \mathbb{V} ns \in \mathbb{N}^*.\langle \mathrm{Q}(s, \mathsf{x}, ns)\rangle\ \mathtt{send}(\mathsf{x}, \mathsf{v})\ \langle \mathrm{Q}(s, \mathsf{x}, ns') \wedge ns = n \oplus \mathsf{v}\rangle$$

$$\top \vdash \mathbb{V} ns \in \mathbb{N}^* \twoheadrightarrow_\bot \mathbb{N}^+.\langle \mathrm{Q}(s, \mathsf{x}, ns)\rangle\ \mathtt{receive}(\mathsf{x})\ \langle \exists ns' \in \mathbb{N}^*.\, \mathrm{Q}(s, \mathsf{x}, ns') \wedge ns = \mathtt{ret} : ns'\rangle$$

where $\mathrm{Q}(s, x, ns)$ represents a queue at $x$ with contents $ns$.

For the verification, we create a region **client** for sharing the channel. Since the intent of the program seems to be to store a natural number as the contents of the channel, the abstract state of the region keeps track of this number $n$.

$$I(\mathbf{client}_r(s, x, n)) \triangleq ((\mathrm{Q}(x, [n]) * \lfloor \mathrm{U} \rfloor_r * \lceil \mathrm{O} \rceil_r) \vee (\mathrm{Q}(x, \epsilon) * \lfloor \mathrm{L}(n) \rfloor_r)) * \lfloor \mathrm{V}(n) \rfloor_r$$

**Interference:** $\mathrm{C}(m, \pi) : n \rightsquigarrow n + 1$

**Guard algebra:**

$$\mathrm{V}(n) \bullet \mathrm{C}(m, \pi) \text{ is defined.} \iff n \geq m \wedge \pi = 1 \Rightarrow n = m$$

$$\mathrm{C}(m_1 + m_2, \pi_1 + \pi_2) = \mathrm{C}(m_1, \pi_1) \bullet \mathrm{C}(m_2, \pi_2) \qquad m_1, m_2 \geq 0, \pi_1, \pi_2 > 0$$

$$\mathrm{C}(m, \pi) \bullet \mathrm{V}(n) = \mathrm{C}(m + d, \pi) \bullet \mathrm{V}(n + d) \qquad d \geq 0$$

$$\mathrm{U} = \mathrm{L}(n) \bullet \mathrm{K}(n)$$

$$\mathrm{U} \bullet L(n) \text{ is undefined.}$$

$$\mathrm{L}(n) \bullet \mathrm{K}(m) \text{ is defined.} \iff n = m$$

Total: O We are able to prove termination, and that the end value is 2.

$$\{\text{emp}\}$$
$$\texttt{x := newQueue();}$$
$$\{\text{Q}(\texttt{x}, \epsilon)\}$$
$$\texttt{send(x, 0);}$$
$$\{\text{Q}(\texttt{x}, [0])\}$$
$$\{\textbf{client}_r(\texttt{x}, 0) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(0,1) \rfloor_r\}$$

| | |
|---|---|
| $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(0, ½) \rfloor_r\}$ | $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(0, ½) \rfloor_r\}$ |
| $\texttt{v := receive(x);}$ | $\texttt{v := receive(x);}$ |
| $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(0, ½) \rfloor_r * \lfloor \text{K}(\texttt{v}) \rfloor_r * \lceil \text{O} \rceil_r\}$ | $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(0, ½) \rfloor_r * \lfloor \text{K}(\texttt{v}) \rfloor_r * \lceil \text{O} \rceil_r\}$ |
| $\texttt{send(x, v + 1);}$ | $\texttt{send(x, v + 1);}$ |
| $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(1, ½) \rfloor_r\}$ | $\{\exists n.\, \textbf{client}_r(\texttt{x}, n) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(1, ½) \rfloor_r\}$ |

$$\{\exists n.\, \textbf{client}_r(\texttt{x}, 2) * \lfloor \text{G} \rfloor_r * \lfloor \text{C}(2,1) \rfloor_r\}$$

### E.7 Proof of Example 2.1

We produce here a formal version of the proof presented in Section 2:

```
lock(x);          ||  while(!d){ // d false initially
[done] := true;   ||    lock(x); d := [done]; unlock(x);
unlock(x);        ||  }
```

where the locks are assumed to be fair.

$I(\mathbf{c}_r(x, done, l, d)) \triangleq \exists s.\, \text{L}(s, x, l) * done \mapsto d * ((\lceil \text{U} \rceil_r \wedge l = 0) \vee (\lceil \text{L} \rceil_r \wedge l = 1)) * ((\lceil \text{D} \rceil_r \wedge d) \vee (\lceil \text{W} \rceil_r \wedge \neg d))$

**Interference:**

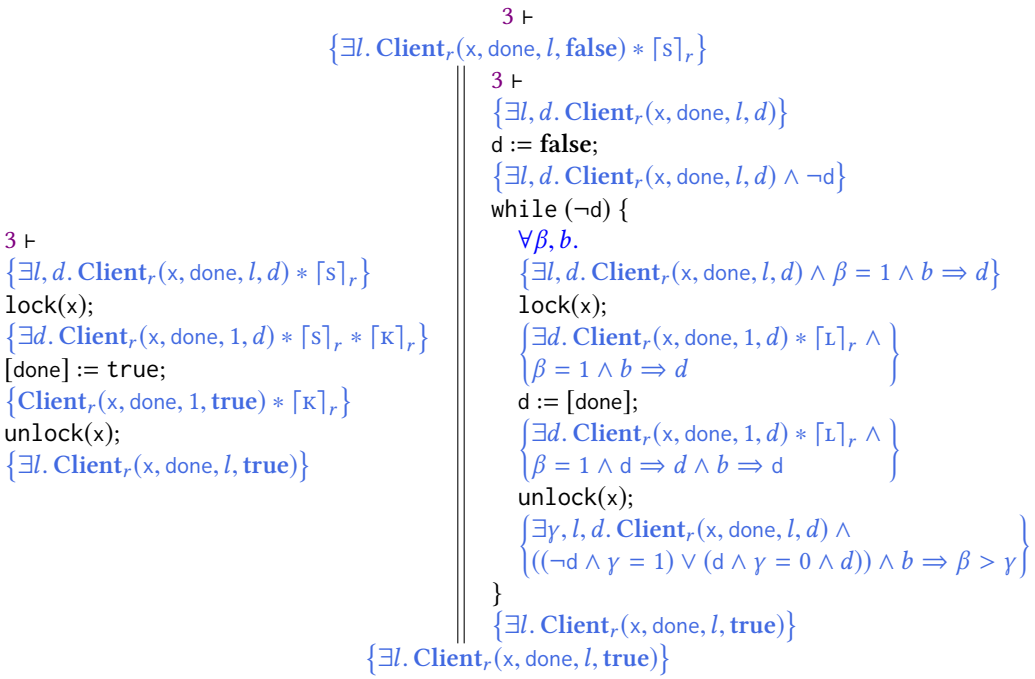
$$\text{S} : (l, d) \rightsquigarrow (l, \texttt{true})$$
$$\mathbf{0} : (0, d) \rightsquigarrow (1, d)$$
$$\mathbf{0} : (1, d) \rightsquigarrow (0, d)$$

Obligation algebra:

$$\text{U} = \text{L} \bullet \text{K} \qquad\qquad \text{D} = \text{W} \bullet \text{S}$$

With layers: $\text{lay}(\text{U}) = \text{lay}(\text{D}) = \text{lay}(\text{L}) = \text{lay}(\text{W}) = 0$, $\text{lay}(\text{K}) = 1$, $\text{lay}(\text{S}) = 2$.

Total: $\text{U} \bullet \text{D}$




## E.8 MCS Lock

```
lock(x) {
  c := alloc(2);
  [c] := true;
  [c + 1] := null;
  p := FAS(x + 1, c);
  if (p ≠ null) {
    [p + 1] := c;
    v := [c];
    while(v) {
      v := [c];
    }
  } else {
    [c] := false;
  }
  [x] := c;
}
```

```
unlock(x) {
  h := [x];
  n := [h + 1];
  if (n = null) {
    b := CAS(x + 1, h, null);
    if(!b) {
      while(n = null) {
        n := [h + 1];
      }
      [n] := false;
    }
  } else {
    [n] := false;
  }
}
```

$$\top \vdash \forall l \in \{0,1\} \twoheadrightarrow_{\perp} \{0\}, o \in \mathbb{N}. \langle \mathsf{L}(s, \mathsf{x}, l, o) \rangle \ \mathsf{lock(x)} \ \langle \mathsf{L}(s, \mathsf{x}, 1, o+1) \wedge l = 0 \rangle$$
$$\perp \vdash \langle \mathsf{L}(s, \mathsf{x}, 1, o) \rangle \ \mathsf{unlock(x)} \ \langle \mathsf{L}(s, \mathsf{x}, 0, o) \rangle$$

$$\mathsf{L}((r, r'), x, l, o) \triangleq \exists \overline{hd} \in \mathrm{Addr}. \ \mathbf{mcs}_r(r', x, l, o, \overline{hd}) * \lfloor \mathsf{G} \rfloor_r$$

$$\mathsf{G} : \forall o \in \mathbb{N}, hd, hd' \in \mathrm{Addr}. (0, o, hd) \rightsquigarrow (1, o+1, hd') \quad \mathsf{G} : \forall o \in \mathbb{N}, hd \in \mathrm{Addr}. (1, o, hd) \rightsquigarrow (0, o, hd)$$

$$I(\mathbf{mcs}_r(r', x, l, o, \overline{hd})) \triangleq \exists \overline{hd'} \in \text{Addr}.\, \mathbf{mcsh}_{r'}(x, l, o, \overline{hd}, \overline{hd'}) * \lfloor \text{G'} \rfloor_{r'}$$

$$I(\mathbf{mcsh}_r(x, l, o, \overline{hd}, \overline{hd'})) \triangleq \exists \overline{lst} \in \text{Addr}.\, x \mapsto \overline{hd}, \overline{lst} * \text{True} *$$
$$\left( \begin{array}{c} (\overline{lst} = \mathbf{null} \wedge l = 0 * [\textsc{EmptyQueue}(o)]_r) \vee \\ \left( \begin{array}{c} \exists t \in \mathbb{N}, d \in \mathbb{B}, hd, hd' \in \text{Addr}.\, [\textsc{State}(o,t)]_r * hd \mapsto d, \overline{hd'} * queue(hd', o+1, t) \wedge o < t * \\ \left( \begin{array}{c} (l = 1 \wedge hd = \overline{hd} \wedge \neg d \wedge \lfloor \textsc{Head}(\overline{hd}) \rfloor_r * \lfloor \textsc{Ownership}(\overline{hd}) \rfloor_r) \vee \\ (l = 0 \wedge hd \neq \overline{hd} \wedge ((d \wedge \lfloor \textsc{First}(\overline{hd}) \rfloor_r) \vee (\neg d \wedge \lfloor \textsc{Head}(\overline{hd}) \rfloor_r))) \end{array} \right) * \\ ((\overline{hd'} \doteq \mathbf{null}) \vee (hd' = \overline{hd'} \wedge \lceil \textsc{Update}(o, hd, hd') \rceil_r * \lfloor \textsc{Next}(o, h') \rfloor_r)) \end{array} \right) \end{array} \right)$$

$$\text{queue}(h, o, t) \triangleq \exists h', \overline{h'} \in \text{Addr}.\, h \mapsto \text{True}, \overline{h'} * queue(h', o+1, t) * ((\overline{h'} \doteq \mathbf{null}) \vee (h' = \overline{h'} \wedge \lceil \textsc{Update}(o, h, h') \rceil_r * \lfloor \textsc{Next}(o, h') \rfloor_r)) * \lfloor \textsc{Node}(o, h) \rfloor_r \quad o + 1 < t$$
$$\text{queue}(h, o, o+1) \triangleq h \mapsto \text{True}, \mathbf{null} * \lfloor \textsc{Node}(o, h) \rfloor_r$$
$$\text{queue}(h, o, o) \triangleq h \mapsto \text{emp} \wedge h = \mathbf{null}$$

$$\text{G} : \forall o \in \mathbb{N}, \overline{hd}, \overline{hd'}, \overline{hd''} \in \text{Addr}.\, (0, o, \overline{hd}, \overline{hd'}) \rightsquigarrow (1, o+1, \overline{hd'}, \overline{hd''}) \quad \text{G} : \forall o \in \mathbb{N}, hd \in \text{Addr}.\, (1, o, \overline{hd}, \overline{hd'}) \rightsquigarrow (0, o, \overline{hd}, \overline{hd'})$$
$$\text{G} : \forall l \in \{0, 1\}, o \in \mathbb{N}, \overline{hd} \in \text{Addr}, \overline{hd'} \in \text{Addr}.\, (l, o, \overline{hd}, \mathbf{null}) \rightsquigarrow (l, o, \overline{hd}, \overline{hd'})$$

Guard algebra:

$$\textsc{EmptyQueue}(o) = \textsc{State}(o, o)$$

$$\textsc{State}(o, o) = \textsc{State}(o, o+1) \bullet \textsc{SelfSignal}(h) \bullet \textsc{First}(h)$$
$$\textsc{SelfSignal}(h) \bullet \textsc{First}(h) = \textsc{Ownership}(h) \bullet \textsc{Head}(h)$$

$$\textsc{State}(o, o+1) \bullet \textsc{First}(p) = \textsc{State}(o, o+2) \bullet \textsc{OtherSignal}(o+1, p, h) \bullet \textsc{Node}(o+1, h) \bullet \textsc{First}(p)$$
$$\textsc{State}(o, o+1) \bullet \textsc{Head}(p) = \textsc{State}(o, o+2) \bullet \textsc{OtherSignal}(o+1, p, h) \bullet \textsc{Node}(o+1, h) \bullet \textsc{Head}(p)$$
$$\textsc{State}(o, t+1) \bullet \textsc{Node}(t, p) = \textsc{State}(o, t+2) \bullet \textsc{OtherSignal}(t+1, p, h) \bullet \textsc{Node}(t+1, h) \bullet \textsc{Next}(t, c) \bullet \textsc{Node}(t, p) \qquad o < t$$
$$\textsc{OtherSignal}(t, p, h) \bullet \textsc{Head}(h) = \textsc{Ownership}(h) \bullet \textsc{Head}(h)$$
$$\textsc{Next}(t, p) = \textsc{Next}(t, p) \bullet \textsc{SignalWitness}(t, p)$$

$$\textsc{State}(o, t) \bullet \textsc{Node}(o+1, h) \bullet \textsc{Ownership}(\_) \bullet \textsc{Head}(\_) = \textsc{State}(o+1, t) \bullet \textsc{Head}(h)$$
$$\textsc{State}(o, t) = \textsc{State}(o, t) \bullet \textsc{StateWitness}(o, t)$$
$$\textsc{State}(o, t) \bullet \textsc{StateWitness}(o', t') \text{ is defined.} \iff o \geq o' \wedge t \geq t'$$

$$\textsc{State}(o, t) \text{ is defined.} \iff o \leq t$$

$$\textsc{EmptyQueue}(\_) \bullet \textsc{SelfSignal}(\_) \text{ is undefined.}$$
$$\textsc{Head}(\_) \bullet \textsc{SelfSignal}(\_) \text{ is undefined.}$$
$$\textsc{First}(h) \bullet \textsc{SelfSignal}(h') \text{ is defined.} \iff h = h'$$

$$\textsc{EmptyQueue}(\_) \bullet \textsc{Ownership}(\_) \text{ is undefined.}$$
$$\textsc{First}(\_) \bullet \textsc{Ownership}(\_) \text{ is undefined.}$$
$$\textsc{Head}(h) \bullet \textsc{Ownership}(h') \text{ is defined.} \iff h = h'$$
$$\textsc{Ownership}(\_) \bullet \textsc{Ownership}(\_) \text{ is un1defined.}$$

$$\textsc{EmptyQueue}(\_) \bullet \textsc{OtherSignal}(\_, \_, \_) \text{ is undefined.}$$
$$\textsc{State}(o, t) \bullet \textsc{OtherSignal}(t', p, h) \text{ is defined.} \iff o \leq t' < t$$

$$\textsc{Next}(t, p) \bullet \textsc{SignalWitness}(t', p') \text{ is defined.} \iff (t = t' \Rightarrow p = p')$$

