

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

EMANUELE D’OSUALDO, Imperial College London
AZADEH FARZAN, University of Toronto
PHILIPPA GARDNER, Imperial College London
JULIAN SUTHERLAND, Imperial College London

We introduce TaDA Live, a separation logic for reasoning compositionally about the termination of blocking fine-grained concurrent programs. The logic contributes several innovations to obtain modular rely/guarantee style reasoning for liveness properties and to blend them with logical atomicity. We illustrate the subtlety of our specifications and reasoning on some paradigmatic examples.

1 INTRODUCTION

Compositional reasoning for fine-grained concurrent programs interacting with shared memory is a fundamental, open research problem. We are beginning to obtain a good understanding of compositional reasoning about *safety properties* of concurrent programs: i.e. if the program terminates and the input satisfies the precondition, then the program does not fault and the result satisfies the postcondition. Following [2, 25], which introduced concurrent separation logic for reasoning compositionally about coarse-grained concurrent programs, there has been a flowering of work on modern concurrent separation logics for reasoning about safety properties of fine-grained concurrent programs: e.g. CAP [8], TaDA [4]; Iris [15]; and FCSL [23]. It is now possible to provide compositional reasoning about safety properties of concurrent programs, with specifications that match the intuitive software interface understood by 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: i.e. something good eventually happens. Examples of progress properties include termination, livelock-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 parameter called ‘fair’ 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. The goal of this paper, is to devise a verification system able to compositionally prove total specifications for *blocking* concurrent programs. Blocking happens when a thread is waiting for an action to be performed by some other thread. This pattern of interaction represents a key challenge for compositional termination arguments.

There has been work on reasoning about progress for synchronisation patterns of coarse-grained language primitives such as primitive locks and communication channels [1, 14, 17]. In this work, all the blocking behaviour of a program is focussed on the use of these primitive constructs. Although verification is challenging even under this assumption, this approach suffers from two drawbacks: first, it makes it difficult to have abstract specifications for high-level blocking components since the specification will need to refer to actions generated by the blocking primitives; second, it does

not generalise to the combination of blocking primitives and blocking that arises from more general busy-waiting patterns.

Consider instead fine-grained concurrent programs where the primitives never block and instead blocking constructs are implemented by sophisticated busy-waiting patterns. For example, high-level blocking synchronisation primitives like locks and channels are implemented on top of non-blocking primitives such as *compare-and-swap*. One way to handle this more general setting is by history-based reasoning [11, 16]: specifications are described using abstract histories and interference is given by a rely/guarantee relation on such histories. This approach 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 verify program specifications that are based on more standard abstract state updates, rather than general histories.

Another natural way of dealing with busy-waiting blocking patterns is to use refinement to show that their behaviour can be replaced by some blocking primitive. With this approach, blocking code can be replaced with some primitive blocking code which implements the same functionality without busy waiting. LiLi [21], the only concurrent separation logic to prove directly termination results for fine-grained concurrency with blocking, adopts this approach. However, to accurately capture the behaviour of the fine-grained code, the abstract code used as specification is non-atomic and represents the termination argument as *behaviour* of the abstract code. Any proof which tries to use the specification in a larger proof would have to reconstruct the termination argument from the structure of the abstract code. This results in duplication of effort and unnecessarily complicated proofs. We discuss this further in Section 6 and Appendix.

We explore a radical new way of representing and reasoning about blocking. Instead of using blocking primitives embedded directly into an operational semantics, we think of blocking behaviour as the *reliance of termination on liveness properties of the environment*. The proof of safety properties usually requires the establishment of resource invariants of the form “always Ψ ” for some condition Ψ : for example, a lock has the invariant property that it is always either locked or unlocked. In contrast, we build our termination arguments on what we call *liveness invariants* of the form “always **eventually** Ψ ”: for example, a lock operation is blocking because it terminates conditionally on the property that the environment will induce traces where the lock is always eventually unlocked.

We introduce TaDA Live, a compositional separation logic for reasoning about the termination of fine-grained concurrent programs. Our view of blocking is the source of all TaDA Live’s innovations:

- *Abstract atomic specifications for blocking operations* which express termination guarantees conditionally on an *environment liveness assumption* of the form “always eventually Ψ ”.
- *Extension of Rely/Guarantee* compositional reasoning to incorporate thread-local liveness invariants, through the introduction of liveness ghost state called *obligations*.
- *Layered liveness invariants* to allow the sound composition of arguments based on mutually dependent liveness invariants. This mechanism is crucial for proving that there is no deadlock.

TaDA Live is fully compositional, obtaining thread-locality and modularity as a simple consequence of using our environment liveness invariants. The specifications leak only the details strictly necessary for expressing termination properties, and abstract atomicity enables direct reasoning about dependencies between abstract actions as opposed to indirect reasoning via the primitive actions.

Our work on TaDA Live builds on TaDA [4], a compositional separation logic for reasoning about safety properties of fine-grained concurrent programs. The proof of soundness of TaDA Live required a number of technical innovations. A substantial re-definition of the model partly influenced by ideas from Iris [15], partly attempting at a more direct refinement-like semantics,

simplifies the understanding and manipulation of the semantics of the specifications. The simplification allowed for a principled extension of the model to include obligations and to express the associated liveness constraints. Inspired by [19], we have introduced a *subjective* composition operation for obligations, to keep track of available liveness assumptions in a thread-local way. Using TaDA Live, we have verified a number of paradigmatic examples including spin lock, CLH lock, our distinguishing client, a blocking counter module and a lock-coupling set. We present here and in Appendix two fine-grained implementations of locks (CLH and spin lock) and a simple client using locks to illustrate the main points of our reasoning principles.

2 TADA LIVE SPECIFICATIONS

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 $\text{lock}(x)$ operation. Only one thread will succeed, leaving the others to wait until the $\text{unlock}(x)$ operation is called by the winning thread.

We use a simple, fine-grained concurrent while language for manipulating shared state. The shared state comprises 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, $\text{CAS}(x, v_1, v_2)$, 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, $\text{FAS}(x, v)$, 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 queues requests, keeping a head and a tail pointer (at 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 have the same safety behaviour, and hence satisfy the same TaDA safety specification.

TaDA specifications combine data abstraction, introduced for concurrent separation logics 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 $\text{lock}(x)$; the environment continues to lock and unlock the lock during this invocation. *Linearizability* [13] is probably the best known technique for describing abstract atomicity. It is a correctness condition for concurrent traces that, when satisfied, enables us to find a sequential trace which has no invocation overlaps and is equivalent to the original concurrent trace. One well-known technique for establishing linearizability is to identify the ‘linearization points’, illustrated by the bullet points in Figure 2, which are steps in the invocation of an operation where the abstract state change becomes visible to the client: with the

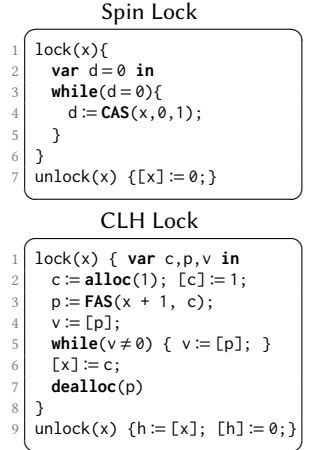


Fig. 1. Two locks.

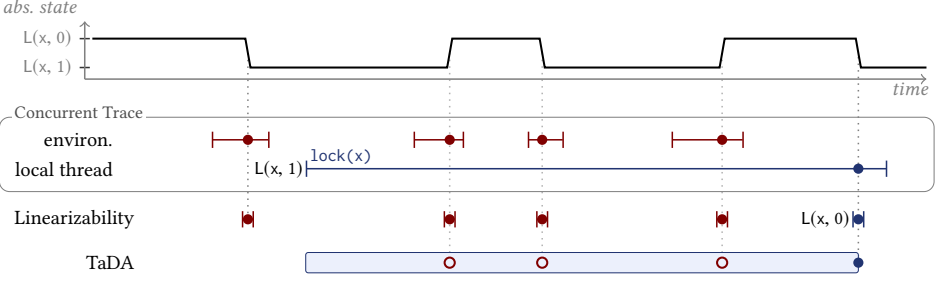


Fig. 2. Linearizability versus TaDA.

spin lock, the linearization point is the successful CAS; with the CLH lock, the linearization point is line 6. The linearization points give rise to a sequential trace where the order of the operations coincides with the order of execution of the corresponding linearization points, in Fig. 2.

One aim of linearizability is to reduce reasoning about concurrent traces of primitive events to reasoning about sequential traces of operation invocations, enabling the use of sequential specifications to describe the operations. In the case of `lock`, the use of sequential specifications is problematic. The linearisation point satisfies the triple $\vdash \{L(x, 0)\} \text{lock}(x) \{L(x, 1)\}$. However, using this triple as a specification of the whole operation does not work. As illustrated in Fig. 2, a lock can also be called when it is locked. Relaxing the triple to $\vdash \{L(x, 0) \vee L(x, 1)\} \text{lock}(x) \{L(x, 1)\}$ is not enough: the same triple holds for a simple assignment $x := 1$; the triple does not express the property that, upon termination of the operation, we can claim that we have acquired the lock.

TaDA safety specifications describe the inherently concurrent behaviour of the `lock` operation accurately. Consider the execution of the abstract atomic lock operation in Fig. 2, labelled TaDA. 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, called the *interference phase* of the call. During this phase, the environment is able to make changes to the abstract state, represented in the figure by \circ , but the local thread can only change the lock from unlocked to locked at the linearization point, represented in the figure by \bullet . The TaDA safety specification for `lock` is the partial *atomic triple*:

$$\vdash \forall l \in \{0, 1\}. \langle L(x, l) \rangle \text{lock}(x) \langle L(x, 1) \wedge l = 0 \rangle \quad (1)$$

The *interference precondition* $\forall l \in \{0, 1\}. \langle L(x, l) \rangle$ describes the interference phase. It states that the environment must preserve the existence of the lock at x but may change the value of l , and the implementation of the lock must tolerate these environmental changes. The *pseudo-quantifier* $\forall l \in \{0, 1\}$ is unusual, behaving like an evolving universal quantifier in that the environment is able to keep changing l over time and behaving like an existential quantifier in that the implementation can assume that the lock always exists with $l \in \{0, 1\}$. The triple (1) states that, if the environment satisfies the interference precondition and the operation terminates, then the implementation guarantees that, just before the the linearization point, the lock must have been available for locking ($l = 0$) and, just afterwards, the lock has been locked by the operation ($L(x, 1)$). Exclusive ownership of the lock after the operation terminates can be derived from the $l = 0$ assertion in the postcondition: just before we locked it, nobody else can claim that they owned the lock.

The partial specification of `unlock` is $\vdash \forall l \in \{1\}. \langle L(x, l) \rangle \text{unlock}(x) \langle L(x, 0) \rangle$. This triple¹ states that, to be used correctly, the `unlock` operation requires the lock to be locked and not

¹We typically omit the pseudo-quantifier in the case where the set is just one element.

changed by the environment during the interference phase; in return, the operation promises to atomically set the lock to be unlocked.

TaDA Live Total Specification. TaDA Live builds on the TaDA specification format. To turn the TaDA triple for lock into a total specification, the termination guarantee must depend on the environment: if the environment decides to hold the lock indefinitely, no lock implementation should allow the lock operation to terminate. A common approach to represent this behaviour, used for example by [21], is to appeal to a notion of *primitive blocking* built into the operational semantics. Intuitively, this associates to a blocking command an *enabledness* condition: that is, a condition \mathbb{B} on the state such that if \mathbb{B} holds then the command can take a step, otherwise not. The injection of this concept of primitive blocking seems artificial in fine-grained concurrency: at the machine level, every command can always take a step. In addition, for fine-grained primitives, the reasoning only needs to assume a *weakly fair* scheduler that promises to eventually execute a step of every active thread. In contrast, for blocking primitives, a *strongly fair* scheduler assumption may be needed (for example, for the distinguishing client discussed later), where the scheduler promises never to produce a trace where a command is infinitely often enabled but never executed. Representing and appealing to these fairness assumptions in the reasoning is tricky. Since fine-grained primitives can be used to implement both blocking constructs with the strongly fair, or weakly fair semantics, we should be able to find a uniform specification format that can abstractly represent all these forms of blocking (primitive or derived) without resorting to a notion of primitive blocking.

We obtain a uniform treatment of blocking by making a fundamentally different choice. We express blocking behaviour as a *liveness condition* on the *environment* during the interference phase of an abstractly atomic operation. The CLH lock operation will terminate under weak fairness, provided that, if the lock is locked by the environment during the interference phase, the environment will *eventually* unlock it. In general, a blocking operation will require an environment that is *live*: it will always eventually bring the abstract state to a *good* state (e.g. unlocked).

In Fig. 3(a) we show a diagram of the evolution of the abstract state induced by a live environment in the interference phase of lock. Note that we do not require the environment to promise to eventually keep the lock always unlocked. Progress towards termination of the lock is guaranteed by the progress measure charted in Fig. 3(b): every time the environment unlocks, the value of l decreases from 1 to 0; when the environment locks, although l increases to 1, the number q of threads in front of us in the queue decreases. One crucial aspect of our specification design is that we do not want to expose the progress argument to the client *unless part of the argument needs to be made by the client*. With CLH, the part of the argument appealing to the queue of threads is completely internal to the implementation of the operation, while the argument for the environment's liveness must be provided by the client (the implementation has no power over this). The TaDA Live total specification of the CLH lock is:

$$\vdash \mathbb{W}l \in \{0, 1\} \rightarrow \{0\}. \langle L(x, l) \rangle \text{lock}(x) \langle L(x, 1) \wedge l = 0 \rangle \quad (2)$$

The interference precondition is $\mathbb{W}l \in \{0, 1\} \rightarrow \{0\}. \langle L(x, l) \rangle$ with the pseudo-quantifier now incorporating the environment liveness condition. As well as stating that the environment can keep changing the lock, the interference precondition also states that if the lock is in a bad state ($l \in \{0, 1\} \setminus \{0\}$) then the environment must always eventually change it to a good state ($l \in \{0\}$). 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: when the lock is unlocked and we are not at the head of the queue, the current head is guaranteed internally

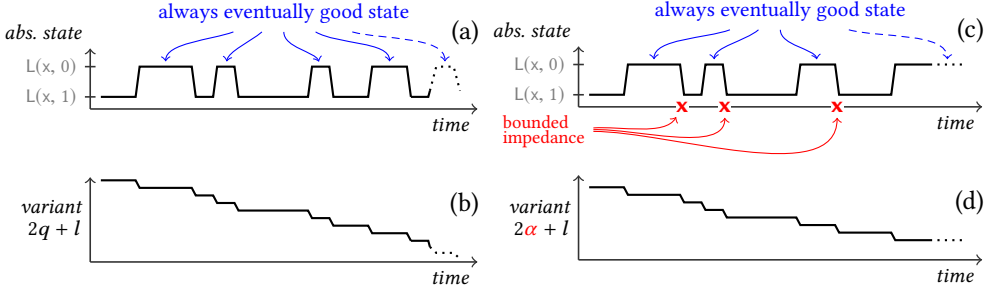


Fig. 3. Live environment (a); measure of progress for CLH lock where q is the number of threads ahead in the queue (b); live environment with bounded impedance (c); measure of progress for spin lock (d).

to acquire the lock and move us one step closer to the head of the queue; if the lock is locked, the implementation can appeal to the assumption provided by the pseudo-quantifier, and argue that eventually the lock is going to be unlocked. We prove this formally in the Appendix.