Obligation algebra:

$$\textsc{EmptyQueue}(\_) = \textsc{EmptyQueue}(0)$$
$$\textsc{EmptyQueue}(o) = \textsc{State}(o, o)$$
$$\textsc{State}(o, o) = \textsc{State}(o, o+1) \bullet \textsc{Ticket}(o, h) \quad o < t$$
$$\textsc{State}(o, t) = \textsc{State}(o, t+1) \bullet \textsc{Update}(t, h, h') \bullet \textsc{Ticket}(t, h') \quad o < t$$
$$\textsc{State}(o, t) \bullet \textsc{Update}(o, \_, h') \bullet \textsc{Ticket}(o, h') = \textsc{State}(o+1, t)$$

$$\text{lay}(\text{T}(t, \_)) = t + 1$$
$$\text{lay}(\_) = 0$$

Total: $\textsc{EmptyQueue}(0)$


⊤ ⊢
∀l ∈ {0,1} ↠⊥ {0}, o ∈ ℕ.
⟨L(s, x, l, o)⟩
∀hd ∈ Addr.
⟨mcs_r(x, l, o, hd) ∗ ⌊G⌋_r⟩
⊤; r : ∀l ∈ {0,1}, o ∈ ℕ, hd, hd′ ∈ Addr. (l, o, hd) ↠⊥ (0, o, hd) ⟶ (1, o, hd′) ∧ l = 0 ⊢
{∃l ∈ {0,1}, o ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦}
c := alloc(2);
[c] := true;
[c + 1] := null;
{∃l ∈ {0,1}, o ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗ c ↦ True, null}
p := FAS(x + 1, c);
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗ ⌈Ticket(t′, c)⌉_r′ ∗
((p = null ∧ ⌊SelfSignal⌋_r′) ∨ (p ≠ null ∧ ⌊OtherSignal(t′, p, c)⌋_r′ ∗ ⌈Update(t′, p, c)⌉_r′)) ∧ o ≤ t′}
∀t′. t′ + 1 ⊢
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗
((p = null ∧ ⌊SelfSignal⌋_r′) ∨ (p ≠ null ∧ ⌊OtherSignal(t′, p, c)⌋_r′ ∗ ⌈Update(t′, p, c)⌉_r′)) ∧ o ≤ t′}
if (p ≠ null) {
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗
⌊OtherSignal(t′, p, c)⌋_r′ ∗ ⌈Update(t′, p, c)⌉_r′ ∧ o ≤ t′}
[p + 1] := c;
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗ ⌊OtherSignal(t′, p, c)⌋_r′ ∧ o ≤ t′}
v := [c];
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗
(((⌊OtherSignal(t′, p, c)⌋_r′ ∧ v ∧ o ≤ t′) ∨ (⌊Ownership(c)⌋_r′ ∧ ¬v ∧ l = 0))}
while (v) {
∀b ∈ 𝔹, β. ⊢
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗
⌊OtherSignal(t′, p, c)⌋_r′ ∧ v ∧ o ≤ t′ ∧ β = 1 ∧ b ⇒ o = t′}
v := [c];
{∃l ∈ {0,1}, o, t′ ∈ ℕ, hd ∈ Addr, γ. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗
(((⌊OtherSignal(t′, p, c)⌋_r′ ∧ v ∧ o ≤ t′ ∧ γ = 1) ∨ (⌊Ownership(c)⌋_r′ ∧ ¬v ∧ l = 0 ∧ γ = 0)) ∧ b ⇒ γ < β}
}
{∃o ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, 0, o, hd) ∗ r ⇛ ♦ ∗ ⌊Ownership(c)⌋_r′}
} else {
{∃l ∈ {0,1}, o ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o, hd) ∗ r ⇛ ♦ ∗ ⌊SelfSignal(c)⌋_r′}
[c] := False;
{∃o ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, 0, o, hd) ∗ r ⇛ ♦ ∗ ⌊Ownership(c)⌋_r′}
}
{∃o, t′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, 0, o, hd) ∗ r ⇛ ♦ ∗ ⌊Ownership(c)⌋_r′ ∗ ⌈T(t′, c)⌉_r′}
[x] := c;
{∃l, l′ ∈ {0,1}, o, o′ ∈ ℕ, hd, hd′ ∈ Addr. mcs_r(r′, x, l′, o′, hd′) ∗ r ⇛ ((l, o, hd), (1, o, c)) ∧ l = 0}
⟨mcs_r(r′, x, 1, o) ∗ ⌊G⌋_r ∧ l = 0⟩
⟨L(s, x, 1, o) ∧ l = 0⟩
Cons; Sub s = (r, r′); A∃Elim
Make Atomic
QL; Cons; Frame: ∃t′ ∈ ℕ. ⌈Ticket(t′, c)⌉_r


⊤ ⊢
⟨L(s, x, 1, o)⟩
Cons; Sub s = r; A∃Elim
∀hd ∈ Addr.
⟨mcs_r(r′, x, l, o, hd) ∗ ⌊G⌋_r⟩
Make Atomic
⊤; r : ∀l ∈ {0,1}, o ∈ ℕ, hd ∈ Addr. (1, o, hd) ⟶ (0, o + 1, hd) ⊢
{∃hd ∈ Addr. mcs_r(r′, x, 1, o, hd) ∗ r ⟾ ♦}
h := [x];
{mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦}
n := [h + 1];
{mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦ ∗ (n ≐ null ∨ (n ≠ null ∧ ⌊SignalWitness(o, n)⌋_r′))}
if (n = null) {
{mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦}
b := CAS(x + 1, h, null);
{∃t ∈ ℕ. mcs_r(r′, x, 1, o, h) ∗ ((b = 1 ∧ r ⟾ ((1, o, h), (0, o + 1, h))) ∨ (b = 0 ∧ o + 1 < t ∧ r ⟾ ♦ ∗ ⌊StateWitness(o, t)⌋_r′))}
if (b = 0) {
{∃t ∈ ℕ. mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦ ∗ ⌊StateWitness(o, t)⌋_r′ ∧ o + 1 < t}
Frame; ∃Elim; Open Region; A∃Elim
∀hd′ ∈ Addr.
⟨mcsh_r′(x, 1, o, h, hd′) ∗ ⌊StateWitness(o, t)⌋_r′ ∧ o + 1 < t⟩
Make Atomic
{∃hd′ ∈ Addr. mcsh_r′(x, 1, o, h, hd′) ∗ ⌊StateWitness(o, t)⌋_r′ ∗ r′ ⟾ ♦ ∧ o + 1 < t}
while (n = null) {
∀b ∈ 𝔹, β. ⊢
{∃hd′ ∈ Addr. mcsh_r′(x, 1, o, h, hd′) ∗ ⌊StateWitness(o, t)⌋_r′ ∗ r′ ⟾ ♦ ∧ o + 1 < t ∧ n = null ∧ β = 1 ∧ b ⇒ hd′ ≠ null}
n := [h + 1];
{∃hd′ ∈ Addr, γ. mcsh_r′(x, 1, o, h, hd′) ∗ ((⌊StateWitness(o, t)⌋_r′ ∗ r′ ⟾ ♦ ∧ o + 1 < t ∧ n = null ∧ γ = 1) ∨ (⌊SignalWitness(o, n)⌋_r′ ∗ r′ ⟾ ((1, o, h, n), (1, o, h, n)) ∧ n ≠ null ∧ n = hd′ ∧ γ = 0)) ∧ b ⇒ γ < β}
}
{mcsh_r′(x, 1, o, h, n) ∗ ⌊SignalWitness(o, n)⌋_r′ ∗ r′ ⟾ ((1, o, h, n), (1, o, h, n)) ∧ n ≠ null}
⟨∃t ∈ ℕ. mcsh_r′(x, 1, o, h, n) ∗ ⌊SignalWitness(o, n)⌋_r′ ∧ n ≠ null⟩
{mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦ ∗ ⌊SignalWitness(o, n)⌋_r′}
[n] := False;
{∃l ∈ {0,1}, o′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o′, hd) ∗ r ⟾ ((1, o, h), (0, o + 1, h))}
}
{∃l ∈ {0,1}, o′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o′, hd) ∗ r ⟾ ((1, o, h), (0, o + 1, h))}
} else {
{mcs_r(r′, x, 1, o, h) ∗ r ⟾ ♦ ∗ ⌊SignalWitness(o, n)⌋_r′}
[n] := False;
{∃l ∈ {0,1}, o′ ∈ ℕ, hd ∈ Addr. mcs_r(r′, x, l, o′, hd) ∗ r ⟾ ((1, o, h), (0, o + 1, h))}
}
{∃l, l′ ∈ {0,1}, o, o′ ∈ ℕ, hd, hd′ ∈ Addr. mcs_r(r′, x, l′, o′, hd′) ∗ r ⟾ ((l, o, hd), (1, o, c)) ∧ l = 0}
⟨mcs_r(r′, x, 1, o) ∗ ⌊G⌋_r ∧ l = 0⟩
⟨L(s, x, 1, o) ∧ l = 0⟩

### E.9 Optimistic Set

```
validate(x, p, c) {
  prev := x;
  n := [prev + 2];
  nk := [n];
  pk := [p];
  while(nk < pk) {
    prev := n;
    n := [n + 2];
    nk := [n];
  }
  return ((n = c) && (p = prev));
}
```

```
add(x, k) {
  b := true;
  while(b) {
    c := x;
    ck := [c]
    while(ck < k) {
      p := c;
      c := [c + 2];
      ck := [c];
    }
    pl := [p + 1];
    lock(pl);
    cl := [c + 1];
    lock(cl);
    valid := validate(x, p, c);
    if(valid) {
      if(ck = k) {
        added := false;
      } else {
        nn := alloc(3);
        [nn] := k;
        nnl := newLock();
        [nn + 1] := nnl;
        [nn + 2] := c;
        [p + 2] := nn;
        added := true;
      }
      b := false;
    }
    unlock(cl);
    unlock(pl);
  }
  return added;
}
```

```
remove(x) {
  b := true;
  while(b) {
    c := x;
    ck := [c]
    while(ck < k) {
      p := c;
      c := [c + 2];
      ck := [c];
    }
    pl := [p + 1];
    lock(pl);
    cl := [c + 1];
    lock(cl);
    valid := validate(x, p, c);
    if(valid) {
      if(ck = k) {
        n := [c + 2];
        [p + 2] := n;
        removed := true;
      } else {
        removed := false;
      }
      b := false;
    }
    unlock(cl);
    unlock(pl);
  }
  return removed;
}
```

$\forall\phi.\ \top \vdash \mathbb{W}S \in \mathcal{P}(\mathbb{Z}), \alpha.\left\langle \mathrm{OptSet}(s, \mathsf{x}, S, \alpha) \wedge \phi(\alpha) < \alpha \right\rangle\ \mathsf{add}(\mathsf{x}, \mathsf{k})\ \left\langle \mathrm{OptSet}(s, \mathsf{x}, S \cup \{\mathsf{k}\}, \phi(\alpha)) \wedge (\mathrm{ret} \iff \mathsf{k} \notin S) \right\rangle$
$\forall\phi.\ \top \vdash \mathbb{W}S \in \mathcal{P}(\mathbb{Z}), \alpha.\left\langle \mathrm{OptSet}(s, \mathsf{x}, S, \alpha) \wedge \phi(\alpha) < \alpha \right\rangle\ \mathtt{remove}(\mathsf{x}, \mathsf{k})\ \left\langle \mathrm{OptSet}(s, \mathsf{x}, S \setminus \{\mathsf{k}\}, \phi(\alpha)) \wedge (\mathrm{ret} \iff \mathsf{k} \in S) \right\rangle$

Just like the lock-coupling list set, the optimistic set data structure consists of a linked list, with each node containing an element (the key) of the set in increasing order and a reference to a lock. Similarly, the key and lock reference associated with each node is fixed once added to the list, and the reference to the next element in the linked list can only be changed by a thread holding the associated lock.

However, rather than performing a hand-over-hand locking routine, the `add` and `remove` operations optimistically traverse the list without obtaining any locks. Thus, it may reach nodes that have been removed from the list, or miss nodes that are added in concurrently. Once the optimistic traversal believes it has reached the nodes to act on, it locks them and then runs a verification procedure to check they are indeed in the linked list. However,

to guarantee the termination of the traversal process, we need to guarantee that there are only a finite number of concurrent add and remove operations, so the specifications will need an ordinal to limit impedence.

Due to the posibility of traversing nodes that are no longer in the linked list, the data structure is better thought of as a Directed Acyclic Graph(DAG) of degree 1 with the property that the keys associated with the nodes increase as we follow the links in the DAG. Furthermore, this DAG, contains two special nodes, the head node, with value $-\infty$ and, the end node, with value $\infty$. These nodes have special properties: the value of every node, except for the head and end node, reachable from the head, is considered in the abstract set the data structure represents, and the end node is reachable from every node in the DAG.

To abstractly represent this DAG, we use the type $\mathsf{AOptList} \triangleq \mathsf{Addr} \rightharpoonup \mathsf{Val} \times \mathsf{Addr} \times \{0,1\} \times \mathsf{Addr} \times \mathbb{O} \times \mathbb{O}$, where $\mathbb{O}$ is the set of ordinals parametrising the logic and $\mathsf{Val} = \mathbb{Z} \cup \{-\infty, \infty\}$. This mapping from the addresses of a node to a tuple of the node's value, a lock pointer, the state of said lock, a pointer to the next node (or null for the end node) and two ordinals used for assigning layers to the locks will be used to abstractly represent the DAG. To allow nested locking toward the end node, the layers of the associated locks must decrease going toward the end node. The first ordinal in a cell's state, $\beta$, is the associated lock's layer, and the second, $\gamma$, is a "gap". No other node in the DAG can have a layer between $\beta$, the node's layer, and $\beta \oplus \gamma$, where $\oplus$ represents the Hessenberg sum of ordinals. This gap allows us to preserve gaps in the layer structure for the addition of future nodes. It is an invariant that given an ordinal $\alpha$ limiting the impedence on the module, the gap associated with each node, $\gamma$, must be such that $\gamma \geq 2^\alpha$, allowing each gap to be subdivided each time $\alpha$ is decreased.

AOptList will parametrize a guard (and obligation), $\textsc{State}(h, e, C, Out)$, where $h, e \in \mathsf{Addr}$, are the head and end nodes, $C \in \mathsf{AOptList}$, is the representation of the DAG and $Out \in \mathcal{P}(\mathsf{Addr})$, is the set of nodes which are not reacheable from the head node. The following invariants represent the informal ones above:

- $\forall c, c' \in \mathsf{Addr}, k \in \mathsf{Val}, \beta. \left( \begin{array}{c} C(c) = (k, \_, \_, c', \beta, \_) \wedge \\ k \neq \infty \end{array} \right) \Rightarrow \left( \begin{array}{c} \exists k' \in \mathsf{Val}, \beta', \gamma'. c' \neq \mathtt{null} \wedge \\ C(c') = (k', \_, \_, \_, \beta, \gamma) \wedge \\ k' > k \wedge \beta \geq \beta' \oplus \gamma' \end{array} \right)$
- $Out \subseteq \mathrm{dom}(C)$
- $e \notin Out$
- $\forall c \in \mathrm{dom}(C). c \in Out \iff \neg\, \mathrm{Reach}(h, C, c)$
- $C(h) = (-\infty, \_, \_.\_, \_, 0)$
- $C(e) = (\infty, \_, \_, \mathtt{null}, 0, \_)$

Where the inductive predicate $\mathrm{Reach}(h, C, c)$ is defined as:

$$\mathrm{Reach}(h, C, c) \triangleq (h = c) \vee (\exists h' \in \mathsf{Addr}. C(h) = (\_, \_, \_, h', \_, \_) \wedge \mathrm{Reach}(h', C, c))$$