Now let us consider the spin lock implementation. The spin lock operation cannot promise to terminate just by relying on a live environment. The problem is that when the environment locks the lock, there is no measure of progress that decreases: we are genuinely delayed by this action. We call this effect *impedance*. We conceptualise impedance as a greater *leaking* of the progress argument to the client. In the spin lock example, the whole of the progress argument needs to be provided by the client: the client needs to ensure that the environment will always eventually unlock the lock, and that it will only impede the operation a bounded number of times. To represent this extra *bounded impedance* requirement (depicted in 3(c)), we extend the abstract state of the lock with an ordinal α , an *impedance budget* that strictly decreases when the lock state is set to 1. We arrive at the following TaDA Live specification for spin lock:

$$\forall \phi. \vdash \forall l \in \{0, 1\} \Rightarrow \{0\}, \alpha. \langle L(x, l, \alpha) \wedge \phi(\alpha) < \alpha \rangle \text{lock}(x) \langle L(x, 1, \phi(\alpha)) \wedge l = 0 \rangle \quad (3)$$

The lock is now represented by the predicate assertion $L(x, l, \alpha)$ with ordinal α , which can also be changed by the environment during the interference phase. As well as expressing the dependency on a live environment on l , this triple states that every lock operation consumes the budget α by a non-trivial amount, thus providing a logical measure of progress from good to bad states. The initial value of the budget and the function ϕ from ordinals to ordinals is determined by the client, which must demonstrate that the budget is enough to make all its calls.

While the TaDA Live specification of `unlock` for the CLH lock is the same as that for TaDA, the specification for spin lock needs to incorporate the ordinals: $\vdash \langle L(x, 1, \alpha) \rangle \text{unlock}(x) \langle L(x, 0, \alpha) \rangle$. The impedance budget α is preserved by `unlock`. This encodes the fact that `unlock` does not impede the other operations, but also that by unlocking we cannot increase the budget. By combining these assumptions about the budget (it decreases when locking, stays constant when unlocking), the spin lock implementation can conclude termination using the progress measure in 3(d). Crucially, for spin lock, the *whole* of the progress argument is provided by (and thus visible to) the client.

The impedance budget technique was first introduced to concurrent separation logics for non-blocking operations using Total TaDA [5]. Here, we smoothly integrate ordinals into TaDA Live that fully supports blocking. It is interesting to note that the CLH and spin lock implementations behave as a primitive blocking lock under the strongly fair and weakly fair scheduler assumptions respectively. This means that we can do all our reasoning uniformly relying on weak fairness, and still handle a mixture of abstract operations with the different termination guarantees. Additionally, the logic is not sensitive to the fact that these abstract operations are seen as primitive or as implemented by more primitive means.

3 TADA LIVE VERIFICATION

We show how TaDA Live achieves compositional verification, using a distinguishing client which terminates with CLH locks but not with spin locks.

Example 3.1 (Distinguishing client). The distinguishing client is:

lock(x);		var d=false in
[done] := true;		while(¬d){
unlock(x);		lock(x); d := [done]; unlock(x);
		}

Under weak fairness, when x is a spin lock, this client program does not always terminate. It is possible for the lock invocation of the left thread to be scheduled infinitely often but always in a state in which 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. In contrast, when x is a CLH lock, this client program is guaranteed to terminate: a fair scheduler 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*.

We show that the distinguishing client terminates with the CLH lock, by proving the Hoare triple $\vdash \{L(x, 0) * \text{done} \mapsto \text{false}\} \mathbb{C}_\ell \parallel \mathbb{C}_r \{True\}$, where \mathbb{C}_ℓ and \mathbb{C}_r are the left and right threads of the example, respectively. Since our triples are total, this triple immediately guarantees termination of the program. Our overall argument is as follows. The CLH specification guarantees termination of a call to `lock(x)` if the lock is always eventually unlocked by the environment. This is intuitively true for both threads: they always unlock the lock after having acquired it. The call to `lock(x)` will therefore terminate in both threads. The only other potentially non-terminating operation is the while loop in the right thread. The loop is implementing a busy-wait pattern on `done`, and needs the help of the left thread to terminate. We will be able to prove that since `done` is going to be eventually set to `true` (and never reset to `false`), the loop will terminate.

Let us formalise the argument in TaDA Live. The two threads of the distinguishing client both access the lock x and the heap cell `done`. In TaDA, such shared resource is represented by a *shared region*, which comprises an abstraction of the contents of the shared heap and a protocol for coordinated interference on the region. We use *region assertions* to describe such shared regions, a technique first invented in the CAP logic [8] and now used by many modern concurrent separation logics, albeit in slightly different ways. In our example, we use the region assertion $c_r(x, \text{done}, l, d)$, where c is a region type, r is a region identifier, and (x, done, l, d) is the abstract state of the region where l and d denote the abstract values associated with addresses x and `done` respectively. Although in this case the abstraction is not hiding any detail, since both l and d are visible, in general the abstraction of the contents is an essential mechanism for reasoning about *abstract* atomicity.

The region assertion $c_r(x, \text{done}, l, d)$ is substantially different from the assertion $L(x, l) * \text{done} \mapsto d$ in that it declares a shared region that can be accessed by many threads rather than resource owned by one thread. The region is affected by concurrent interference as specified by an associated interference protocol \mathcal{T}_c . For example, here we want to formalise the fact that only the left thread will ever change the `done` flag, and at most once from `false` to `true`. To do so, we introduce a form of ghost state called *guard* \mathbf{D} , which gives exclusive permission to update the `done` flag. Formally, guards (probably first introduced in deny-guarantee reasoning [9]) form a partial commutative monoid (PCM), where in this case $\mathbf{D} \bullet \mathbf{D}$ is undefined to capture exclusive permission: if a thread owns \mathbf{D} nobody else can own it at the same time. To link \mathbf{D} with the ability to change `done`, the

protocol \mathcal{T}_c has the transition $\mathbf{D} : d = \text{false} \rightsquigarrow d = \text{true}$, and no other transition that can modify d , to encode the fact that, once set, the flag cannot be reset to false . Using guards, one can only express what *may* happen due to interference. In blocking code, such as our example, we need to formalise the fact that something *must* happen due to interference: e.g. the eventual assignment $[\text{done}] := \text{true}$ must happen. To do this, TaDA Live introduces a new kind of ghost state called *obligations*. They also form a PCM, but their semantics encodes liveness invariants: an environment owning obligation O will guarantee that eventually O will be fulfilled. In our example, we declare an obligation \mathbf{D} (reusing the symbol we used for the corresponding guard) which, if owned, represents the *responsibility* of setting done . Symmetrically, if \mathbf{D} is known to be owned by the environment, it allows us to *assume* that done will eventually be set to true . This obligation invariant is again expressed in the protocol by extending the previous transition to $\mathbf{D} : (d = \text{false}, \mathbf{D}) \rightsquigarrow (d = \text{true}, \mathbf{0})$ which states that whichever thread is updating d from false to true will go from owning the responsibility to do that (owning obligation \mathbf{D}) to not having the responsibility ($\mathbf{0}$). Repeating the same story for the lock, we obtain the full protocol \mathcal{T}_c :

$$\mathbf{0} : ((0, d), \mathbf{0}) \rightsquigarrow ((1, d), \mathbf{K}) \quad (4)$$

$$\mathbf{K} : ((1, d), \mathbf{K}) \rightsquigarrow ((0, d), \mathbf{0}) \quad (5)$$

$$\mathbf{D} : ((l, \text{false}), \mathbf{D}) \rightsquigarrow ((l, \text{true}), \mathbf{0}) \quad (6)$$

where the transitions describe how the abstract state² (l, d) and the obligations are affected by the update. We introduce the guard \mathbf{K} and obligation \mathbf{K} (with $\mathbf{K} \bullet \mathbf{K}$ undefined) to represent the ability and the responsibility of unlocking the lock, respectively. Transition (4) allows anybody to acquire the lock (the $\mathbf{0}$ guard means no permission needed), with the effect of obtaining the obligation \mathbf{K} .