This DAG is abstractly represented as $CS \in \mathcal{P}(\mathsf{Addr} \times \mathsf{Val} \times \{0,1\} \times \mathsf{Addr})$ where $\forall (c, k, l, c') \in \mathsf{Addr} \times \mathsf{Val} \times \mathsf{Addr}. (c, k, c') \in CS \iff C(c) = (k, \_, l, c', \_, \_)$. The add and remove operations are represented with the following guard labeled transitions over this abstraction:

$$\text{G' : } \forall CS \in \mathcal{P}(\mathsf{Addr} \times \mathsf{Val} \times \{0,1\} \times \mathsf{Addr}), Out \in \mathcal{P}(\mathsf{Addr}), c, c', c'' \in \mathsf{Addr}, k, k' \in \mathbb{Z}, l \in \{0,1\}, \alpha, \beta < \alpha.$$
$$(CS \uplus \{(c, k, 1, c'), (c', k', 1, c'')\}, Out, \alpha) \rightsquigarrow$$
$$(CS \uplus \{(c, k, 1, \overline{c}), (\overline{c}, \overline{k}, 0, c'), (c', k', 1, c'')\}, Out, \beta) \wedge c \notin Out \wedge k < \overline{k} < k'$$

$$\text{G' : } \forall CS \in \mathcal{P}(\mathsf{Addr} \times \mathsf{Val} \times \{0,1\} \times \mathsf{Addr}), Out \in \mathcal{P}(\mathsf{Addr}), c, \overline{c}, c' \in \mathsf{Addr}, k, \overline{k} \in \mathbb{Z}, \alpha, \beta < \alpha.$$
$$(CS \uplus \{(c, k, 1, \overline{c}), (\overline{c}, \overline{k}, 1, c')\}, Out, \alpha) \rightsquigarrow$$
$$(CS \uplus \{(c, k, 1, c'), (\overline{c}, \overline{k}, 1, c')\}, Out \uplus \{\overline{c}\}, \beta) \wedge \overline{c} \notin Out$$

and these for locking and unlocking nodes:

$$\text{G' : } \forall CS \in \mathcal{P}(\mathsf{Addr} \times \mathsf{Val} \times \{0,1\} \times \mathsf{Addr}), Out \in \mathcal{P}(\mathsf{Addr}), c \in \mathsf{Addr}, k \in \mathbb{Z}, l \in \{0,1\}, \alpha, \beta < \alpha.$$
$$(CS \uplus \{(c, k, 0, c')\}, Out, \alpha) \rightsquigarrow$$
$$(CS \uplus \{(c, k, 1, c')\}, Out, \alpha)$$

$$\text{G' : } \forall CS \in \mathcal{P}(\mathsf{Addr} \times \mathsf{Val} \times \{0,1\} \times \mathsf{Addr}), Out \in \mathcal{P}(\mathsf{Addr}), c \in \mathsf{Addr}, k \in \mathbb{Z}, l \in \{0,1\}, \alpha, \beta < \alpha.$$
$$(CS \uplus \{(c, k, 1, c')\}, Out, \alpha) \rightsquigarrow$$
$$(CS \uplus \{(c, k, 0, c')\}, Out, \alpha)$$

Here $Out$ is as above and $\alpha$ is an ordinal, bounding the number of updates to the data structure. We also defined this auxiliary function over $CS$:

$$\mathrm{ReachA}(h, CS, c) \triangleq (h = c) \vee (\exists h' \in \mathsf{Addr}.\, (h, \_, \_, h') \in CS \wedge \mathrm{ReachA}(h', CS, c))$$

The State guard (and obligation) must have algebraic rules allowing these updates:

$$\textsc{State}(h, l, C[c \to (k, la, 1, c', \beta, \gamma), c' \to (k', la', 1, c'', \beta', \gamma')], Out) \bullet \textsc{KIn}(c, k, la, c', \beta, \gamma) \bullet \textsc{KIn}(c', k', la', c'', \beta', \gamma') =$$
$$\textsc{State}(h, l, C[c \to (k, la, 1, \overline{c}, \beta, \gamma), \overline{c} \to (\overline{k}, \overline{la}, 0, c', \overline{\beta}, \overline{\gamma}), c' \to (k', la', 1, c'', \beta', \gamma^\dagger)], Out) \bullet$$
$$\textsc{KIn}(c, k, la, \overline{c}, \beta, \gamma) \bullet \textsc{UIn}(\overline{c}, \overline{k}, \overline{la}, c', \overline{\beta}, \overline{\gamma}) \bullet \textsc{KIn}(c', k', la', c'', \beta', \gamma^\dagger)$$

where $k < \overline{k} < k'$, $\beta \geq \overline{\beta} \oplus \overline{\gamma}$ and $\overline{\beta} \geq \beta' \oplus \gamma^\dagger$.

$$\textsc{State}(h, l, C[c \to (k, la, 1, c', \beta, \gamma), c' \to (k', la', 1, c'', \beta', \gamma')], Out) \bullet \textsc{KIn}(c, k, la, c', \beta, \gamma) \bullet \textsc{UIn}(c', k', la', c'', \beta', \gamma') =$$
$$\textsc{State}(h, l, C[c \to (k, la, 1, c'', \beta, \gamma), c' \to (k', la', 1, c'', \beta', \gamma')], Out \uplus \{c'\}) \bullet \textsc{KIn}(c, k, la, c'', \beta, \gamma) \bullet \textsc{UOut}(c', k', la', c'', \beta', \gamma')$$

where $c'' \neq \texttt{null}$. Note that both of these updates preserve all of the invariants required by the State guard. Here the UIn, LIn and KIn and UOut, LOut and KOut obligations have similar function to the U, L and K in appendix E.7 with the extra connotation of being reacheable from the head node or not, respectively. This can be expressed with the following algebraic laws:

$$\textsc{UOut}(c,k,ls,c',\beta,\gamma) = \textsc{KOut}(c,k,ls,c',\beta) \bullet \textsc{LOut}(c,k,ls,c',\beta,\gamma)$$
$$\textsc{UIn}(c,k,ls,c',\beta,\gamma) = \textsc{KIn}(c,k,ls,c',\alpha) \bullet \textsc{LIn}(c,k,ls,c',\beta,\gamma)$$
$$\textsc{UIn}(c,\_,\_,\_,\_,\_) \bullet \textsc{KIn}(c,\_,\_,\_,\_,\_) \text{ is undefined.}$$
$$\textsc{UOut}(c,\_,\_,\_,\_,\_) \bullet \textsc{KOut}(c,\_,\_,\_,\_,\_) \text{ is undefined.}$$
$$\textsc{UIn}(c,\_,\_,\_,\_,\_) \bullet \textsc{KOut}(c,\_,\_,\_,\_,\_) \text{ is undefined.}$$
$$\textsc{UOut}(c,\_,\_,\_,\_,\_) \bullet \textsc{KIn}(c,\_,\_,\_,\_,\_) \text{ is undefined.}$$
$$\textsc{KIn}(c,k,ls,c',\beta,\gamma) \bullet \textsc{LIn}(c,\overline{k},\overline{ls},\overline{c'},\overline{\beta},\overline{\gamma}) \text{ is defined.} \Leftrightarrow k = \overline{k} \wedge ls = \overline{ls} \wedge c' = \overline{c'} \wedge \beta = \overline{\beta} \wedge \gamma = \overline{\gamma}$$
$$\textsc{KOut}(c,k,ls,c',\beta,\gamma) \bullet \textsc{LOut}(c,\overline{k},\overline{ls},\overline{c'},\overline{\beta},\overline{\gamma}) \text{ is defined.} \Leftrightarrow k = \overline{k} \wedge ls = \overline{ls} \wedge c' = \overline{c'} \wedge \beta = \overline{\beta} \wedge \gamma = \overline{\gamma}$$
$$\textsc{State}(h,l,C,Out) \bullet \textsc{KIn}(c,k,la,c',\beta,\gamma) \text{ is defined.} \Leftrightarrow C(c) = (k,la,\_,c',\beta,\gamma) \wedge c \notin Out$$
$$\textsc{State}(h,l,C,Out) \bullet \textsc{KOut}(c,k,la,c',\beta,\gamma) \text{ is defined.} \Leftrightarrow C(c) = (k,la,\_,c',\beta,\gamma) \wedge c \in Out$$

and, as described above, to associated $\beta$ the layer of each lock with the associated "key" obligation we give KIn and KOut the following layers:

$$\text{lay}(\textsc{KIn}(\_,\_,\_,\_,\beta,\_)) = \beta$$
$$\text{lay}(\textsc{KOut}(\_,\_,\_,\_,\beta,\_)) = \beta$$

All other obligations have layer 0.

The property that holding a "key" guard (KIn or KOut), guarantees reachability, or non reacheability from the head node respectively is exploited by the verification procedure, `validate`, to guarantee that two adjacent nodes are correctly locked.

We must also add the following algebraic laws to the guard algebra to encode the invariant that the lock associated with a node cannot change:

$$\textsc{LIn}(c,k,la,c',\alpha,\gamma) = \textsc{LIn}(c,k,la,c',\alpha,\gamma) \bullet \textsc{Witness}(c,la)$$
$$\textsc{LOut}(c,k,la,c',\alpha,\gamma) = \textsc{LOut}(c,k,la,c',\alpha,\gamma) \bullet \textsc{Witness}(c,la)$$
$$\textsc{State}(h,l,C,Out) \bullet \textsc{Witness}(c,la) \text{ is defined.} \Leftrightarrow C(c) = (\_,la,\_,\_,\_,\_)$$

All this ghost state is related to the concrete state by the interpretation of the region **optset** as follows:

$$I(\mathbf{optseth}_r(x,CS,Out,\alpha)) \triangleq \exists C \in \text{AOptList}, e \in \text{Addr}.\ ([\textsc{State}(x,e,C,Out)]_r \wedge \text{Rel}(x,e,CS,C)) *$$
$$\circledast_{c \in \text{dom}(C)} \left( \begin{array}{c} \exists k \in \text{Val}, la, c' \in \text{Addr}, l \in \{0,1\}, \beta, \gamma.\ C(c) = (k,la,l,c',\beta,\gamma) \wedge c \mapsto k,la,c' * Lock(la,l) * \\ \left( \begin{array}{c} (c \notin Out \wedge ((l=1 \wedge [\textsc{LIn}(c,k,la,c',\beta,\gamma)]_r) \vee (l=0 \wedge [\textsc{UIn}(c,k,la,c',\beta,\gamma)]_r))) \vee \\ (c \in Out \wedge ((l=1 \wedge [\textsc{LOut}(c,k,la,c',\beta,\gamma)]_r) \vee (l=0 \wedge [\textsc{UOut}(c,k,la,c',\beta,\gamma)]_r))) \end{array} \right) \wedge \gamma \geq 2^\alpha \end{array} \right)$$

where:

$$\text{Rel}(x,e,CS,C) \triangleq (\forall (c,k,l,c') \in \text{Addr} \times \text{Val} \times \{0,1\} \times \text{Addr}.$$
$$(c,k,l,c') \in CS \Leftrightarrow C(c) = (k,\_,l,c',\_,\_)) \wedge \{(x,-\infty,\_,\_),(e,\infty,\_,\texttt{null})\} \subseteq CS$$

Finally, by defining an extra region, we can hide the uncessary details:

$$I(\mathbf{optset}_r(r',x,S,\alpha)) \triangleq \exists CS \in \mathcal{P}(\text{Addr}\times\mathbb{Z}\times\{0,1\}\times\text{Addr}), Out \in \mathcal{P}(\text{Addr}).\ \mathbf{optseth}_{r'}(x,CS,Out,\alpha) * \lfloor \text{G'} \rfloor_{r'} \wedge \text{keys}(CS,Out) = S$$

$$\mathrm{keys}(CS, Out) = \{\, k \mid (c, k, \_, \_) \in C, c \notin Out, -\infty < k < \infty \,\}$$

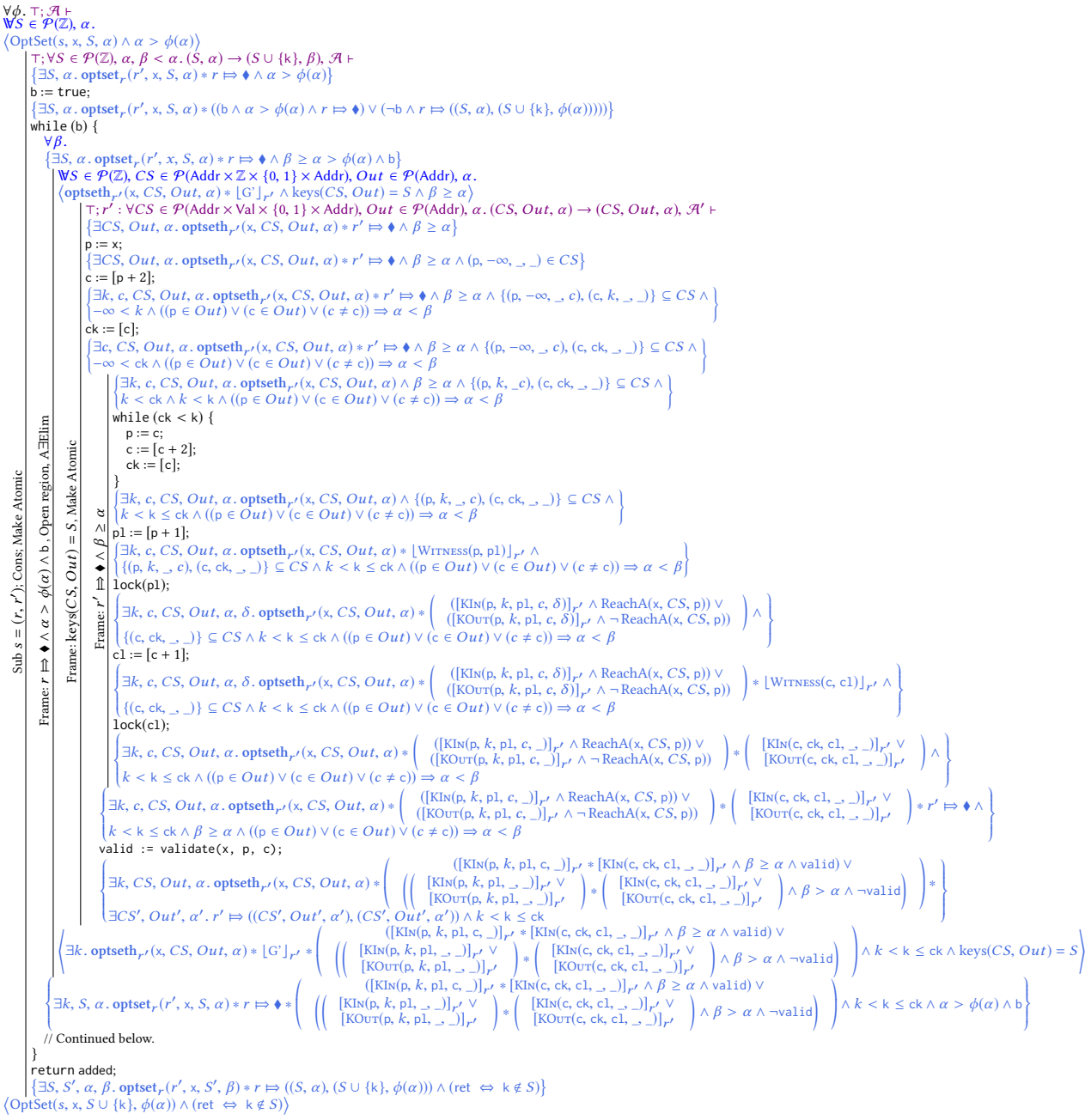
and the following guard labeled transition system:

$$\mathrm{G} : \forall S \in \mathcal{P}(\mathbb{Z}), k \in \mathbb{Z}, \alpha, \beta < \alpha.\, (S, \alpha) \rightsquigarrow (S \uplus \{k\}, \beta) \qquad \mathrm{G} : \forall S \in \mathcal{P}(\mathbb{Z}), k \in \mathbb{Z}, \alpha, \beta < \alpha.\, (S \uplus \{k\}, \alpha) \rightsquigarrow (S, \beta)$$

allows us to define an abstract predicate as in the original specification:

$$\mathrm{OptSet}((r, r'), x, S, \alpha) \triangleq \mathbf{optset}_r(r', x, S, \alpha) * \lfloor \mathrm{G} \rfloor_r$$

With all of this auxiliary state setup, we can proceed to the proof of the module's operations.



$\left\{\begin{array}{l}\exists k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, k, \_, c), (\mathsf{c}, \mathsf{ck}, \_, \_)\} \subseteq CS \wedge k < \mathsf{ck} \wedge k < \mathsf{k} \wedge \\ ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$

`while (ck < k) {`

$\forall \mu.$

$\left\{\begin{array}{l}\exists k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, k, \_, c), (\mathsf{c}, \mathsf{ck}, \_, \_)\} \subseteq CS \wedge \\ k < \mathsf{ck} < \mathsf{k} \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{ck}, CS)| \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$

`p := c;`

$\left\{\begin{array}{l}\exists CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, \mathsf{ck}, \_, \_)\} \subseteq CS \wedge \mathsf{p} = \mathsf{c} \\ \mathsf{ck} < \mathsf{k} \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{ck}, CS)| \wedge (\mathsf{p} \in Out) \Rightarrow \alpha < \beta\end{array}\right\}$

`c := [c + 2];`

$\left\{\begin{array}{l}\exists k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, \mathsf{ck}, \_, c), (\mathsf{c}, k, \_, \_)\} \subseteq CS \wedge \\ \mathsf{ck} < \mathsf{k} \wedge \mathsf{ck} < k \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{ck}, CS)| \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$

`ck := [c];`

$\left\{\begin{array}{l}\exists k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, k, \_, c), (\mathsf{c}, \mathsf{ck}, \_, \_)\} \subseteq CS \wedge \\ k < \mathsf{k} \wedge k < \mathsf{ck} \wedge \mu > \alpha * 2 + |gKeys(\mathsf{ck}, CS)| \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$

`}`

$\left\{\begin{array}{l}\exists k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(x, CS, Out, \alpha) \wedge \{(\mathsf{p}, k, \_, c), (\mathsf{c}, \mathsf{ck}, \_, \_)\} \subseteq CS \wedge \\ k < \mathsf{k} \leq \mathsf{ck} \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', \mathsf{x}, S, \alpha) * r \Rrightarrow \blacklozenge * \left(\begin{array}{c}(\lfloor \textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)\rfloor_{r'} * \lfloor \textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)\rfloor_{r'} \wedge \beta \geq \alpha \wedge \mathsf{valid}) \vee \\ \left(\left(\begin{array}{c}[\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \_, \_)]_{r'} \vee \\ [\textsc{Out}(\mathsf{p}, k, \mathsf{pl}, \_, \_)]_{r'}\end{array}\right) * \left(\begin{array}{c}[\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} \vee \\ [\textsc{Out}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'}\end{array}\right) \wedge \beta > \alpha \wedge \neg\mathsf{valid}\right)\end{array}\right) \wedge \\ k < \mathsf{k} \leq \mathsf{ck} \wedge \alpha > \phi(\alpha) \wedge \mathsf{b}\end{array}\right\}$$

```
if (valid) {
```

$\{\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * r \Rrightarrow \blacklozenge * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} \wedge k < \mathsf{k} \leq \mathsf{ck} \wedge \beta \geq \alpha > \phi(\alpha)\}$

```
  if (ck = k) {
```

$\{\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * r \Rrightarrow \blacklozenge * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{k}, \mathsf{cl}, \_, \_)]_{r'} \wedge k < \mathsf{k} \wedge \beta \geq \alpha > \phi(\alpha)\}$

```
    added := False;
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{k}, \mathsf{cl}, \_, \_)]_{r'} * \\ \exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \beta > \alpha\end{array}\right\}$$

```
  } else {
```

$\{\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * r \Rrightarrow \blacklozenge * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} \wedge k < \mathsf{k} < \mathsf{ck} \wedge \beta \geq \alpha > \phi(\alpha)\}$

```
    nn := alloc(3);
    [nn] := k;
    nnl := newLock();
    [nn + 1] := nnl;
    [nn + 2] := c;
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{nn} \mapsto \mathsf{k}, \mathsf{nnl}, \mathsf{c} * \mathsf{Lock}(\mathsf{nnl}, 0) * \\ [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} \wedge k < \mathsf{k} < \mathsf{ck} \wedge \beta \geq \alpha > \phi(\alpha)\end{array}\right\}$$

```
    [p + 2] := nn;
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{nn}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} * \\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge \beta > \alpha\end{array}\right\}$$

```
    added := True;
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{k}, \mathsf{cl}, \_, \_)]_{r'} * \\ \exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \beta > \alpha\end{array}\right\}$$

```
  }
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{k}, \mathsf{cl}, \_, \_)]_{r'} * \\ \exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \beta > \alpha\end{array}\right\}$$

```
  b := False;
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', x, S, \alpha) * [\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)]_{r'} * [\textsc{In}(\mathsf{c}, \mathsf{k}, \mathsf{cl}, \_, \_)]_{r'} * \\ \exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \beta > \alpha \wedge \neg\mathsf{b}\end{array}\right\}$$

```
}
```

$$\left\{\begin{array}{l}\exists k, S, \alpha.\ \mathbf{optset}_r(r', \mathsf{x}, S, \alpha) * \left(\begin{array}{c}\left(\begin{array}{c}\lfloor \textsc{In}(\mathsf{p}, k, \mathsf{pl}, \mathsf{c}, \_)\rfloor_{r'} * \lfloor \textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)\rfloor_{r'} * \\ \exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \neg\mathsf{b}\end{array}\right) \vee \\ \left(\left(\begin{array}{c}[\textsc{In}(\mathsf{p}, k, \mathsf{pl}, \_, \_)]_{r'} \vee \\ [\textsc{Out}(\mathsf{p}, k, \mathsf{pl}, \_, \_)]_{r'}\end{array}\right) * \left(\begin{array}{c}[\textsc{In}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'} \vee \\ [\textsc{Out}(\mathsf{c}, \mathsf{ck}, \mathsf{cl}, \_, \_)]_{r'}\end{array}\right) * r \Rrightarrow \blacklozenge \wedge \alpha > \phi(\alpha) \wedge \mathsf{b}\right)\end{array}\right) \wedge \beta > \alpha\end{array}\right\}$$

```
unlock(cl);
unlock(pl);
```

$$\left\{\exists k, S, \alpha.\ \mathbf{optset}_r(r', \mathsf{x}, S, \alpha) * \left(\begin{array}{c}(\exists S', \alpha'.\ r \Rrightarrow ((S', \alpha'), (S' \cup \{\mathsf{k}\}, \phi(\alpha'))) \wedge (\mathsf{added} \iff \mathsf{k} \in S') \wedge \neg\mathsf{b}) \vee \\ (r \Rrightarrow \blacklozenge \wedge \mathsf{b} \wedge \alpha > \phi(\alpha))\end{array}\right) \wedge \beta > \alpha\right\}$$

$$\left\{\begin{array}{l}\exists k, k', c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{p}, k, \_, c, \_)]_{r'} \wedge \text{ReachA}(\mathsf{x}, CS, \mathsf{p})) \vee \\ ([\text{KOUT}(\mathsf{p}, k, \_, c, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{x}, CS, \mathsf{p}))\end{array}\right) * \left(\begin{array}{c}[\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'}\end{array}\right) \wedge \\ k < k' \wedge \beta \geq \alpha \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$$

```
prev := x;
```

$$\left\{\begin{array}{l}\exists k, k', c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{p}, k, \_, c, \_)]_{r'} \wedge \text{ReachA}(\mathsf{prev}, CS, \mathsf{p})) \vee \\ ([\text{KOUT}(\mathsf{p}, k, \_, c, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{prev}, CS, \mathsf{p}))\end{array}\right) * \left(\begin{array}{c}[\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'}\end{array}\right) \wedge \\ k < k' \wedge \mathsf{x} = \mathsf{prev} \wedge ((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta\end{array}\right\}$$

```
n := [prev + 2];
nk := [n];
pk := [p];
```

$$\left\{\begin{array}{l}\exists k', prev_k, c, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}[\text{KIN}(\mathsf{p}, \mathsf{pk}, \_, c, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{p}, \mathsf{pk}, \_, c, \_)]_{r'}\end{array}\right) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ \mathsf{pk} < k' \wedge (((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta) \wedge ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk}\end{array}\right\}$$

Frame:

$$\left\{\begin{array}{l}\exists k', prev_k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk}\end{array}\right\}$$

```
while(nk < pk) {
```

$$\forall \mu.$$

$$\left\{\begin{array}{l}\exists k', prev_k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk} \wedge \\ \mathsf{nk} < \mathsf{pk} \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{nk}, CS)|\end{array}\right\}$$

```
  prev := n;
```

$$\left\{\begin{array}{l}\exists k', CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{prev}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{prev}, CS, \mathsf{c}))\end{array}\right) \wedge \\ \wedge(\mathsf{prev}, \mathsf{nk}, \_, \_) \in CS \wedge \mathsf{nk} < \mathsf{pk} \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{nk}, CS)|\end{array}\right\}$$

```
  n := [n + 2];
```

$$\left\{\begin{array}{l}\exists k', CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{prev}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{prev}, CS, \mathsf{c}))\end{array}\right) \wedge \\ ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \_, \_, \_) \in CS \wedge (\mathsf{prev}, \mathsf{nk}, \_, \_) \in CS \wedge \mathsf{nk} < \mathsf{pk} \wedge \mu \geq \alpha * 2 + |gKeys(\mathsf{nk}, CS)|\end{array}\right\}$$

```
  nk := [n];
```

$$\left\{\begin{array}{l}\exists k', prev_k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk} \wedge \mu > \alpha * 2 + |gKeys(\mathsf{ck}, CS)|\end{array}\right\}$$

```
}
```

$$\left\{\begin{array}{l}\exists k', prev_k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk} \leq \mathsf{nk}\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists k', prev_k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}[\text{KIN}(\mathsf{p}, \mathsf{pk}, \_, c, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{p}, \mathsf{pk}, \_, c, \_)]_{r'}\end{array}\right) * \left(\begin{array}{c}([\text{KIN}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \text{ReachA}(\mathsf{n}, CS, \mathsf{c})) \vee \\ ([\text{KOUT}(\mathsf{c}, k', \_, \_, \_)]_{r'} \wedge \neg\text{ReachA}(\mathsf{n}, CS, \mathsf{c}))\end{array}\right) \wedge \\ \mathsf{pk} < k' \wedge (((\mathsf{p} \in Out) \vee (\mathsf{c} \in Out) \vee (c \neq \mathsf{c})) \Rightarrow \alpha < \beta) \wedge ((\mathsf{prev} = \mathsf{p}) \Rightarrow \mathsf{p} \notin Out) \wedge (\mathsf{n}, \mathsf{nk}, \_, \_) \in CS \wedge \\ (\mathsf{prev}, prev_k, \_, \_) \in CS \wedge prev_k < \mathsf{pk} \leq \mathsf{nk}\end{array}\right\}$$

```
return ((n=c) && (p=prev));
```

$$\left\{\exists k, CS, Out, \alpha.\ \mathbf{optseth}_{r'}(\mathsf{x}, CS, Out, \alpha) * \left(\begin{array}{c}([\text{KIN}(\mathsf{p}, k, \mathsf{p1}, \mathsf{c}, \_)]_{r'} * [\text{KIN}(\mathsf{c}, \mathsf{ck}, \mathsf{c1}, \_, \_)]_{r'} \wedge \beta \geq \alpha \wedge \mathsf{ret}) \vee \\ \left(\left(\begin{array}{c}[\text{KIN}(\mathsf{p}, k, \mathsf{p1}, \_, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{p}, k, \mathsf{p1}, \_, \_)]_{r'}\end{array}\right) * \left(\begin{array}{c}[\text{KIN}(\mathsf{c}, \mathsf{ck}, \mathsf{c1}, \_, \_)]_{r'} \vee \\ [\text{KOUT}(\mathsf{c}, \mathsf{ck}, \mathsf{c1}, \_, \_)]_{r'}\end{array}\right) \wedge \beta > \alpha \wedge \neg\mathsf{ret}\right)\end{array}\right) \wedge k < \mathsf{k} \leq \mathsf{ck}\right\}$$

## F Comparison with LiLi

LiLi went a long way in understanding how to specify and verify blocking operations. Although we share most of our goals with LiLi, our approach to specification/verification of progress is radically different, leading to a more compositional verification system.

### F.1 Specifications

LiLi is based on proving simulations, so specifications are themselves programs. To specify an abstractly atomic blocking operation with an atomic specification, LiLi introduces a primitive-blocking specification construct: **await**($\mathbb{B}$){$\mathbb{C}$} executes $\mathbb{C}$ atomically if scheduled in a state where $\mathbb{B}$ is true. As both fair- and spin-lock acquisition are specified using **await**($l = 0$){$l := 1$} the specifications include a flag to indicate whether the **await** should be considered starvation-free (SF) or only deadlock-free (DF). LiLi describes only how to verify an implementation against these specifications; client reasoning is not handled. The contextual refinement result, however, allows using the specifications of a module A to prove correctness of a module B built on top of A. The methodology consists of taking the verified specifications of A's operations, which are in the form **await**($\mathbb{B}$){$\mathbb{C}$}$_{\mathsf{SF/DF}}$, and transforming them into *non-atomic* programs using so-called "wrappers". Then, the transformed specifications are inlined in B in place of the calls to A's operations, before proving B correct. For example, a fair lock (**await**($l = 0$){$l := 1$}$_{\mathsf{SF}}$) will be transformed into a program with a global queue of thread identifiers, and locking would correspond to two separate events: the request of the lock by enqueuing of the current thread id in the queue, and the lock acquisition once the current thread is the head of the queue. The verification of a fair-lock client will not see much simplification from the inlining of the wrapped specifications. In general, for both SF and DF operations, the wrappers will generate more than one event and will contain ghost state. In other words: LiLi's specifications are not truly atomic.

More deeply, the SF and DF properties are not represented in the specifications as logical facts that can be used directly in the logic, but implicitly represented in the dynamic behaviour of the wrappers. This leads, in the verification, to duplication in the proofs: for SF for example, the implementation would need to prove starvation freedom, only for the client to reprove progress of the wrapper to translate the behaviour back to logical facts in the assertions.

We believe that representing blocking with environment liveness assumptions (through $\twoheadrightarrow$) and bounded impedence assumptions (though ordinals) is the key to solving this issue. TaDA Live does not distinguish between client and library verification: it does both *uniformly* in a single system, demonstrating the generality and usefulness of its specifications.

Moreover, injecting a primitive blocking **await** command just to give the specifications of abstractly blocking code, raises a number of ackward issues. For example, to interpret the specifications it is necessary to specify whether they are to be interpreted under weakly- or strongly- fair scheduling, a difference that is immaterial for the underlying code. Another example is the fact that an implementation satisfying **await**($\mathbb{B}$){$\mathbb{C}$} is *required not to terminate* in a context where $\mathbb{B}$ is forever false. It is not clear why a specification should require *non termination*, since the client cannot observe it. Instead, our specifications only require termination under the stated assumptions, thus accepting code that terminates under weaker assumptions, as formalised by the LiveW rule.

### F.2 Verification

LiLi does not directly handle client reasoning within the logics, only verification of an implementation against its specification. Even using the refinement result to verify a client in LiLi using the specifications the imported module would only work if the client is itself another module. The proof system does not include a parallel rule.

Our environment liveness condition was informed by LiLi's *definite progress* condition. In LiLi, progress rely/guarantee is expressed through *definite actions* of the form $P \rightsquigarrow Q$, intuitively requiring that when the system is in a state satisfying $P$, eventually a state satisfying $Q$ will be reached. Definite actions are similar in spirit to our obligations, but differ in two crucial aspects: locality of ghost state; and the abstraction of the environment.

***Locality of ghost state:*** in LiLi, cycles in the argument are avoided by having $P$ in a definite action unambiguously assign the responsibility of fulfilling the action to some thread. This is done by assigning unique thread ids and keeping a global ghost state (updated through ghost code) that records a virtual queue of threads ordered by dependency. To be able to assign responsibility this way, one is forced to manipulate in the proof ghost state that relates all the active threads at the same time.

Our PCM-based obligations allows the proof to represent dependencies between threads as locally as possible: the layered obligations relate only threads that are directly causally related.

***Abstraction of the environment:*** Typically, proofs require a number of inter-dependent definite actions. As LiLi does not have a layering structure, one cannot describe each action separately, and then describe their dependences. Instead, one needs to lump them together in a chain of definite actions precisely describing all possible interactions between these actions. This approach makes the proofs scale poorly as the number of inter-dependent actions increases.

For example, consider a resource protected by two fair locks:

```
op(x) {
  x1 := [x+1]; x2 := [x+2];
  lock(x1);
  lock(x2);
  [x] := 1;
  unlock(x2);
  unlock(x1);
}
```

LiLi's definite action for a thread $t$ would say *"if* `x1` *is owned by* $t$ *and* `x2` *is not owned nor requested by* $t$*, then eventually either* `x1` *will be released or* `x2` *will be requested; if* `x2` *is owned by* $t$ *then it will be released".* [3] This approach makes the proofs scale poorly in the number of locks, because reasoning about locking `x1` requires considering how to measure progress when it will be released as a result of locking and unlocking `x2`. Namely, to prove `lock(x1)` terminates one shows that: when `x1` is requested, we are enqueued in its queue, with $n$ threads ahead of us. Then, each of these threads $t'$ can choose to either release `x1`, or to acquire `x2`. In the former case there's progress, in the latter we need to consider the number $m$ of threads ahead of $t'$ in the queue for acquiring `x2`. Each time `x2` we see progress because $t'$ sees progress toward acquiring `x2`, which brings it closer to releasing `x1`. Once $t'$ acquired `x2` it can only release it after a finite amount of time, and then it cannot try to acquire it again (this needs to be enforced with some additional ghost state recording the number of times each thread tries to acquire `x2` so we can bound this measure by 1). All in all, we have a measure of progress $(n, m)$ where $n, m \in \mathbb{N} \cup \{\omega\}$ for $n$ the threads ahead of us in the queue for `x1` and $m$ the threads in front of the current holder of `x1` in the queue for `x2`. $m$ must be $\omega$ once `x1` to allow a decrease to some finite number $m'$ when the threads joins the `x2` queue.