When considering blocking code compositionally, one needs to represent locally facts about what are the responsibilities of the environment. TaDA Live introduces a way to do so, inspired by the subjective separation of [19], in the form of two assertions: $[\mathbf{K}]_r^L$, representing local ownership of obligation \mathbf{K} , and $[\mathbf{K}]_r^E$, indicating that the environment owns (at least) \mathbf{K} . What makes these assertions interesting is the way they compose: that is, $[\mathbf{D}]_r^L \Leftrightarrow [\mathbf{D}]_r^L * [\mathbf{D}]_r^E$. If we start with local obligation \mathbf{D} and we want to fork into two threads, we use $*$ to give responsibility of \mathbf{D} to one thread and knowledge that the environment has this responsibility to the other.

When assertions involve shared regions, they may be invalidated by environment interference. For example, the assertion $c_r(x, \text{done}, l, \text{false})$ can be invalidated by an environment executing the allowed transition (6). Assertions that cannot be invalidated by the environment are called *stable*. For example, the assertion $c_r(x, \text{done}, l, \text{false}) * [\mathbf{D}]_r$ is stable since the environment cannot own the \mathbf{D} guard necessary to be able to perform transition (6). Since the interference protocol similarly specifies how obligations change as one updates a shared region, environment obligation assertions are also subjected to interference. For example, $\exists l, d. c_r(x, \text{done}, l, d) * [\mathbf{K}]_r^E$ is not stable, as the environment may execute (5) loosing \mathbf{K} . A stable version of the assertion is³ $\exists l, d. c_r(x, \text{done}, l, d) * l = 1 \Rightarrow [\mathbf{K}]_r^E$ which states that the environment owns \mathbf{K} only when the lock is locked. In TaDA Live, the pre- and postconditions of Hoare triples must be stable.

To complete the definition of shared region c , we link its abstract state to the actual heap content that it encapsulates using the *region interpretation*:

$$\begin{aligned} \mathcal{I}(c_r(x, \text{done}, l, d)) \triangleq & L(x, l) * \text{done} \mapsto d * (([\mathbf{K}]_r^L \wedge l = 0) \vee ([\mathbf{K}]_r^E \wedge l = 1)) * \\ & * (([\mathbf{D}]_r^L \wedge d) \vee ([\mathbf{D}]_r^E \wedge \neg d)) \end{aligned}$$

²Strictly speaking, the abstract state is (x, done, l, d) . We simplify, just giving the part of the abstract state that is changed by \mathcal{T}_c .

³When P is pure, $P \Rightarrow Q$ is syntactic sugar for $(\neg P \wedge \text{emp}) \vee Q$.

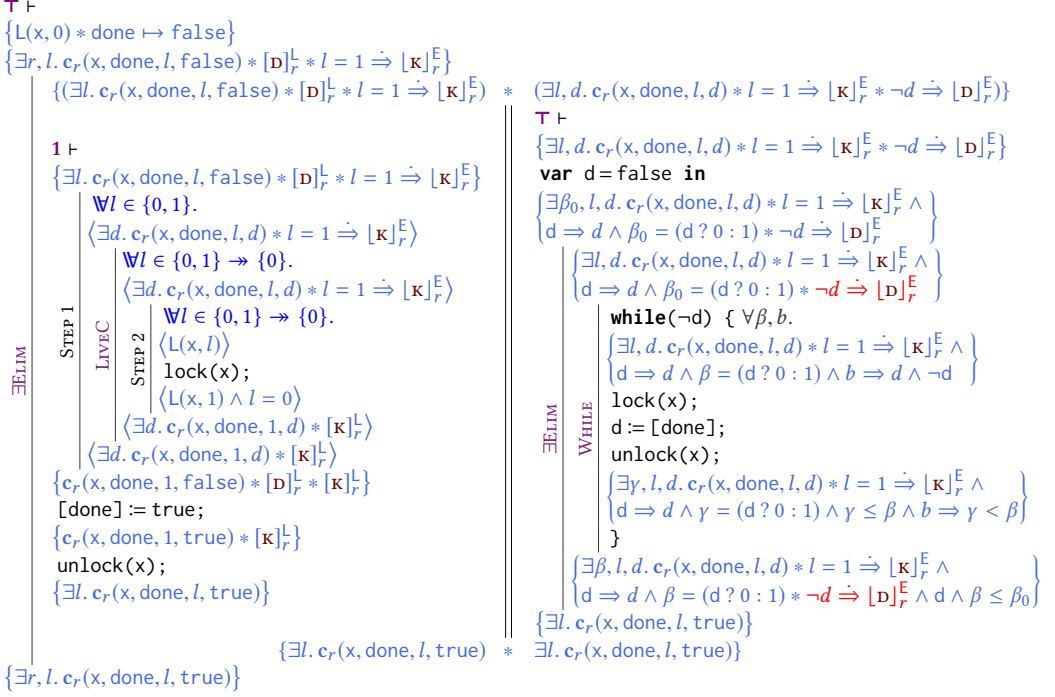


Fig. 4. Proof Sketch of the Distinguishing Client.

This assertion describes a portion of the heap being shared (the lock at x and the cell done) and the linking of the ghost state (the guards and obligations) with the abstract state. The assertion $[K]_r^L$ is an abbreviation for $[K]_r * [K]_r^L$, which indicates local ownership of the guard K and obligation K . The interpretation of a region establishes the invariant that, when $l = 0$, the guard and obligation K will be “owned” by the region (and by no thread as a consequence). When $l = 1$, the $[K]_r^E$ assertion indicates that the obligation K is owned by some thread. Similar links are established between the value of d and D .

We have specified how threads interact abstractly on the shared region. Now we describe the proof sketch of the distinguishing client given in Figure 4, using the simplified TaDA Live rules given in Fig. 6. The first step is to weaken the initial precondition to the shared region c using the consequence rule. In TaDA and other modern separation logics such as Iris, implication is generalised to the *viewshift* construct (\Rightarrow) from [6], which allows the consistent update of ghost information. Here, the precondition can be augmented with the ghost state necessary establish the region invariant with $l = 0, d = \text{false}$, and thus allocate the region $\exists r. c_r(x, \text{done}, 0, \text{false})$. To obtain a stable assertion, we need to weaken the information on l , by existentially quantifying it. We must obtain the obligation D locally since $[D]_r^E$ must be in the region invariant as $d = \text{false}$. We also duplicate the assertion $[K]_r^E$ from the invariant, because it is needed in the proof. This assertion is however not stable on its own and needs to be weakened to $l = 1 \dot{\Rightarrow} [K]_r^E$ which is stable.

Next, we apply the **PAR** rule, which requires an assertion of the form $P_1 * P_2$ for stable P_1 and P_2 . Here, with another application of consequence, we split the local information, giving to both threads the stable assertion $l = 1 \dot{\Rightarrow} [K]_r^E$. Since we assign responsibility of D to the left thread with $[D]_r^L$, we separate $[D]_r^E$ to give it to the right thread. Again, we weaken it to $\neg d \dot{\Rightarrow} [D]_r^E$, which is stable.

Let us focus on the left-hand thread first. The difficult step is the execution of the first instruction, since this is the only potentially non-terminating instruction of the thread. **STEP 1** follows a standard TaDA proof pattern:

$$\frac{\frac{\frac{1 \vdash \mathbb{W}l \in \{0, 1\}. \langle \exists d. c_r(x, \text{done}, l, d) * l = 1 \Rightarrow [\kappa]_r^E \rangle \text{lock}(x) \langle \exists d. c_r(x, \text{done}, 1, d) * [\kappa]_r^L \rangle}{1 \vdash \mathbb{W}l \in \{0, 1\}. \langle c_r(x, \text{done}, l, \text{false}) * [\mathbb{D}]_r^L * l = 1 \Rightarrow [\kappa]_r^E \rangle \text{lock}(x) \langle c_r(x, \text{done}, 1, \text{false}) * [\mathbb{D}]_r^L * [\kappa]_r^L \rangle} \text{FRAMEA}}{1 \vdash \langle \exists l. c_r(x, \text{done}, l, \text{false}) * [\mathbb{D}]_r^L * l = 1 \Rightarrow [\kappa]_r^E \rangle \text{lock}(x) \langle c_r(x, \text{done}, 1, \text{false}) * [\mathbb{D}]_r^L * [\kappa]_r^L \rangle} \text{A}\exists\text{ELIM}}{1 \vdash \langle \exists l. c_r(x, \text{done}, l, \text{false}) * [\mathbb{D}]_r^L * l = 1 \Rightarrow [\kappa]_r^E \rangle \text{lock}(x) \{c_r(x, \text{done}, 1, \text{false}) * [\mathbb{D}]_r^L * [\kappa]_r^L\}} \text{ATOMW}$$

The combination of **A** \exists **E****L****I****M** and **A****T****O****M****W** simply states that if we can prove a command performs an update atomically, and the pre- and postconditions are stable, then we can prove the command also performs the update non-atomically.

STEP 2, in Appendix, lifts the specification of `lock` to where it is being used, which in this case is the region `c`. **STEP 2** uses standard TaDA rules, adapted minorly to propagate the liveness assumption of the pseudo-quantifier.

STEP 1 and **STEP 2** can almost be put together, except that there is a gap. **STEP 2** has the liveness assumption, $\mathbb{W}l \in \{0, 1\} \rightarrow \{0\}$, promising termination conditionally on the environment liveness, whereas **STEP 1** has no liveness assumption, $\mathbb{W}l \in \{0, 1\}$, requiring unconditional termination of the operation. To reconcile the two specifications, TaDA Live provides the **L****I****V****E****C** rule, which states that if in the current context we can prove the environment does indeed satisfy the liveness assumption of the pseudo-quantifier, then the liveness assumption can be removed from the specification. The (simplified) **L****I****V****E****C** rule is:

$$\frac{m \vdash \exists x \in X. P(x) \xrightarrow{M} \exists x' \in X'. P(x') \quad m \vdash \mathbb{W}x \in X \rightarrow X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle}{m \vdash \mathbb{W}x \in X. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle} \text{LIVEC}$$

This rule enables us to join **STEP 1** and **STEP 2** together, discharging the environment assumption given by the pseudo-quantifier, as long as the crucial *environment liveness condition* given by the first premise is satisfied. For our example, the instantiation of the environment liveness is:

$$1 \vdash \exists l \in \{0, 1\}. \exists d. c_r(x, \text{done}, l, d) * l = 1 \Rightarrow [\kappa]_r^E \rightarrow \exists d. c_r(x, \text{done}, 0, d)$$

where M is omitted because it is straightforward. Intuitively, the condition requires that, when we have a lock, then either the lock is in the “target” state ($l = 0$), or there is an obligation in the environment which, when fulfilled, takes us to a target state. In our example, thanks to transition (5) and the assertion $l = 1 \Rightarrow [\kappa]_r^E$, the κ is such an obligation. The condition implies that the target states are always eventually reached (under the assumption that the environment always eventually fulfils its obligations).

The environment liveness condition is used in the **L****I****V****E****C** and **W****H****I****L****E** rules. Its general form is $m \vdash L \xrightarrow{M} T$, where L is an invariant assertion that determines all possible states of the resource, T specifies the target states that are the ones we want to prove are always eventually reached, and M is a well-founded measure of environment progress. The condition requires that, at any point in a trace where every state satisfies L , either: (i) the state is in T ; or (ii) the measure is not increased by the next transition and there exists some obligation in the environment, the fulfilment of which would strictly decrease the measure. This implies that, in infinite traces where L is always true and the environment obligations are always eventually fulfilled, the target states are *always eventually* reached. Note that, when transitions terminate in T , the progress measure can be reset. This means that that the environment liveness condition does not ensure that eventually T is always true.

We now focus on the right-hand thread. The difficult step is the application of the **W****H****I****L****E** rule. In addition to the loop invariant, a standard technique for proving non-blocking termination is to use a well-founded *variant* to decrease at every iteration. For blocking loops like this one, however, a

local variant is impossible to find since there will inevitably be iterations where there is absolutely no measurable progress. The (simplified) **WHILE** rule for TaDA Live is:

$$\frac{\begin{array}{l} m \vdash L \xrightarrow{M} T \quad \text{non-incr}(L, M) \\ \forall \beta \leq \beta_0. \vdash \{P(\beta) \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma. P(\gamma) \wedge \gamma \leq \beta\} \\ \forall \beta \leq \beta_0. \vdash \{P(\beta) * T \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma. P(\gamma) \wedge \gamma < \beta\} \end{array}}{\vdash \{P(\beta_0) * L\} \mathbf{while}(\mathbb{B})\{\mathbb{C}\} \{\exists \beta. P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta \leq \beta_0\}} \text{WHILE}$$

Some aspects of the standard **WHILE** rule are still present: P is the loop invariant, parametrised by an ordinal-valued variant β . The two triples in the premises require the verification of the loop body in two situations: when the iteration starts from a blocked state, in which case the variant β is only required not to increase ($\gamma \leq \beta$); when the iteration starts from an unblocked state (T), in which case the variant has to decrease strictly ($\gamma < \beta$). Notice that the L has been framed off, which means it will hold constantly for the duration of the loop. In addition, the rule requires the proof of an environment liveness condition stating that the unblocked states T are always eventually reached. By itself, this is not enough to build a termination argument: the states T could be always eventually reached, but the loop could be always scheduled so that it will never witness T being reached. Thus, the while loop also needs T to *eventually always* being true. The conjunction of the environment liveness condition with the $\text{non-incr}(L, M)$ judgement achieves this since $\text{non-incr}(L, M)$ states that the environment progress measure M can never increase whilst L holds. This ensures that the measure cannot be reset when T is reached and, in the worst case, will eventually reach 0 in which case T will be true until the next iteration.

The **WHILE** rule given in Fig. 6 and used in Fig. 4 combines the two triples in the premises into one for convenience:

$$\forall \beta \leq \beta_0. \forall b \in \text{Bool}. \vdash \{P(\beta) * (b \dot{\Rightarrow} T) \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma. P(\gamma) \wedge \gamma \leq \beta * (b \dot{\Rightarrow} \gamma < \beta)\}$$

In our example, T is $d = \text{false}$. The environment liveness condition asks us to find, in states with $d = \text{true}$, an obligation in the environment ($\lfloor \mathbf{D} \rfloor_r^E$ given by L , displayed in red in Fig. 4) which, when fulfilled, takes us closer to T . Here \mathbf{D} can only be fulfilled by executing (6) which immediately makes T hold. We also have to show that eventually the environment cannot invalidate T , which in this case holds trivially as no transitions in the protocol allow setting $d = \text{true}$ from $d = \text{false}$. Finally, the proof of the loop body is a routine derivation. Notice that the proof of the $\text{lock}(x)$ call can reuse the derivation we had for the left-hand thread, followed by an application of **FRAMEH** to match the loop invariant.