Our layered obligations solve this problem elegantly: each lock $\mathtt{x}_i$ is associated with an obligation $\mathtt{R}_i$ to release it and the two obligations are declared dependent using the layers: $\mathrm{lay}(\mathtt{R}_1) > \mathrm{lay}(\mathtt{R}_2)$. In the reasoning for locking `x1`, it is sufficient to use the assumption of liveness of the obligation of `x1` only; the environment is free to implement its fulfilment by relying on liveness of any obligation of lower layer (i.e. the one associated with `x2`).

This also circumvents the issue of keeping track of the number of times `x2` can be acquired/released while holding `x1` because we know that any thread holding $\mathtt{R}_1$ needs to show it fulfills it in finite time, so, regardless of how many times, it will only be able to acquire/release `x2` finitely many times.

[3] Note that even to state this liveness invariant one needs to distinguish two separate events for each `lock` operation, the request and the acquisition of the lock.

# G The Model

We introduce the basic components of the model of the logics. Heaps are defined as usual, and form a PCM with classical separation. Our assertions however predicate over various forms of ghost state. To model this we adopt the Views framework [6]. A *world* is a local heap instrumented with local information about ghost state: it is a subjective view of a system configuration. We assume enumerable sets of abstract predicate names $\Sigma$, region type names $\Sigma^{\mathsf{r}}$, abstract states AState, region identifiers RId. To each region type $\mathbf{t} \in \Sigma^{\mathsf{r}}$ we associate a guard algebra $\mathsf{Guard}_{\mathbf{t}}$ and an obligation algebra $\mathsf{Oblig}_{\mathbf{t}}$. By Oblig we denote set of all obligations for all region types. An *atomicity tracker* is an element of the set $\{\blacklozenge, \lozenge\} \uplus (\mathsf{AState} \times \mathsf{AState})$ and composition of atomicity trackers is defined so that $\blacklozenge \cdot \lozenge = \lozenge \cdot \blacklozenge = \blacklozenge$, $\lozenge = \lozenge \cdot \lozenge$, and $(x, y) \cdot (x, y) = (x, y)$, undefined otherwise. Given a partial commutative operation $\bullet$ over $S$, and two partial maps $f : X \rightharpoonup S$, $g : X \rightharpoonup S$, such that $\mathrm{dom}(f) = \mathrm{dom}(g) = Y \subseteq X$, we extend the operation to maps by setting, for all $y \in Y$, $(f \bullet g)(y) = f(y) \bullet g(y)$. The operation is undefined if the domains of the two functions are not equal. Heaps compose classically with $\uplus$ if their domain is disjoint.

**Definition G.1** (World)**.** A world $w \in \mathsf{World}$ is a tuple $w = (h, \alpha, \rho, \gamma, \theta, \chi, \mathcal{L}, \ell, \tau)$ where

- $h : \mathsf{Addr} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$ is the local heap; Val includes Addr and all the basic values, here for simplicity we include only $\{\texttt{false}, \texttt{true}\} \cup \mathbb{N}$.
- $\alpha \in \mathcal{M}_{\mathsf{f}}(\Sigma \times \mathsf{Val}^*)$ is a finite multiset of tuples of abstract predicate names and values; these are used to give semantics to abstract predicate assertions $\mathsf{A}(\vec{z})$.
- $\rho : \mathsf{RId} \rightharpoonup \Sigma^{\mathsf{r}} \times \mathsf{Lvl} \times \mathsf{Val}^* \times \mathsf{AState}$ is a map giving semantics to region assertions $\mathbf{t}^{\lambda}_{r}(\vec{z}, x)$.
- $\gamma : \mathsf{RId} \rightharpoonup \mathsf{Guard}_{\mathbf{t}}$ is a map giving semantics to guard assertions.
- $\theta : \mathsf{RId} \rightharpoonup \mathsf{Oblig}_{\mathbf{t}}$ is a map giving semantics to obligation assertions.
- $\chi : \mathsf{RId} \rightharpoonup \{\blacklozenge, \lozenge\} \uplus (\mathsf{AState} \times \mathsf{AState})$ is a map giving semantics to atomicity tracking assertions.
- $\mathcal{L}$ is a layer structure.
- $\ell : \mathsf{RId} \rightharpoonup \mathsf{Oblig} \rightharpoonup \mathcal{L}$ is a map associating obligations to layers per region identifier. We will ensure, by construction, that for all $\mathrm{O}_1, \mathrm{O}_2 \in \mathsf{Oblig}$ such that $\ell(r)(\mathrm{O}_1) = m_1$ and $\ell(r)(\mathrm{O}_2) = m_2$ for some $r \in \mathsf{RId}$ and $\mathrm{lay}(\mathrm{O}_1) \leq \mathrm{lay}(\mathrm{O}_2)$ we have $m_1 \leq m_2$.
- $\tau : \mathsf{RId} \rightharpoonup \mathsf{LayTr}$ is a map from region identifiers to a layer transformation (see Definition G.6).

and $\mathrm{dom}(\chi) \subseteq \mathrm{dom}(\rho) = \mathrm{dom}(\gamma) = \mathrm{dom}(\theta) = \mathrm{dom}(\ell)$. For a world $w$, we will write $h_w$, $\alpha_w$, $\rho_w$, etc…for the corresponding components of $w$.

Worlds form a PCM $(\mathsf{World}, \cdot, \mathbf{emp})$ where $\mathbf{emp}$ is the set of worlds $w$ with $h_w = \emptyset$, $\alpha_w = \emptyset$, and $\gamma(r) = \mathbf{0}$, $\theta_w(r) = \mathbf{0}$ for every $r \in \mathrm{dom}(\rho_w)$. Worlds composition is defined by

$$(h_1, \alpha_1, \rho, \gamma_1, \theta_1, \chi_1, \mathcal{L}, \ell, \tau) \cdot (h_2, \alpha_2, \rho, \gamma_2, \theta_2, \chi_2, \mathcal{L}, \ell, \tau) = (h_1 \uplus h_2, \alpha_1 \uplus \alpha_2, \rho, \gamma_1 \bullet \gamma_2, \theta_1 \bullet \theta_2, \chi_1 \bullet \chi_2, \mathcal{L}, \ell, \tau)$$

The semantics of assertions in terms of sets of worlds is standard, and in what follows we shall conflate the concept of syntactic assertion and the set of worlds it denotes.

**Definition G.2** (Atomicity Context)**.** An atomicity context, writen $\mathcal{A}$ is a set of elements of the form $(r, x, X \twoheadrightarrow_k X', Y(x))$, where: $r \in \mathsf{RId}$, $x$ is a logical variable, $X$ is a set of abstract states over which $x$ can vary, $X' \subseteq X$ is the set of live states, $k \in \mathcal{L}$ is the layer of the pseudo-quantifier, and $Y$ is a function with $x$ as a free variable, describing the set of abstract states of the region $r$ after the linearization point, depending on the state before it. We require that for each $r \in \mathsf{RId}$ there is at most one tuple with first component $r$ in $\mathcal{A}$. The domain of $\mathcal{A}$, written $\mathrm{dom}(\mathcal{A})$, is the set of region identifiers occurring as the first component of any of the elements of $\mathcal{A}$. Given $\mathcal{A}$, we define $\mathsf{World}_{\mathcal{A}}$ to be the set of worlds $w$ where $\mathrm{dom}(\chi_w) = \mathrm{dom}(\mathcal{A})$.

We defined a partial order over partial functions, $f \lesssim f'$, which asserts that $f'$ may be defined over a larger domain that $f$, but must agree with $f$ over its domain.

**Definition G.3** (Rely). The *rely relation* $\mathbf{R}_{\lambda;\mathcal{A}} \subseteq \mathrm{World}_{\mathcal{A}} \times \mathrm{World}_{\mathcal{A}}$ is defined as:

$$\frac{\begin{array}{c} g \# g' \qquad (s,s') \in \mathcal{T}_{\mathbf{t}}(g')^* \quad \lambda' < \lambda \\ \chi(r) = \blacklozenge \Rightarrow \exists (r, x, X \twoheadrightarrow_k X', \_) \in \mathcal{A}.\, s' \in X \qquad \rho \lesssim \rho' \qquad \ell \lesssim \ell' \qquad \tau \lesssim \tau' \qquad r \notin \mathrm{dom}(\rho) \end{array}}{(h, \alpha, \rho[r \mapsto (\mathbf{t}, \lambda', \vec{v}, s)], \gamma[r \mapsto g], \theta, \chi, \mathcal{L}, \ell, \tau)\ \mathbf{R}_{\lambda;\mathcal{A}}\ (h, \alpha, \rho'[r \mapsto (\mathbf{t}, \lambda', \vec{v}, s')], \gamma[r \mapsto g], \theta, \chi, \mathcal{L}, \ell', \tau')}$$

$$\frac{\exists (r, x, X \twoheadrightarrow_k X', Y(x)) \in \mathcal{A}.\, s \in X \wedge s' \in Y(s) \qquad \lambda' < \lambda \qquad \rho \lesssim \rho' \qquad \ell \lesssim \ell' \qquad \tau \lesssim \tau' \qquad r \notin \mathrm{dom}(\rho)}{(h, \alpha, \rho[r \mapsto (\mathbf{t}, \lambda', \vec{v}, s)], \gamma, \theta, \chi[r \mapsto \lozenge], \mathcal{L}, \ell, \tau)\ \mathbf{R}_{\lambda;\mathcal{A}}\ (h, \alpha, \rho'[r \mapsto (\mathbf{t}, \lambda', \vec{v}, s')], \gamma, \theta, \chi[r \mapsto (s,s')], \mathcal{L}, \ell', \tau')}$$

An assertion $P$ is said *stable* with respect to some level $\lambda$ and atomicity context $\mathcal{A}$, written $\lambda; \mathcal{A} \vDash P$ stable, if it is closed under the rely, that is $\mathbf{R}_{\lambda;\mathcal{A}}(P) \subseteq P$.

**Definition G.4** (Guarantee). The *guarantee relation* $\mathbf{G}_{\lambda;\mathcal{A}} \subseteq \mathrm{World}_{\mathcal{A}} \times \mathrm{World}_{\mathcal{A}}$ is defined as:

$$\begin{array}{l} w\ \mathbf{G}_{\lambda;\mathcal{A}}\ w' \overset{\text{def}}{\iff} (\forall r \in \mathrm{dom}(\rho_w).\, \lambda(\rho_w(r)) \geq \lambda \Rightarrow s(\rho_w(r)) = s(\rho_{w'}(r))) \wedge \\ \qquad \forall r \in \mathrm{dom}(\mathcal{A}). \left( \begin{array}{c} (\chi_w(r) = \chi_{w'}(r) \wedge s(\rho_w(r)) = s(\rho_{w'}(r))) \vee \\ \left( \begin{array}{c} \chi_w(r) = \blacklozenge \wedge \chi_{w'}(r) = (s(\rho_w(r)), s(\rho_{w'}(r))) \wedge \\ \wedge (s(\rho_w(r)), s(\rho_{w'}(r))) \in \mathcal{A}(r) \end{array} \right) \end{array} \right) \end{array}$$

**Definition G.5** (Stability, View). An assertion $P$ is said *stable* with respect to some level $\lambda$ and atomicity context $\mathcal{A}$, written $\lambda; \mathcal{A} \vDash P$ stable, if it is closed under the rely, that is $\mathbf{R}_{\lambda;\mathcal{A}}(P) \subseteq P$.

We define the set of views $\mathrm{View}_{\lambda;\mathcal{A}}$ as the set of all sets of all predicates $P$ such that $\lambda; \mathcal{A} \vDash P$ stable.

**Definition G.6** (Layer Transformations). A layer transformation $\mu \in \mathsf{LayTr}$ is a strictly monotonic map between two layer structures: $\mathcal{L}$ and $\mathcal{L}'$. That is:

$$\forall m_1, m_2 \in \mathcal{L}.\, m_1 < m_2 \Rightarrow \mu(m_1) < \mu(m_2)$$

In the definition of worlds, $\tau$ is a mapping from $r \in \mathsf{RId}$ to a layer transformation that maps the layers of the region $r$ to $\mathcal{L}$, the global layer structure. This allows for a uniform treatment of layers accross regions.

In the following definitions, we assume $\delta$ is a (not necessarily exhaustive) set of abstract predicates and region types definitions.

**Definition G.7** (Region collapse). Given a level, $\lambda$, the region collapse of a world $w$ is:

$$w \downarrow^{\Delta}_{\lambda} = \left\{ w \bullet w' \;\middle|\; w' \in \circledast \{\, I_{\mathbf{t}}(\rho_w(r)) \mid r \in \mathrm{dom}(\rho_w), \exists \lambda' < \lambda, \mathbf{t}.\, \rho_w(r) = (\mathbf{t}, \lambda', \_, \_), I_{\mathbf{t}} \in \Delta \,\} \right\}$$

**Definition G.8** (Abstract predicate collapse).

$$w \lfloor^{\Delta} = \left\{ w \bullet w_1 \bullet \cdots w_n \;\middle|\; \exists n \in \mathbb{N}.\, w_i, \Delta \vDash^n_{id} A^i_{\mu_i}(\vec{v}_i) \right\} \quad \text{where } \alpha_w = \left\{ (A^1_{\mu_1}(\vec{v}_1)), \cdots, (A^n_{\mu_n}(\vec{v}_n)) \right\}$$

where $w, \tau \vDash^n_{\mu} P$ performs a traversal of $P$, stacking the tag maps of abstract predicates to check if each region's obligations are appropriately mapped to global layer structure, $\mathcal{L}$, by $\tau$. Here are some of the key traversal cases:

$$
\begin{array}{rcl}
w, \Delta \vDash_{\mu}^{n+1} A_{\mu'}(\vec{v}) & \Leftrightarrow & (w, \Delta \vDash_{\mu\circ\mu'}^{n} i_A(A_{\mu'}(\vec{v}))) \vee (i_A \notin \Delta) \\
w, \Delta \vDash_{\mu}^{n} \mathbf{t}_r^{\lambda}(\vec{v}, x) & \Leftrightarrow & \left( \begin{array}{l} \rho_w(r) = (\mathbf{t}, \lambda, \vec{v}, x) \wedge h_w = \emptyset \wedge \alpha_w = \emptyset \wedge \\ \forall m \in \mathrm{dom}(\mu).\, \tau_w(r)(m) = \mu(m) \wedge \\ \wedge \forall r' \in \mathrm{dom}(\rho).\, \gamma_w(r') = \mathbf{0} \wedge \theta_w(r') = \mathbf{0} \end{array} \right) \\
w, \Delta \vDash_{\mu}^{n} P \vee Q & \Leftrightarrow & (w, \Delta \vDash_{\mu}^{n} P) \vee (w, \Delta \vDash_{\mu}^{n} Q) \\
w_1 \bullet w_2, \Delta \vDash_{\mu}^{n} P * Q & \Leftrightarrow & (w_1, \Delta \vDash_{\mu}^{n} P) \wedge (w_2, \Delta \vDash_{\mu}^{n} Q) \\
& \ldots &
\end{array}
$$

**Definition G.9** (Reification)**.**

$$\lfloor w \rfloor_{\lambda}^{\Delta} = \left\{ h_{\bar{w}} \;\middle|\; \bar{w} \in w \downarrow_{\lambda}^{\Delta} \lfloor^{\Delta} \right\}$$

**Definition G.10** (Viewshift)**.**

$$\lambda; \mathcal{A}; \vDash_{\Delta} P \Rrightarrow Q \;\Leftrightarrow\; \forall w \in P.\, \exists \bar{w} \in w \downarrow_{\lambda}^{\Delta} \lfloor^{\Delta}, w' \in Q, \bar{w}' \in w' \downarrow_{\lambda}^{\Delta} \lfloor^{\Delta}.\, \bar{w} \Rightarrow_{Ax(\Delta)} \bar{w}' \wedge w\; \mathbf{G}_{\lambda;\mathcal{A}}\; w' \wedge \mathrm{LiveInv}(w, w')$$

where the relation $\Rightarrow_{\Delta}$ allows for substitution of abstract predicates that are not defined in $\Delta$ via the axioms in $\Delta$ and the LiveInv predicate checks that newly created regions obey the invariant that its obligations sum up to the region's total. It is defined as follows:

$$\mathrm{LiveInv}(w, w') \triangleq \forall r \in \mathrm{dom}(\rho_{w'}) \setminus \mathrm{dom}(\rho_w).\, \forall \overline{w}' \in I(\rho_{w'}(r)).\, \theta_{w' \bullet \overline{w}'}(r) = \mathbf{1}$$

# H Semantics and soundness

## H.1 Semantics of atomic triples

The semantic of atomic triples in TaDA-Live are defined with respect to a trace semantic, $[\![\mathbb{C}]\!] \subseteq \mathsf{SchedTrace}$ which gives a concurrent operational semantic to our language. A scheduler trace, $st \in \mathsf{SchedTrace}$, represents a concurrent trace of the state of a computation annotated with information about the threads in the trace. This information allows us to state the fairness constraint on the traces.