Layers. We have outlined the termination argument for the distinguishing client but, unless refined, this line of argument can lead to unsound circularity. Consider a variant of the distinguishing client, where the lock is acquired outside of the loop in the right thread: $\mathbb{C}'_r = \text{lock}(x); \mathbf{while}(\neg d)\{d := [\text{done}]\}; \text{unlock}(x)$. The program $\mathbb{C}_\ell \parallel \mathbb{C}'_r$ is non-terminating even when using a CLH lock. To see where our argument goes wrong, reconsider the application of **LIVEC**. There we appealed to the environment responsibility of fulfilling \mathbf{k} whilst we are continuously holding (and thus not fulfilling) \mathbf{D} . If the environment also relies on our fulfilment of \mathbf{D} to fulfil \mathbf{k} , as is the case for $\mathbb{C}_\ell \parallel \mathbb{C}'_r$, we have an unsound circular argument. Our solution is to associate layers with obligations, i.e. elements of a well-founded partial order using a function lay with the intuition that, if $\text{lay}(O_1) < \text{lay}(O_2)$, then local fulfilment of O_2 can depend on the environment's fulfilment of O_1 . Using layers, we can refine our argument: the environment liveness condition can only appeal to progress using obligations known to be held by the environment, *if they have layer strictly lower than any obligation we may be holding*. In the proof of the left thread, this requires $\text{lay}(\mathbf{k}) < \text{lay}(\mathbf{D})$ since we appeal to the environment responsibility to fulfil \mathbf{k} while holding \mathbf{D} . With

$\mathbb{C}_\ell \parallel \mathbb{C}_r$, the same constraint is fine to complete the termination argument of the loop and the lock operation in the body of the loop. With $\mathbb{C}_\ell \parallel \mathbb{C}'_r$, the proof of \mathbb{C}'_r requires $\text{lay}(\mathbf{D}) < \text{lay}(\mathbf{K})$ which leads to a contradiction. If a proof requires to appeal to fulfilment of obligations of layer k we say the proof assumes k is live.

To conclude our proof for $\mathbb{C}_\ell \parallel \mathbb{C}_r$, the **PAR** rule checks that, in the postcondition of each thread, there is no pending obligation of layers on which the other thread might depend. In our example, this is trivially satisfied as both postconditions do not hold obligations. If we had forgotten to unlock x in the left thread for example, we would obtain $\lfloor \mathbf{K} \rfloor_r^L$ in the postcondition. Since we use a classical interpretation of separation, it is not possible to use consequence to remove the local obligation without fulfilling it. The **PAR** rule would detect the problem by seeing that the layer of some obligation in the postcondition of the left thread, is lower or equal than the layers assumed live by the proof of the right thread. This information is stored in the context of the judgement (i.e. what is on the left of the turnstile) as a layer indicating an upper-bound on the layers that may be assumed live by a proof of the judgement. Here $\mathbf{0} = \text{lay}(\mathbf{K}) < \text{lay}(\mathbf{D}) = \mathbf{1}$. The **PAR** rule would see that the right thread may use liveness of both \mathbf{D} and \mathbf{K} , since the layer in the context is \mathbf{T} , so it requires neither are owned by the left thread at its postcondition.

The layer in the context is also used crucially when framing: when we apply any of the two frame rules, the frame can only mention obligations of layer higher or equal than the one in the context. This way we don’t forget that we are continuously holding responsibilities at that layer. This is used in **STEP 1** of Fig. 4 when we frame $\lfloor \mathbf{D} \rfloor_r^L$ as part of the left proof, which is valid since $\text{lay}(\mathbf{D}) = \mathbf{1}$. It is this application of **FRAMEA** that forced the triple for the left thread to have $\mathbf{1}$ in the context instead of the less restrictive \mathbf{T} . In the simplified **WHILE** rule above, we omitted that the loop invariant cannot hold obligations that we may need to invoke in the environment liveness condition.

Spin Lock. Where does the argument fail, if instead of CLH locks we used spin locks? The crucial difference is that spin locks have the impedance budget given by ordinal α in their abstract state. To start the proof, we need to choose an ordinal for the lock and show that, every time we call lock, the ordinal will be high enough for us to make it strictly decrease. Since there is no bound on the number of times lock is going to be called, this initial ordinal cannot be found.

4 THE TADA LIVE PROGRAM LOGIC

This section summarises the formal definitions needed to understand the TaDA Live proof system; see Appendix for details. Our commands come from a standard heap-manipulating While language with compare-and-swap (**CAS**) and fetch-and-set (**FAS**) primitives. For simplicity, our function definitions are not recursive; all unbounded behaviour is expressed using **while**.

The TaDA Live assertions, in Fig. 5, are built from the standard *classical* connectives and quantifiers of separation logic,⁴ TaDA region and guard assertions, and new TaDA Live obligation and layer assertions. We assume some basic infinite sets: *region types*, $\text{RType} \ni \mathbf{t}$; *region identifiers*, $\text{RId} \ni r$; *guards*, $\text{Guard} \ni G$; *levels*, $\text{Lvl} \triangleq \mathbb{N} \ni \lambda$; and *abstract states*, $\text{AState} \ni a, b$, including sets and lists of values. Levels are a technical device, also used in e.g. TaDA and Iris, to prevent regions being opened up twice. We also assume a user-supplied, well-founded partial order of layers, $(\mathcal{L}, \leq, \mathbf{T}, \perp)$, with $k_1 < k_2 \triangleq k_1 \leq k_2 \wedge k_2 \not\leq k_1$.

⁴TaDA interprets the separating conjunction intuitionistically. With TaDA Live, we interpret it classically in order to not lose information about the obligations.

$$\begin{aligned}
P ::= & \mathbb{B} \mid \exists x. P \mid \mathbb{E} \in X \mid \neg P \mid P \wedge Q \mid \text{emp} \mid P * Q \mid \mathbb{E} \mapsto \mathbb{E} \mid \mathfrak{t}_r^\lambda(\mathbb{E}) \mid r \Rightarrow d \\
& \mid \lceil \mathbb{E} \rceil_r \mid \lfloor \mathbb{E} \rfloor_r^L \mid \lfloor \mathbb{E} \rfloor_r^E \mid \text{emp}_{\text{Ob}}^R \mid r \triangleright m \\
d ::= & \blacklozenge \mid \blacklozenge \mid (\mathbb{E}, \mathbb{E}) \qquad \mathfrak{t} \in \text{RType}, \lambda \in \text{Lvl}, r \in \text{Rld} \cup \text{LVar}, R \subseteq \text{Rld}, m \in \mathcal{L}.
\end{aligned}$$

Fig. 5. Syntax of Assertions. Logical expressions, \mathbb{E} , and logical boolean expressions, \mathbb{B} , are standard.

Assertions and their models are built around partial commutative monoids.⁵ Heaps form a cancellative PCM with disjoint union. Guards provide support for custom auxiliary ghost state. They give rise to a *guard algebra* $(\text{Grd}, \bullet, \{\mathbf{0}\})$, for $\text{Grd} \subseteq \text{Guard}$, which is a PCM specified by the user of the logic. Similarly to guards, obligations represent ghost state for describing the *liveness invariants*. They form an *obligation algebra* and are associated with layers.

Definition 4.1 (Obligation Algebras). TaDA Live is parametrised by a *layered obligation structure*: that is, a pair $(\text{Oblig}, \text{lay})$ where $\text{Oblig} \subseteq \text{Guard}$ and $\text{lay}: \text{Oblig} \rightarrow \mathcal{L}$ such that $\forall O \in \text{Oblig}. \perp < \text{lay}(O) \leq \top$. A *obligation algebra* is a cancellative⁶ guard algebra $(\text{Obl}, \bullet, \{\mathbf{0}\})$ where $\text{Obl} \subseteq \text{Oblig}$ and $\forall O_1, O_2 \in \text{Obl}. O_1 \sqsubseteq O_2 \Rightarrow \text{lay}(O_1) \geq \text{lay}(O_2)$. The set $\text{AObl} \subseteq \text{Oblig}$ is a subset of obligations that we call *atoms*. For each obligation algebra Obl ,

$$\text{AObl} \cap \text{Obl} = \{O \in \text{Obl} \mid \forall O_1, O_2 \in \text{Obl}. O \sqsubseteq O_1 \bullet O_2 \Rightarrow O \sqsubseteq O_1 \vee O \sqsubseteq O_2\}$$

In practice, obligation algebras are often constructed from some basic set of atoms (e.g. the \mathbf{k} and \mathbf{d} of Fig. 4), to which we assign some layers, and then extend the layers to the compositions of atoms by taking the minimum layer of the composed atoms (e.g. since $\text{lay}(\mathbf{k}) < \text{lay}(\mathbf{d})$, we can set $\text{lay}(\mathbf{k} \bullet \mathbf{d}) = \text{lay}(\mathbf{k})$).

We summarise the intuitive meaning of our assertions. Their models and satisfaction relation are given in the Appendix.

- TaDA *region assertion* $\mathfrak{t}_r^\lambda(a)$ asserts the existence of a shared region with type \mathfrak{t} , identity r , level λ and abstract state a . Region assertions represent shared resources and, hence, are duplicable. We have $\vdash \mathfrak{t}_r^\lambda(a) \Leftrightarrow \mathfrak{t}_r^\lambda(a) * \mathfrak{t}_r^\lambda(a)$.
- TaDA *atomic tracking assertion* $r \Rightarrow \blacklozenge$ gives permission to perform a single atomic change of the state of region r . Once the change is performed, the assertion becomes $r \Rightarrow (a_1, a_2)$ recording the abstract states just before and after the change (the linearization point). The assertion $r \Rightarrow \blacklozenge$ asserts that the environment has the permission to do the atomic update. We have $\vdash r \Rightarrow \blacklozenge * r \Rightarrow \blacklozenge \Rightarrow \text{False}$, and $\vdash r \Rightarrow \blacklozenge \Leftrightarrow (r \Rightarrow \blacklozenge * r \Rightarrow \blacklozenge)$.
- TaDA *guard assertion* $\lceil G \rceil_r$ asserts that the guard G is held locally. Guard composition is reflected by separation: $\vdash \lceil G_1 \bullet G_2 \rceil_r \Leftrightarrow \lceil G_1 \rceil_r * \lceil G_2 \rceil_r$.
- TaDA Live *local obligation assertion* $\lfloor O \rfloor_r^L$ asserts that obligation O is held locally. We have $\vdash \lfloor O_1 \bullet O_2 \rfloor_r^L \Leftrightarrow \lfloor O_1 \rfloor_r^L * \lfloor O_2 \rfloor_r^L$. Separating conjunction is interpreted classically precisely so that we do not loose local obligation information: that is, $\vdash \lfloor O \rfloor_r^L \not\Rightarrow \text{emp}$. It is often useful to use the same guard algebra for guards and obligations. We write $\lfloor O \rfloor_r^L \triangleq \lfloor O \rfloor_r * \lfloor O \rfloor_r^L$.
- TaDA Live *environment obligation assertion* $\lfloor O \rfloor_r^E$ asserts that O is held by the environment: $\vdash \lfloor O_1 \bullet O_2 \rfloor_r^E \Leftrightarrow \lfloor O_1 \rfloor_r^E * \lfloor O_2 \rfloor_r^E$. Unlike for local obligations, it is possible to loose this information, $\vdash \lfloor O \rfloor_r^E \Rightarrow \text{emp}$, because we not need to keep track of the full obligations held by

⁵A (multi-unit) *partial commutative monoid* (PCM) is a tuple (X, \bullet, E) comprising a set X , an associative, commutative binary *partial* composition operation $\bullet: X \times X \rightarrow X$ and a set of unit elements E , such that $\forall x \in X. \exists e \in E. x \bullet e = x$. For $x, y \in X$, write $x \# y$ if $x \bullet y \neq \perp$ and $x \sqsubseteq y$ if $\exists x_1. y = x \bullet x_1$. A PCM is *cancellative* when, for any $x_1, x_2, x_3 \in X$, if $x_1 \bullet x_2 = x_1 \bullet x_3$ then $x_2 = x_3$.

⁶Cancellativity is a simplifying assumption. Although it can be lifted, we found no evidence that non-cancellative obligation algebras are needed in proofs.

the environment, just a lower bound. The composition of environment and local obligation assertions is subtle, inspired by the subjective separation of [19]. The existence of the local obligation can be recorded in a frame: $\vdash [O]_r^L \Leftrightarrow [O]_r^L * [O]_r^E$. We also have the derived law $\vdash [O_1 \bullet O_2]_r^L \Leftrightarrow ([O_1]_r^L * [O_2]_r^E) * ([O_1]_r^E * [O_2]_r^L)$, giving knowledge to each thread of the obligations delegated to the other.

- TaDA Live *empty obligation assertion* emp_{Ob}^R asserts that no obligation is locally held for regions with identifiers in R .
- TaDA Live *layer assertion* $r \triangleright m$ asserts that the layer of the obligations held locally for region with identifier r is greater or equal than m . We often use notation such as $r \triangleright m \leq m'$ to denote $r \triangleright m \wedge m \leq m'$.

Each region type \mathbf{t} is associated with an *interference protocol* $\mathcal{T}_{\mathbf{t}}$ and an *interpretation* $\mathcal{I}(\mathbf{t}_r^\lambda(a))$. The interference protocol is a function $\mathcal{T}_{\mathbf{t}}: \text{Guard} \rightarrow \wp((\text{AState} \times \text{Oblig}) \times (\text{AState} \times \text{Oblig}))$ that associates to each guard a set of transitions between abstract states of the region and obligations. $\mathcal{T}_{\mathbf{t}}(G)$ represent the allowed updates to the region, given ownership of the guard G . Every function $\mathcal{T}_{\mathbf{t}}$ is required to satisfy three properties:

- monotonicity in the guards: $\forall G_1, G_2. G_1 \sqsubseteq G_2 \Rightarrow \mathcal{T}_{\mathbf{t}}(G_1) \subseteq \mathcal{T}_{\mathbf{t}}(G_2)$;
- reflexivity: $((a, \mathbf{0}_{\mathbf{t}}), (a, \mathbf{0}_{\mathbf{t}})) \in \mathcal{T}_{\mathbf{t}}(\mathbf{0}_{\mathbf{t}})$ for all $a \in \text{AState}$;
- closure under obligation frames: $\forall O_1, O_2, O$, if $((a_1, O_1), (a_2, O_2)) \in \mathcal{T}_{\mathbf{t}}(G)$ and $O_1 \# O$ and $O_2 \# O$ then $((a_1, O_1 \bullet_{\mathbf{t}} O), (a_2, O_2 \bullet_{\mathbf{t}} O)) \in \mathcal{T}_{\mathbf{t}}(G)$.

The interpretation of a region, $\mathbf{t}_r^\lambda(a)$, is an assertion which describes the resource that is being shared by the region: that is, $\mathcal{I}(\mathbf{t}_r^\lambda(a)) = P$. The assertion P is required to be stable and only own local obligations of the region $r: \vdash P \Rightarrow \text{emp}_{\text{Ob}}^{\text{Rid} \setminus \{r\}}$. Notice that P can itself contain region assertions. The levels, λ , provide a technical device to avoid inconsistencies due to this nesting of regions. To improve readability, we usually omit the details related to levels, as they can be easily inferred from the structure of the proofs, if needed.