The state of a computation is represented by a heap and a global variable store, a mapping from variable names to their values, and each transition is labeled by a thread identifier, $t \in \mathbb{N} \cup \{\mathsf{env}\}$, and a set of the active threads after the transition, $\mathit{TIDs} \subset \mathbb{N}$. As a simplifying assumption, we assume that program variables are alpha renamed to make them unambiguous. Thread identifiers labeling transitions are either the identifiers of threads spawned by the execution of the command, represented with a natural number, or env. These thread identifiers label transitions performed by the command and the environment respectively.

**Definition H.1** (Program Variable Name)**.** The set PVar denotes the set of program variables.

**Definition H.2** (Heap)**.** A heap, $h \in \mathsf{Heap} \triangleq \mathsf{Addr} \rightharpoonup_{\mathbf{fin}} \mathsf{Val}$ is a partial finite mapping from Addr, a set of addresses, to a set of values, Val.

**Definition H.3** (Variable Store)**.** A variable store, $\sigma \in \mathsf{Store} \triangleq \mathsf{PVar} \rightharpoonup \mathsf{Val}$, is a partial mapping from program variable names to their assigned values.

**Definition H.4** (Scheduler Trace)**.** SchedTrace is defined by the following grammar:

$$\mathsf{SchedTrace} \triangleq (\mathsf{Heap}, \mathsf{Store}) \xrightarrow{(\mathbb{N} \cup \{\mathsf{env}\}, \mathcal{P}(\mathbb{N}))} \mathsf{SchedTrace}$$

All $st \in \mathsf{SchedTrace}$ are coinductive, terminating traces of commands will eventually have only transitions with env labeled transitions.

For $i \in \mathbb{N}$ and $st \in \mathsf{SchedTrace}$, the notation $st[i]$ and $st(i)$ represent the $\mathrm{i}^{\mathrm{th}}$ heap and variable store pair and label in the trace respectively.

The trace semantic interleaves steps from the command with arbitrary environmental steps. The semantic of atomic triples asserts both liveness and safety constraints on these traces.

The safety constraints are similar to those of TaDA ([4]). They require that, as long as the trace starts in a state satisfying the precondition of the atomic triple, all transitions performed by the command satisfy the safety constraints of the guarantee relation, as long as the environmental steps satisfy the safety constraints of the rely relation as well as the triple's pseudoquantifier. Moreover, any terminating traces must terminate in a state satisfying the triple's postcondition.

The liveness constraints require that traces that are both fair and have a safe environment and whose environmental steps satisfy the liveness constraints impossed by obligations as well as those of the pseudoquantifier, must terminate.

Fairness of a scheduler trace can be checked by verifying that for every active thread after a transition, there exists some point later on in the thread which is performed by this thread and by the environment, as we must also be fair to the environment to allow it to fullfil its liveness constraints.

**Definition H.5** (fairness)**.** $st \in \mathsf{SchedTrace}$ is fair if it satifies the following predicate:

$$\mathit{fair}(st) = \forall i \in \mathbb{N}.\, \exists \mathit{TIDs} \subseteq \mathbb{N}.\, st(i) = (\_, \mathit{TIDs}) \wedge \forall t' \in \mathit{TIDs} \cup \{\,\mathsf{env}\,\}.\, \exists j > i.\, st(j) = (t', \_)$$

**Definition H.6** (Semantic triple)**.** The semantic triple is defined as follows:

$$\begin{aligned} &m; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X \twoheadrightarrow_k X'.\langle P_{\mathsf{h}} \,|\, P_{\mathsf{a}}(x)\rangle \;\mathbb{C}\; \exists y \in Y.\langle Q_{\mathsf{h}}(x,y) \,|\, Q_{\mathsf{a}}(x,y)\rangle \;\Leftrightarrow \\ &\quad \forall st \in [\![\mathbb{C}]\!].\, \exists it \in \mathsf{InstrTrace}.\, st, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\, \langle P_{\mathsf{h}} | P_{\mathsf{a}}(x)\rangle \exists y \in Y.\, \langle Q_{\mathsf{h}}(x,y) | Q_{\mathsf{a}}(x,y)\rangle \wedge \\ &\qquad\qquad \mathit{playerMatch}(st, it) \wedge (\mathit{fair}(st) \wedge \mathit{safe}(it) \wedge \neg \mathit{locterm}(it)) \Rightarrow \neg \mathit{envLive}^{m;k}_{\mathcal{A}}(it) \end{aligned}$$

where

$$\begin{aligned} \mathit{playerMatch}(\_ \xrightarrow{(i,\_)} ht, \_ \xrightarrow{\mathsf{loc}} st) &= \mathit{playerMatch}(ht, it) \\ \mathit{playerMatch}(\_ \xrightarrow{\mathsf{env}} ht, \_ \xrightarrow{\mathsf{env}} st) &= \mathit{playerMatch}(ht, it) \end{aligned}$$

The trace instrumenting triple:

$$st, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\, \langle P_{\mathsf{h}} | P_{\mathsf{a}}(x)\rangle \exists y \in Y.\, \langle Q_{\mathsf{h}}(x,y) | Q_{\mathsf{a}}(x,y)\rangle$$

imposes the necessary safety constraints while generating an instrumented trace, $it \in \mathsf{InstrTrace}$, which contains the necessary information to verify that the scheduler trace, $st$, it corresponds to has environmental steps that satisfy the correct safety and liveness constrains, as well as if it terminates.

To verify if a trace terminates, the instrumented traces must label its transitions according to whether it was performed by the environment, env, or by the local command, loc, known as *players*. The predicate *playerMatch* checks whether a scheduler trace and an instrumented trace agree on the players responsible for each transition.

The state tracked by the instrumented traces is either:

- $\lightning$, the error state, indicating that the environment performed a step that did not conform to the rely relation, or the trace started in a state that did not satisfy the precondition of the atomic triple.
- $(w, \theta, \psi)$, a triple where:
  - $w$, a world giving the local view of the command.
  - $\theta$, a partial map from RId to obligations, tracking the obligations in the shared state.
  - $\psi$, A partial map defined over $\mathsf{RId} \cup \{\mathsf{Curr}\}$ that tracks the fulfilment of operations in $\mathcal{A}$ and their environmental liveness requirements.

The safety of the environment of a scheduler trace can then be checked by asserting that the instrumented trace it corresponds to never reaches the $\lightning$ state.

**Definition H.7** (Instrumented Traces)**.** An Instrumented trace is defined with the following grammar:

$$\mathsf{InstrTrace} ::= (\mathsf{World} \times (\mathsf{RId} \to \mathsf{Oblig}) \times (\mathsf{RId} \cup \{\mathsf{Curr}\} \to \{\bot, \top, \mathsf{Done}\})) \cup \{\lightning\} \xrightarrow{\mathsf{Player}} \mathsf{InstrTrace}$$

where $\mathsf{Player} \triangleq \{\mathsf{env}, \mathsf{loc}\}$.

For $it \in \mathsf{InstrTrace}$ and $i \in \mathbb{N}$, the short hands $it[i]$ and $it(i)$ references the i$^{\text{th}}$ state in the instrumented trace and and the player performing the i$^{\text{th}}$ transition in $it$ respectively.

**Definition H.8** (Safe)**.** An instrumented trace is *safe* if it never reaches a state not allowed by the environmental and local safety restrictions. Such a trace never reaches the error state $\lightning$. This is checked with the predicate *safe*:

$$\mathit{safe}(it) = \forall i \in \mathbb{N}.\, it[i] \neq \lightning$$

**Definition H.9** (Local Termination)**.** An instrumeted trace $it \in \mathsf{InstrTrace}$ locally terminates if it eventually contains no local transitions. This can be checked with the predicated *locterm*:

$$\mathit{locterm}(it) \triangleq \exists i \in \mathbb{N}.\, \forall j \geq i.\, it(j) = \mathsf{env}$$

The semantic environmental liveness predicate, $\mathit{envLive}^{m;k}_{\mathcal{A}}$, verifies that the environmental steps of a safe instrumented trace satisfy the liveness requirements of the environment. This constitutes checking:

- At every point in the trace, every atomic obligation, $o$, that the environment holds is eventually returned to the shared state, unless an obligation of lower layer, $o'$, is held continuously by one of the local threads.
- At every point in the trace, the liveness requirement of every pseudoquantifier is eventually satisfied, including the current one, unless an obligation, $o$, of layer less than the $k$ of that pseudoquantifier, is continuously held by a local thread.

**Definition H.10** (Semantic Environmental Liveness)**.** The *Semantic environmental liveness* predicate is defined as follows:

$$\mathit{envLive}^{m;k}_{\mathcal{A}}(it) \triangleq \left( \begin{array}{c} \left( \begin{array}{c} \forall o, r.\, \big(\ \mathit{atom}(o) \land \mathrm{lay}(o) < m \land \neg \mathit{obligLive}(it, o, r)\ \big) \Rightarrow \\ \exists i \in \mathbb{N}, o', r'.\, (\mathrm{lay}(o') < \mathrm{lay}(o) \lor o = o') \land \forall j \geq i.\, o' \leq \theta_{w(it[j])}(r') \end{array} \right) \land \\ \left( \begin{array}{c} \neg \mathit{currPQLive}(it) \Rightarrow \exists i \in \mathbb{N}, o, r. \\ \mathit{atom}(o) \land \mathrm{lay}(o) < k \land \forall j \geq i.\, o \leq \theta_{w(it[j])}(r) \end{array} \right) \land \\ \left( \begin{array}{c} \forall (r, \_, \_ \twoheadrightarrow_{k_r} \_, \_) \in \mathcal{A}.\, \neg \mathit{PQLive}_{\mathcal{A}}(it, r) \Rightarrow \exists i \in \mathbb{N}, o, r'. \\ \mathit{atom}(o) \land \mathrm{lay}(o) < k_r \land \forall j \geq i.\, o \leq \theta_{w(it[j])}(r') \end{array} \right) \end{array} \right)$$

where:

$$\mathit{obligLive}(it, o, r) \triangleq \forall i \in \mathbb{N}.\, \exists j \geq i.\, o \leq \theta(it[j])(r)$$
$$\mathit{currPQLive}(it) \triangleq \forall i \in \mathbb{N}.\, \exists j \geq i.\, \psi(it[j])(\mathsf{Curr}) \in \{\top, \mathsf{Done}\}$$
$$\mathit{PQLive}(it, r) \triangleq \forall i \in \mathbb{N}.\, \exists j \geq i.\, o \leq \psi(it[j])(r) \in \{\top, \mathsf{Done}\}$$

**Definition H.11** (Trace Semantic)**.** The trace semantic, $[\![\_]\!] : \mathsf{Cmd} \to \mathsf{SchedTrace}$, gives the set of Scheduler Traces a given command can produce given any initial state and environmental interference.

To produce this, we first need an auxiliary semantics that, given a thread-pool, $T \in \mathbb{N} \rightharpoonup_{\mathbf{fin}} \mathit{Store} \times \mathbb{C}$, and a starting state, $h$, generates all possible traces starting from this state and thread pool with arbitrary environmental interference interleaved in at arbitrary points.

A thread-pool is a mapping from a thread to a pair of a local variable store for the thread and the remaining command for the thread.

The auxiliary semantic is defined as follows:

$$\langle T \rangle(s, h) = \left\{ s, h \xrightarrow{(\mathsf{env}, \mathrm{dom}(T))} st \;\middle|\; st \in \langle T \rangle(s, h') \right\} \cup \left\{ s, h \xrightarrow{(t, \mathrm{dom}(T'))} st \;\middle|\; s, h, T \xrightarrow{t} s', h', T', st \in \langle T' \rangle(s', h') \right\}$$

The first set interleaves in arbitrary environmental steps, labeled with env as a thread identifier. The second performs a local step, defined with respect to the thread-pool small-step semantics in figure 17. Steps of this semantic are labeled by the thread identifier of the local thread that performed the transition. The auxiliary semantic keeps track of this, as well as the set of active threads after each local step.

The thread small-step semantic is in turn defined in terms of the command semantic in figure 18, which is labeled with a thread operation, A thread operation is either $\bullet$, indicating that the step does not spawn new threads, or $\mathit{fork}(t, s, \mathbb{C})$, when the step spawns a new thread. $t$ is the threads new fresh thread identifier, which is stored locally, allowing the thread to join on the thread it spawned later with the operation `join`.

The operation `join` is mediated by the thread-semantic, and allowed to progress if the thread it is joining on has terminated.

Using the auxiliary semantic, we can defined the command semantics as follows:

$$[\![\mathbb{C}]\!] = \bigcup \{\, \langle \emptyset[t \mapsto \mathbb{C}] \rangle(s, h) \mid t \in \mathbb{N}, s \in \mathsf{Store}, h \in \mathsf{Heap} \,\}$$

$$\frac{s, h, \mathbb{C} \xrightarrow{\bullet} s', h', \mathbb{C}'}{s, h, T[t \mapsto \mathbb{C}] \xrightarrow{t} s', h', T[t \mapsto \mathbb{C}']}$$

$$\frac{s, h, \mathbb{C} \xrightarrow{fork(t', \mathbb{C}'')} s', h', \mathbb{C}'}{h, T[t \mapsto (s, \mathbb{C})] \xrightarrow{t} s', h', T[t \mapsto \mathbb{C}', t' \mapsto \mathbb{C}'']}$$

$$\frac{t \notin \mathrm{dom}(T)}{h, T[t \mapsto (s, \texttt{skip})] \xrightarrow{t} h', T}$$

$$\frac{t \notin \mathrm{dom}(T)}{h, T[t \mapsto (s, \texttt{return } \mathbb{E}; \mathbb{C})] \xrightarrow{t} h', T}$$

$$\frac{[\![\mathbb{E}]\!]^E_s \notin \mathrm{dom}(T) \cup \{t\}}{s, h, T[t \mapsto \texttt{join(}\mathbb{E}\texttt{)}; \mathbb{C}] \xrightarrow{t} s, h, T[t \mapsto \mathbb{C}]}$$

**Figure 17.** Thread pool semantic.

$$\frac{\mathrm{fv}(\mathbb{E}) \subseteq \mathrm{dom}(s)}{s, h, \mathtt{x} := \mathbb{E} \xrightarrow{\bullet} s[\mathtt{x} \mapsto [\![\mathbb{E}]\!]^E_s], h, \mathtt{skip}} \qquad \frac{\mathrm{fv}(\mathbb{E}) \subseteq \mathrm{dom}(s) \quad [\![\mathbb{E}]\!]^E_s \in \mathrm{dom}(h)}{s, h, \mathtt{x} := \mathtt{[}\mathbb{E}\mathtt{]} \xrightarrow{\bullet} s[\mathtt{x} \mapsto h([\![\mathbb{E}]\!]^E_s)], h, \mathtt{skip}}$$

$$\frac{\mathrm{fv}(\mathbb{E}_1) \cup \mathrm{fv}(\mathbb{E}_2) \subseteq \mathrm{dom}(s) \quad [\![\mathbb{E}_1]\!]^E_s \in \mathrm{dom}(h)}{s, h, \mathtt{[}\mathbb{E}_1\mathtt{]} := \mathbb{E}_2 \xrightarrow{\bullet} s, h\left[[\![\mathbb{E}_1]\!]^E_s \mapsto [\![\mathbb{E}_2]\!]^E_s\right], \mathtt{skip}}$$

$$\frac{\mathrm{fv}(\mathbb{E}_1) \cup \mathrm{fv}(\mathbb{E}_2) \cup \mathrm{fv}(\mathbb{E}_3) \subseteq \mathrm{dom}(s) \quad [\![\mathbb{E}_1]\!]^E_s \in \mathrm{dom}(h) \quad h([\![\mathbb{E}_1]\!]^E_s) = [\![\mathbb{E}_2]\!]^E_s}{s, h, \mathtt{r} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \xrightarrow{\bullet} s[\mathtt{r} \mapsto 1], h\left[[\![\mathbb{E}_1]\!]^E_s \mapsto [\![\mathbb{E}_3]\!]^E_s\right], \mathtt{skip}}$$

$$\frac{\mathrm{fv}(\mathbb{E}_1) \cup \mathrm{fv}(\mathbb{E}_2) \cup \mathrm{fv}(\mathbb{E}_3) \subseteq \mathrm{dom}(s) \quad [\![\mathbb{E}_1]\!]^E_s \in \mathrm{dom}(h) \quad h([\![\mathbb{E}_1]\!]^E_s) \neq [\![\mathbb{E}_2]\!]^E_s}{s, h, \mathtt{r} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \xrightarrow{\bullet} s[\mathtt{r} \mapsto 0], h, \mathtt{skip}}$$

$$\frac{}{s, h, \mathtt{x} := \mathtt{fork}(\mathbb{C}) \xrightarrow{fork(t, s, \mathbb{C})} s[\mathtt{x} \mapsto t], h, \mathtt{skip}}$$

$$\frac{[\![\mathbb{B}]\!]^B_s}{s, h, \mathtt{if}(\mathbb{B})\{\mathbb{C}_1\}\mathtt{else}\{\mathbb{C}_2\} \xrightarrow{\bullet} s, h, \mathbb{C}_1} \qquad \frac{\neg[\![\mathbb{B}]\!]^B_s}{s, h, \mathtt{if}(\mathbb{B})\{\mathbb{C}_1\}\mathtt{else}\{\mathbb{C}_2\} \xrightarrow{\bullet} s, h, \mathbb{C}_2}$$

$$\frac{[\![\mathbb{B}]\!]^B_s}{s, h, \mathtt{while}(\mathbb{B})\{\mathbb{C}\} \xrightarrow{\bullet} s, h, \mathbb{C}; \mathtt{while}(\mathbb{B})\{\mathbb{C}\}} \qquad \frac{\neg[\![\mathbb{B}]\!]^B_s}{s, h, \mathtt{while}(\mathbb{B})\{\mathbb{C}\} \xrightarrow{\bullet} s, h, \mathtt{skip}}$$

$$\frac{\mathtt{f} \in \mathrm{dom}(\delta) \quad \delta(\mathtt{f}) = (\vec{\mathtt{x}}, \mathbb{C}) \quad s(\vec{\mathtt{x}}) = [\![\vec{\mathbb{E}}]\!]^E_s}{s, h, \mathtt{r} := \mathtt{f}(\vec{\mathbb{E}}) \xrightarrow{\bullet} s, h, \langle s, \mathtt{x}, \mathbb{C}\rangle} \qquad \frac{\bar{s}, h, \mathbb{C} \xrightarrow{t\text{-}op} \bar{s}', h', \mathbb{C}'}{s, h, \langle \bar{s}, x, \mathbb{C}\rangle \xrightarrow{t\text{-}op} s, h', \langle \bar{s}', \mathtt{x}, \mathbb{C}'\rangle}$$

$$\frac{}{s, h, \langle \bar{s}, \mathtt{x}, \mathtt{skip}\rangle \xrightarrow{\bullet} s, h, \mathtt{skip}} \qquad \frac{}{s, h, \langle \bar{s}, \mathtt{x}, \mathtt{return}\ \mathbb{E}; \mathbb{C}\rangle \xrightarrow{\bullet} s\left[\mathtt{x} \mapsto [\![\mathbb{E}]\!]^E_{\bar{s}}\right], h, \mathtt{skip}}$$

$$\frac{s, h, \mathbb{C}_1 \xrightarrow{t\text{-}op} s', h', \mathbb{C}_1\text{'}}{s, h, \mathbb{C}_1; \mathbb{C}_2 \xrightarrow{t\text{-}op} s', h', \mathbb{C}_1\text{'}; \mathbb{C}_2} \qquad \frac{}{s, h, \mathtt{skip}; \mathbb{C} \xrightarrow{\bullet} s', h', \mathbb{C}}$$

**Figure 18.** Command semantic.

**Definition H.12** (Trace Instrumenting triple)**.** We say that a $st \in \mathsf{SchedTrace}$ is instrumented by $it \in \mathsf{InstrTrace}$ with respect to a specification $\mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle$ if: $st, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle$ defined co-inductively as:

$$(s,h \xrightarrow{-} \_), (\lightning \xrightarrow{-} it) \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \ \Leftrightarrow$$
$$\neg\exists w, \theta. \left( \begin{array}{l} s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} \exists x \in X.\ P_\mathsf{h} * P_\mathsf{a}(x) \vee \\ s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} \exists x \in X, y \in Y.\ q_\mathsf{h}(x,y) * q_\mathsf{a}(x,y) \end{array} \right) \wedge \mathit{err}(it)$$

$$s,h \xrightarrow{\mathsf{env}} (s,h' \xrightarrow{t} st), (w,\theta,\psi) \xrightarrow{\mathsf{env}} (\lightning \xrightarrow{pl} it) \vDash^m_{\lambda;\mathcal{A}}$$
$$\mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \ \Leftrightarrow\ \exists x \in X.$$
$$s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} P_\mathsf{h} * P_\mathsf{a}(x) \wedge \mathit{checkTrack}_\mathcal{A}(w,\psi) \wedge$$
$$((x \in X' \wedge \psi(\mathsf{Curr}) = \mathsf{True}) \vee (x \notin X' \wedge \psi(\mathsf{Curr}) = \mathsf{False})) \wedge$$
$$s,h' \xrightarrow{t} st, \lightning \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle$$

$$s,h \xrightarrow{\mathsf{env}} (s,h' \xrightarrow{t} st), (w,\theta,\psi) \xrightarrow{\mathsf{env}} ((w',\theta',\psi') \xrightarrow{pl} it) \vDash^m_{\lambda;\mathcal{A}} \langle P_\mathsf{h}|P_\mathsf{a}\rangle\langle P'_\mathsf{h}|P_\mathsf{a}\rangle \ \Leftrightarrow$$
$$s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} P_\mathsf{h} * P_\mathsf{a} \wedge w\ \mathbf{R}_{\lambda;\mathcal{A}}\ w' \wedge$$
$$\psi(\mathsf{Curr}) = \mathsf{Done} \Rightarrow \psi'(\mathsf{Curr}) = \mathsf{Done} \wedge \mathit{checkTrack}_\mathcal{A}(w,\psi) \wedge$$
$$\left( \begin{array}{l} (\psi(\mathsf{Curr}) = \mathsf{Done} \wedge s,h' \xrightarrow{t} st, A \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \langle P_\mathsf{h}|P_\mathsf{a}\rangle\langle P'_\mathsf{h}|P_\mathsf{a}\rangle) \vee \\ (\psi(\mathsf{Curr}) = \top \wedge s,h' \xrightarrow{t} st, A \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \{P_\mathsf{h}\}\{P'_\mathsf{h}\}) \end{array} \right)$$

$$s,h \xrightarrow{\mathsf{env}} (s,h' \xrightarrow{t} st), (w,\theta,\psi) \xrightarrow{\mathsf{env}} ((w',\theta',\psi') \xrightarrow{pl} it) \vDash^m_{\lambda;\mathcal{A}}$$
$$\mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \ \Leftrightarrow\ \exists x \in X.$$
$$s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} P_\mathsf{h} * P_\mathsf{a}(x) \wedge w\ \mathbf{R}_{\lambda;\mathcal{A}}\ w' \wedge \mathit{checkTrack}_\mathcal{A}(w,\psi) \wedge$$
$$((x \in X' \wedge \psi(\mathsf{Curr}) = \mathsf{True}) \vee (x \notin X' \wedge \psi(\mathsf{Curr}) = \mathsf{False})) \wedge$$
$$s,h' \xrightarrow{t} st, (w',\theta',\psi') \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle$$

$$s,h \xrightarrow{(\_,\_)} (s',h' \xrightarrow{t} st), (w,\theta,\psi) \xrightarrow{\mathsf{loc}} ((w',\theta',\psi') \xrightarrow{pl} it) \vDash^m_{\lambda;\mathcal{A}}$$
$$\mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \ \Leftrightarrow\ \exists x \in X.$$
$$s,h,(w,\theta) \vDash^m_{\lambda;\mathcal{A}} P_\mathsf{h} * P_\mathsf{a}(x) \wedge w\ \mathbf{G}_{\lambda;\mathcal{A}}\ w' \wedge \mathit{checkTrack}_\mathcal{A}(w,\psi) \wedge$$
$$((x \in X' \wedge \psi(\mathsf{Curr}) = \mathsf{True}) \vee (x \notin X' \wedge \psi(\mathsf{Curr}) = \mathsf{False})) \wedge$$
$$\forall r \in \mathrm{dom}(\rho_{w'}) \setminus \mathrm{dom}(\rho_w).\ \rho_{w'}(r) = (\mathbf{t}, \_, \_, \_) \wedge \theta_{w'}(r) \bullet \theta'(r) = \mathbf{1}_\mathbf{t} \wedge$$
$$\left( \begin{array}{l} \left( \begin{array}{l} \exists P'_\mathsf{h} \in \mathsf{Store} \to \mathsf{View}^m_{\lambda;\mathcal{A}}.\ \neg \mathit{done}((w',\theta',\psi') \xrightarrow{pl} it) \wedge \\ s',h' \xrightarrow{t} st, (w',\theta',\psi') \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\ \langle P'_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\ \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \end{array} \right) \vee \\ \left( \begin{array}{l} \exists Q'_\mathsf{h} \in X \times Y \to \mathsf{Store} \to \mathsf{View}^m_{\lambda;\mathcal{A}}.\ \exists y \in Y.\ \psi'(\mathsf{Curr}) = \mathsf{Done} \wedge \\ s',h',(w',\theta') \vDash^m_{\lambda;\mathcal{A}} Q'_\mathsf{h}(x,y) * Q_\mathsf{a}(x,y) \wedge \\ s',h' \xrightarrow{t} st, (w',\theta',\psi') \xrightarrow{pl} it \vDash^m_{\lambda;\mathcal{A}} \{Q'_\mathsf{h}(x,y)\}\{Q_\mathsf{h}(x,y)\} \end{array} \right) \end{array} \right)$$

where:

$$h,(w,\theta_s) \vDash_{\lambda;\mathcal{A}} P \triangleq \exists s \in \mathsf{Store}, E \in \mathsf{View}^\perp_{\lambda;\mathcal{A}}, e \in E, \bar{w} \in \lfloor w \rfloor^\delta_\lambda.\ w \in p(s) \wedge h = h_{\bar{w}\bullet e} \wedge$$
$$(\forall r \in \mathrm{dom}(\rho_w).\ \theta_{\bar{w}}(r) = \theta_w(r) \bullet \theta_s(r)) \wedge (\forall r \in \mathrm{dom}(\rho_{\bar{w}\bullet e}).\ \theta_{\bar{w}\bullet e} = \mathbf{1}_{t(\rho(r))})$$

$$checkTrack_{\mathcal{A}}(w,\psi) \triangleq \forall r \in \mathrm{dom}(\mathcal{A}).\, (r,\_,X \twoheadrightarrow_k X',\_) \wedge \left( \begin{array}{l} (\chi(r) = (\_,\_) \wedge \psi(r) = \mathsf{Done}) \vee \\ \left( \begin{array}{l} \chi(r) \neq (\_,\_) \wedge \exists x \in X.\, \rho(r) = (\_,\_,\_,x) \wedge \\ ((x \in X' \wedge \psi(r) = \mathsf{True}) \vee (x \notin X' \wedge \psi(r) = \mathsf{False})) \end{array} \right) \end{array} \right)$$

$$ht, it \vDash^m_{\lambda;\mathcal{A}} \{P\}\{Q\} \triangleq ht, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in \{*\} \twoheadrightarrow_\perp \{*\}.\, \langle P|\mathsf{emp}\rangle \exists y \in \{*\}.\, \langle Q|\mathsf{emp}\rangle$$
$$st, it \vDash^m_{\lambda;\mathcal{A}} \langle P_\mathsf{h}|P_\mathsf{a}\rangle\langle Q_\mathsf{h}|Q_\mathsf{a}\rangle \triangleq st, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in \{*\} \twoheadrightarrow_\perp \{*\}.\, \langle P_\mathsf{h}|P_\mathsf{a}\rangle \exists y \in \{*\}.\, \langle Q_\mathsf{h}|Q_\mathsf{a}\rangle$$

## H.2 Soundness

### H.2.1 Soundness of Parallel rule

Statement of soundness of the parallel rule:

**Theorem H.13.** *Given:*

$$m;\lambda;\mathcal{A} \vDash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \tag{6}$$
$$m;\lambda;\mathcal{A} \vDash \{P_2\}\ \mathbb{C}_2\ \{Q_2\} \tag{7}$$
$$\lambda;\mathcal{A} \vdash Q_1 \trianglerighteq m \tag{8}$$
$$\lambda;\mathcal{A} \vdash Q_2 \trianglerighteq m \tag{9}$$

*then:*

$$m;\lambda;\mathcal{A} \vDash \{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}$$

*Proof.* Given our assumptions, We will prove:

$$\forall st \in \cup\{\, \langle \emptyset[n \mapsto (s_1,\mathbb{C}_1), m \mapsto (s_2,\mathbb{C}_2)]\rangle(h) \mid n,m \in \mathbb{N}, s_1,s_2 \in \mathsf{Store}, h \in \mathsf{Heap} \,\}$$
$$\exists it \in \mathsf{InstrTrace}.\, st, it \vDash^m_{\lambda;\mathcal{A}} \mathbb{V}x \in X \twoheadrightarrow_k X'.\, \langle P_\mathsf{h}|P_\mathsf{a}(x)\rangle \exists y \in Y.\, \langle Q_\mathsf{h}(x,y)|Q_\mathsf{a}(x,y)\rangle \wedge$$
$$playerMatch(st,it) \wedge (fair(st) \wedge safe(it) \wedge \neg locterm(it)) \Rightarrow \neg envLive^{m;k}_{\mathcal{A}}(it)$$

Which clearly implies our goal.

Take $st \in \cup\{\, \langle \emptyset[n \mapsto (s_1,\mathbb{C}_1), m \mapsto (s_2,\mathbb{C}_2)]\rangle(h) \mid n,m \in \mathbb{N}, s_1,s_2 \in \mathsf{Store}, h \in \mathsf{Heap} \,\}$ arbitrary.

We start off by defining a "scheduler trace combination" operator, $\bowtie_s$, which, if possible, combines two traces from the perspective of two commands and generates a trace of them running in parallel:

$$((h,\sigma_1) \xrightarrow{(i,\mathit{TIDs})} st) \bowtie_s ((h,\sigma_2) \xrightarrow{(\mathsf{env},\mathit{TIDs}')} st') = (h,\sigma_1 \uplus \sigma_2) \xrightarrow{(i,\mathit{TIDs} \uplus \mathit{TIDs}')} (st \bowtie_s st')$$

$$((h,\sigma_1) \xrightarrow{(\mathsf{env},\mathit{TIDs})} st) \bowtie_s ((h,\sigma_2) \xrightarrow{(i,\mathit{TIDs}')} st') = (h,\sigma_1 \uplus \sigma_2) \xrightarrow{(i,\mathit{TIDs} \uplus \mathit{TIDs}')} (st \bowtie_s st')$$

$$((h,\sigma_1) \xrightarrow{(\mathsf{env},\mathit{TIDs})} st) \bowtie_s ((h,\sigma_2) \xrightarrow{(\mathsf{env},\mathit{TIDs}')} st') = (h,\sigma_1 \uplus \sigma_2) \xrightarrow{(\mathsf{env},\mathit{TIDs} \uplus \mathit{TIDs}')} (st \bowtie_s st')$$

**Lemma H.14.** $\exists st_1 \in [\![\mathbb{C}_1]\!], st_2 \in [\![\mathbb{C}_2]\!].\, st_1 \bowtie_s st_2 = st$

Set $st_1$ and $st_2$ using this lemma. Using assumptions 6 and 7, we can get:

$$\exists it_1 \in \mathsf{InstrTrace}.\ st_1, it_1 \vDash^m_{\lambda;\mathcal{A}} \{P_1\}\{Q_1\} \land playerMatch(st_1, it_1) \land (fair(st_1) \land safe(it_1) \land \neg locterm(it_1)) \Rightarrow \neg envLive^{m;k}_{\mathcal{A}}(it_1) \tag{10}$$

$$\exists it_2 \in \mathsf{InstrTrace}.\ st_2, it_2 \vDash^m_{\lambda;\mathcal{A}} \{P_2\}\{Q_2\} \land playerMatch(st_2, it_2) \land (fair(st_2) \land safe(it_2) \land \neg locterm(it_2)) \Rightarrow \neg envLive^{m;k}_{\mathcal{A}}(it_2) \tag{11}$$

Set $it_1$ and $it_2$ using these theorems.

We will now use these theorems to construct a trace $it$ that we will show satisfies the necessary properties for our goal. To do this, we will defined a "instrumented trace combination" operator, $\bowtie_i$, which acts similarly to $\bowtie_s$, but on instrumented traces:

$$(\lightning \xrightarrow{\text{env}} it) \bowtie_i (\_ \xrightarrow{pl} it') = \lightning \xrightarrow{pl} (it \bowtie_i it') \quad (\_ \xrightarrow{\text{env}} it) \bowtie_i (\lightning \xrightarrow{pl} it') = \lightning \xrightarrow{pl} (it \bowtie_i it')$$

$$(\lightning \xrightarrow{\text{loc}} it) \bowtie_i (\_ \xrightarrow{\text{env}} it') = \lightning \xrightarrow{\text{loc}} (it \bowtie_i it') \quad (\_ \xrightarrow{\text{loc}} it) \bowtie_i (\lightning \xrightarrow{\text{env}} it') = \lightning \xrightarrow{\text{loc}} (it \bowtie_i it')$$

$$((w, \theta, \psi) \xrightarrow{\text{env}} it) \bowtie_i ((w', \theta, \psi) \xrightarrow{pl} it') = (w \bullet w', \theta, \psi) \xrightarrow{pl} (it \bowtie_i it') \quad \text{when } w \bullet w' \text{ defined.}$$

$$((w, \theta, \psi) \xrightarrow{\text{loc}} it) \bowtie_i ((w', \theta, \psi) \xrightarrow{\text{env}} it') = (w \bullet w', \theta, \psi) \xrightarrow{\text{loc}} (it \bowtie_i it') \quad \text{when } w \bullet w' \text{ defined.}$$

The combination of two instrumented trace must be in an error, $\lightning$, state when either of the two traces observes a safety violation, as the safety requirements of the two commands running in parallel is necessarily violated. Otherwise, they must both agree on the obligations in the shared state, $\theta$, and on the state of the satisfaction of pseudoquantifiers that are currently in action, $\psi$.

**Lemma H.15.**

$$st_1, it_1 \vDash^m_{\lambda;\mathcal{A}} \{P_1\}\{Q_1\} \land st_2, it_2 \vDash^m_{\lambda;\mathcal{A}} \{P_2\}\{Q_2\} \Rightarrow st, (it_1 \bowtie_i it_2) \vDash^m_{\lambda;\mathcal{A}} \{P_1 * P_2\}\{Q_1 * Q_2\}$$

Using this lemma and assumptions 6 and 7, by setting $it = it_1 \bowtie_i it_2$, we derive the first conjunct of our goal:

$$st, it \vDash^m_{\lambda;\mathcal{A}} \{P_1 * P_2\}\{Q_1 * Q_2\}$$

The second conjunct, $playerMatch(st, it)$, can be proven from the fact that $st = st_1 \bowtie_s st_2$ and $it = it_1 \bowtie_i it_2$.

Finally, to prove the last conjunct, we assume:

$$fair(st) \land safe(it) \land \neg locterm(it) \tag{12}$$

**Lemma H.16.**

$$fair(st_1 \bowtie_s st_2) \Rightarrow fair(st_1) \land fair(st_2)$$

**Lemma H.17.**

$$safe(st_1 \bowtie_s st_2) \Rightarrow safe(st_1) \land safe(st_2)$$

From 12 and these two lemmas, we can derive $fair(st_1)$, $fair(st_2)$, $safe(st_1)$ and $safe(st_2)$.
$it$ could fail to terminate due to a thread in $it_1$, $it_2$ or both, therefore:

$$\neg locterm(it) \Rightarrow \left( \begin{array}{l} (locterm(it_1) \wedge \neg locterm(it_2)) \vee \\ (\neg locterm(it_1) \wedge locterm(it_2)) \vee \\ (\neg locterm(it_1) \wedge \neg locterm(it_2)) \end{array} \right)$$

We now consider each case, one at a time.

First assume $locterm(it_1) \wedge \neg locterm(it_2)$. From these consequences and 10, we can derive $\neg envLive^{m;k}_{\mathcal{A}}(it_2)$. This in turn results in 3 cases:

$$\left( \begin{array}{c} \exists o, r.\ (\ atom(o) \wedge \mathrm{lay}(o) < m \wedge \neg obligLive(it_2, o, r)\ ) \wedge \\ \forall i \in \mathbb{N}, o', r'.\ (\mathrm{lay}(o') \geq \mathrm{lay}(o) \wedge o \neq o') \vee \exists j \geq i.\ o' \notin \theta_{w(it_2[j])}(r') \end{array} \right) \tag{13}$$

$$\left( \begin{array}{c} \neg currPQLive(it_2) \wedge \forall i \in \mathbb{N}, o, r. \\ \neg atom(o) \vee \mathrm{lay}(o) \geq k \vee \exists j \geq i.\ o \notin \theta_{w(it_2[j])}(r) \end{array} \right) \tag{14}$$

$$\left( \begin{array}{c} \forall (r, \_, \_ \twoheadrightarrow_{k_r} \_, \_) \in \mathcal{A}.\ \neg PQLive_{\mathcal{A}}(it_2, r) \wedge \forall i \in \mathbb{N}, o, r'. \\ \neg atom(o) \vee \mathrm{lay}(o) \geq k_r \vee \exists j \geq i.\ o \notin \theta_{w(it_2[j])}(r') \end{array} \right) \tag{15}$$

Considering case 13, set $o$ and $r$. As $\neg obligLive(it, o, r)$, holds, there exists some index $i_2$, such that at every point after this index, the obligation $o$ is not in the shared state. Also, as a consequence of $locterm(it_1)$, there exists some index $i_1$ after which there are no more local steps in $it_1$ and due to our assumption 8, no world in $it_1$ after this point can hold an obligation of layer less than m. Due to this, these worlds cannot hold $o$, as $\mathrm{lay}(o) < m$.

$$\forall j \geq i_1.\ o \notin \theta(w(it_1[j]))(r) \tag{16}$$

$$\forall j \geq i_2.\ o \notin \theta(it_2[j])(r) \tag{17}$$

Also, as:

$$\forall i \in \mathbb{N}, o', r'.\ (\mathrm{lay}(o') \geq \mathrm{lay}(o) \wedge o \neq o') \vee \exists j \geq i.\ o' \notin \theta_{w(it_2[j])}(r')$$

Setting $i = i_2$, $o' = o$ and $r' = r$, we know there exists $j \geq i_2$, such that $o' \notin \theta_{w(it_2[j])}(r')$, however, since it is also not in the shared state, from 17, this implies that:

$$\forall j \geq i_2.\ o \notin \theta(w(it_2[j]))(r)$$

From 17, we can also infer, as both threads must agree on the shared state:

$$\forall j \geq i_2.\ o \notin \theta(it[j])(r)$$

As a consequence, since $o$ is an atom:

$$\forall j \geq \max(i_1, i_2).\ o \notin \theta(w(it[j]))(r) \bullet \theta(it[j])(r)$$

From this, it follows that $envLive^{m;k}_{\mathcal{A}}(it)$ as required.

Case 14 is trivial $\neg currPQLive(it_2)$ cannot hold since the pseudoquantifier of the spec of $\mathbb{C}_2$ is trivial.

The case $\neg locterm(it_1) \wedge locterm(it_2)$ is handled similarly.

Finally, we consider the case $\neg locterm(it_1) \wedge \neg locterm(it_2)$. From this we infer $\neg envLive^{m;k}_{\mathcal{A}}(it_1)$ and $\neg envLive^{m;k}_{\mathcal{A}}(it_2)$.

Consider case where both of the first conjuncts of $envLive^{m;k}_{\mathcal{A}}(it_1)$ and $envLive^{m;k}_{\mathcal{A}}(it_2)$ do not hold and assume $envLive^{m;k}_{\mathcal{A}}(it)$ for a contradiction.

Using these assumptions we can infer there exists, $o_1$, $o_2$, $r_1$, $r_2$, $i_1$, $i_2$, where $o_1$ and $o_2$ are atomic obligations, such that:

$$\forall j \geq i_1.\, o_1 \not\leq \theta(w(it_1[j]))(r_1) \bullet \theta(it_1[j])(r_1)$$
$$\forall j \geq i_2.\, o_2 \not\leq \theta(w(it_2[j]))(r_2) \bullet \theta(it_2[j])(r_2)$$

Set $i = \max(i_1, i_2)$, now, because $envLive^{m;k}_{\mathcal{A}}(it)$, there exists $k_1 \geq i$, such that:

$$o_1 \leq \theta(w(it[k_1])) \bullet \theta(it[k_1])$$

This can be rewritten as (since $it$, $it_1$, $it_2$ all agree on share obligations):

$$o_1 \leq \theta(w(it_1[k_1])) \bullet \theta(w(it_2[k_1])) \bullet \theta(it_1[k_1])$$

As $o_1$ is an atomic obligations, $o_1 \leq \theta(w(it_2[k_1]))$. Similarly, ther exists $k_2 \geq i$, such that $o_2 \leq \theta(w(it_1[k_2]))$. Since $k_1, k_2 \geq i$:

$$\forall j \geq k_1.\, o_1 \leq \theta(w(it_2[j]))(r_1)$$
$$\forall j \geq k_2.\, o_2 \leq \theta(w(it_1[j]))(r_2)$$

Assume $\text{lay}(o_2) < \text{lay}(o_1)$. As $o_1$ is chosen using $\neg envLive^{m;k}_{\mathcal{A}}(it_1)$, we can infer:

$$\forall i \in \mathbb{N}, o', r'.\, (\text{lay}(o') \geq \text{lay}(o) \wedge o \neq o') \vee \exists j \geq i.\, o' \not\leq \theta(w(it_1[j]))(r')$$

Set $o' = o_2$, $r' = r_2$ and $i = max(k_1, k_2)$, then $\exists j \geq max(k_1, k_2).\, o_2 \not\leq \theta(w(it_1[j]))(r_2)$, which is a contradiction.
A similar result derives from $\text{lay}(o_2) \geq \text{lay}(o_1)$, therefore, we can conclude $envLive^{m;k}_{\mathcal{A}}(it)$
The other cases proceed similarly to above.

□

### H.2.2 Soundness of While rule

Statement of soundness of the while rule:

**Theorem H.18.** *Given:*

$$\forall \beta \leq \beta_0.\, m; \lambda; \mathcal{A} \vdash \langle P(\beta) \mid L \rangle \xrightarrow{M} \!\!\!\!\rightarrow\, T \tag{18}$$

$$\text{BoundBad}^{\lambda}_{\mathcal{A}}(T, L, M) \tag{19}$$

$$\text{pv}(T, L, M) \cap \text{mod}(\mathbb{C}) = \emptyset \tag{20}$$

$$\forall \beta \leq \beta_0.\, \forall b \in \{0, 1\}.\, m; \lambda; \mathcal{A} \vdash \left\{P(\beta) * (b \doteq\!\Rightarrow T) \wedge \mathbb{B}\right\}\, \mathbb{C}\, \left\{\exists \gamma.\, P(\gamma) \wedge \gamma \leq \beta * (b \doteq\!\Rightarrow \gamma < \beta)\right\} \tag{21}$$

*then:*

$$m; \lambda; \mathcal{A} \vDash \left\{P(\beta_0) * L\right\}\, \texttt{while}(\mathbb{B})\{\mathbb{C}\}\, \left\{\exists \beta.\, P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta_0 \geq \beta\right\}$$

*Proof.*

**Lemma H.19.**

$$\forall st \in [\![\texttt{while}(\mathbb{B})\{\mathbb{C}\}]\!].$$
$$\exists i \cup \{\omega\}, sts \in \mathsf{SchedTrace}^{\omega}, sfin \in \mathsf{SchedTrace}, its \in \mathsf{InstrTrace}^{\omega}, it \in \mathsf{InstrTrace}.$$

$\forall j \leq i.\, sts(j) \in [\![\mathbb{C}]\!]$

- $\forall j \leq i, b \in \{0,1\}.\, \exists \beta \leq \beta_0.$
  $sts(j), its(j) \vDash \{P(\beta) * (b \doteq\!\Rightarrow T) \wedge \mathbb{B}\}\ \{\exists \gamma.\, P(\gamma) \wedge \gamma \leq \beta * (b \doteq\!\Rightarrow \gamma < \beta)\}$
- $st = exec(sts(0)) \oplus \cdots \oplus exec(sts(i)) \oplus sfin$ *where the function exec take the smallest prefix of its argument containing all the local transitions of the trace. This constructs a trace of the while loop from trace of its body and* $\oplus$ *represents trace concatenation.*
- $st, it \vDash \{P(\beta_0) * L\}\ \texttt{while}(\mathbb{B})\{\mathbb{C}\}\ \{\exists \beta.\, P(\beta) * L \wedge \neg\mathbb{B} \wedge \beta_0 \geq \beta\}$
- $\forall j \in \mathbb{N}.\, st[j], (w(it[j]), \theta(it[j])) \vDash^{m}_{\lambda;\mathcal{A}} \exists \alpha.\, L * M(\alpha)$
- $\forall j \in \mathbb{N}, k \geq j, \alpha, \alpha'.$
  $st[j], (w(it[j]), \theta(it[j])) \vDash^{m}_{\lambda;\mathcal{A}} N(\alpha) \wedge$
  $st[k], (w(it[k]), \theta(it[k])) \vDash^{m}_{\lambda;\mathcal{A}} N(\alpha') \Rightarrow \alpha \geq \alpha'$
- $\forall j \in \mathbb{N}, k \geq j, \alpha, \alpha'.$
  $st[j], (w(it[j]), \theta(it[j])) \vDash^{m}_{\lambda;\mathcal{A}} N(\alpha) * T \wedge$
  $st[k], (w(it[k]), \theta(it[k])) \vDash^{m}_{\lambda;\mathcal{A}} N(\alpha') \wedge \overline{T} \Rightarrow \alpha > \alpha'$
- $fair(st) \Rightarrow \forall j \leq i.\, fair(sts(j))$
- $safe(it) \Rightarrow \forall j \leq i.\, safe(its(j))$
- $envLive^{m;k}_{\mathcal{A}}(it) \Rightarrow \forall j \leq i.\, envLive^{m;k}_{\mathcal{A}}(its(j))$

*These properites follow from safety checks in the environmental liveness condition and* $\mathrm{BoundBad}^{\lambda}_{\mathcal{A}}$.

set $st \in [\![\texttt{while}(\mathbb{B})\{\mathbb{C}\}]\!]$ arbitrary. Use lemma to set $i \cup \{\omega\}, sts \in \mathsf{SchedTrace}^{\omega}, sfin \in \mathsf{SchedTrace}, its \in \mathsf{InstrTrace}^{\omega}, it \in \mathsf{InstrTrace}$.

Now from lemma:

$$st, it \vDash \{P(\beta_0) * L\}\ \texttt{while}(\mathbb{B})\{\mathbb{C}\}\ \{\exists \beta.\, P(\beta) * L \wedge \neg\mathbb{B} \wedge \beta_0 \geq \beta\}$$

Suffice to prove implication. Assume $fair(st)$, $safe(it)$ and $\neg locterm(it)$.

Assume $envLive^{m;k}_{\mathcal{A}}(it)$ for a contradiction. From assumptions, we have:

$$\forall j \leq i.\, fair(sts(j))$$
$$\forall j \leq i.\, safe(its(j))$$
$$\forall j \leq i.\, envLive^{m;k}_{\mathcal{A}}(its(j))$$

From the semantics of the triple for the body of the loop, one can infer:

$$\forall j \leq i.\, locterm(its(j))$$

As $M$ is parametrized by an non-strictly decreasing ordinal, if the trace $it$ does not terminate, the ordinal must eventually stop decreasing.

$$\exists f \in \mathbb{N}, \alpha.\, \forall j \geq f.\, st[j], (w(it[j]), \theta(it[j])) \vDash^{m}_{\lambda;\mathcal{A}} M(\alpha)$$

Consider the case where $st[j], (w(it[f]), \theta(it[f])) \vDash^{m}_{\lambda;\mathcal{A}} T$. In this case, T must remain true for the rest of the trace as otherwise, $\alpha$ would decrease. In this case, as all iterations of the loop body terminate, all future iterations of the loop body hold with $b = 1$, so will force a decrease in the loop variant. This can only happy finitely many

times, causing a contradiction.

Otherwise, if $\neg(st[j], (w(it[f]), \theta(it[f])) \vDash^{m}_{\lambda;\mathcal{A}} T)$, the environmental liveness conditions guarantees that if the environment respects the semantic environmental liveness, which ours does, by assumption, the ordinal must eventually decrease, leading to another contradiction.

Therefore, by contradiction, $\neg envLive^{m;k}_{\mathcal{A}}(it)$.

□