Specification format. In our examples and in the simplified rules of Fig. 6, we only use *atomic* and *Hoare* triples. In general, a command may manipulate some resources P_h non-atomically, and some other resources $P_a(x)$ atomically, at the same time. The specifications in their general form are therefore an hybrid of pure atomic and Hoare triples, called *hybrid* triples:

$$m; \lambda; \mathcal{A} \Vdash \forall x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \mathbb{C} \langle Q_h(x) \mid Q_a(x) \rangle$$

The Hoare precondition P_h is a resource that is owned by the command and, as such, cannot be invalidated by actions of the environment. The command is allowed to manipulate this owned resource non-atomically, provided it satisfies the Hoare postcondition Q_h upon termination. The atomic precondition $P_a(x)$ represents the resource that can be shared between the command and the environment. The environment can update it, but only with the effect of going from $P_a(x)$ for some $x \in X$ to $P_a(x')$ for some $x' \in X$. The command is allowed to update it exactly once from $P_a(x)$ to perform its linearisation point, transforming it to a resource satisfying the atomic postcondition $Q_a(x)$. The atomic postcondition only needs to be true *just after* the linearisation point as the environment is allowed to update it immediately afterwards. The pseudo-quantified variable x has two important uses: it represents the “surface” of allowed interference by the environment; it is bound in the postcondition to the value of the parameter of the atomic precondition *just before* the linearisation point.

The atomic triple is a hybrid triple with $P_h = Q_h = \text{emp}$. A Hoare triple is a hybrid triple with $P_a(x) = Q_a(x) = \text{emp}$. We omit the pseudo-quantifier from an atomic triple when the pseudo-quantified variable does not occur in the triple, and thus could be trivially quantified as

$\forall x \in \text{AVal} \rightarrow_{\perp} \text{AVal}$. When the liveness assumption is trivial, i.e. $\forall x \in X \rightarrow_k X$, we abbreviate it with $\forall x \in X$.

The context of the triple m, λ, \mathcal{A} consists of a layer m , a level λ , and an *atomicity context* \mathcal{A} . These components record information about the proof context of the judgement. The layer m indicates that we are in a context where we are forbidden from considering obligations with layers $\geq m$ as live. The level λ restricts the opening of regions. The atomicity context records information about which pseudo-quantifications were present in the goal judgement (**MkATOM** transfers them from a specification to the context, for instance).

Let us now elaborate on the intuitive liveness meaning of the triple. A hybrid triple guarantees termination of the command *only if* the environment satisfies the layered liveness invariants represented by pseudo-quantifiers (of the specification and in \mathcal{A}) and obligations. Consider the case of liveness invariants encoded by obligations. The idea is to examine the traces instrumented with the logical state, and consider for each position which obligations are held by the environment and which are held locally. Now suppose the environment always eventually fulfils *every* obligation (i.e. for each obligation O there are infinitely many positions where O is not held by the environment). This environment is certainly *good* with respect to the liveness assumptions. When is the environment allowed to keep an obligation O forever? Only when we locally hold forever some obligation of layer strictly smaller than $\text{lay}(O)$. This intuition about obligations extends to liveness assumptions attached to pseudo-quantifications in the triple and in the atomicity context. All these assumptions need to be layered to avoid unsound circularities, which is why the pseudo-quantifier carries a layer k .

5 TADA LIVE RULES

The general TaDA Live proof system for hybrid triples is given in the Appendix. Figure 6 gives key derived rules for atomic and Hoare triples which capture the essence of our liveness reasoning. The most important rules for liveness are the **LIVEC** and **WHILE** rules discussed in Section 3.

*The **LIVEC** and **WHILE** rules.* These rules depend crucially on the environment liveness condition, discussed in detail below. For the full **LIVEC** rule, the triple context is now complete and the pseudo-quantifier is indexed by its layer $k \geq n$. For the **WHILE** rule, we now have the target states $T(\beta)$ can depend on the variant (this parametrisation is for example necessary for verifying spin lock); the $\text{not-incr}(L, M)$ condition in the simplified rule corresponds to the stability requirement $\forall \alpha. \mathcal{A} \vDash \exists \alpha'. L * M(\alpha') \wedge \alpha' \leq \alpha$ stable which requires the measure to never increase by stating that whatever upper bound α on the measure may hold at some point, no transition can make the measure exceed it in the future; a layer condition $\forall \beta \leq \beta_0. \vdash_{\mathcal{A}} P(\beta) \triangleright m(\beta) \geq m$ which requires the loop invariant to only contain obligations of layers that cannot be assumed live; and a straightforward condition on the modified variables of \mathbb{C} .

*The **PAR** rule.* Rule **PAR** is the usual rule of parallel, but with premises constraining the layers. They require that, for every layer k that may be assumed live by thread 1, thread 2 does not hold any obligations at that level or lower in its postcondition, and vice versa. This is because joining a parallel requires both threads to terminate, and obligations held at the postcondition will be held constantly until the other thread terminates.

*The **FRAMEA** and **FRAMEH** rules.* The interesting aspect of the frame rules is their condition on layers: the frame can only contain obligations of layer greater or equal than the one in the context. Rule **LAYW** can be used to lower the layer artificially in preparation for such a step. The idea is that if we are framing an obligation, we will be holding it continuously for the execution of the command, which means we are forbidden from assuming obligations of higher layer live.

$$\begin{array}{c}
\frac{n; \lambda; \mathcal{A} \vdash \exists x \in X. P(x) \xrightarrow{M} \exists x' \in X'. P(x') \quad m, k \geq n \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle} \text{LIVEC} \\
\\
\frac{\forall \beta \leq \beta_0. m(\beta); \lambda; \mathcal{A} \vdash L \xrightarrow{M} T(\beta) \quad \forall \beta \leq \beta_0. \vdash_{\mathcal{A}} P(\beta) \triangleright m(\beta) \geq m \quad \forall \alpha. \mathcal{A} \vDash \exists \alpha'. L * M(\alpha') \wedge \alpha' \leq \alpha \text{ stable} \quad \text{pv}(T, L, M) \cap \text{mod}(\mathbb{C}) = \emptyset \quad \forall \beta \leq \beta_0. \forall b \in \text{Bool}. m; \lambda; \mathcal{A} \vdash_{\Phi} \{P(\beta) * (b \Rightarrow T(\beta)) \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma. P(\gamma) \wedge \gamma \leq \beta * (b \Rightarrow \gamma < \beta)\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P(\beta_0) * L\} \text{ while}(\mathbb{B})\{\mathbb{C}\} \{\exists \beta. P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta \leq \beta_0\}} \text{WHILE} \\
\\
\frac{\frac{m_1; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1\} \mathbb{C}_1 \{Q_1\} \quad \vdash_{\mathcal{A}} Q_1 \triangleright m_2 \leq m \quad m_1 \leq m_2}{m_2; \lambda; \mathcal{A} \vdash_{\Phi} \{P_2\} \mathbb{C}_2 \{Q_2\} \quad \vdash_{\mathcal{A}} Q_2 \triangleright m_1 \leq m} \text{PAR} \quad \frac{m_1 \leq m_2}{m_1; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \mathbb{C} \{Q\}} \text{LAYW}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1 * P_2\} \mathbb{C}_1 \parallel \mathbb{C}_2 \{Q_1 * Q_2\}} \text{PAR} \\
\\
\frac{\text{fv}(R(x)) \cap \text{mod}(\mathbb{C}) = \emptyset \quad \forall x \in X. \vdash_{\mathcal{A}} R(x) \triangleright m \quad \forall x \in X. \mathcal{A} \vDash R(x) \text{ stable}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle} \text{FRAMEA} \\
\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P(x) * R(x) \rangle \mathbb{C} \langle Q(x) * R(x) \rangle} \text{FRAMEA} \\
\\
\frac{\text{fv}(R) \cap \text{mod}(\mathbb{C}) = \emptyset \quad \vdash_{\mathcal{A}} R \triangleright m \quad \mathcal{A} \vDash R \text{ stable} \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \mathbb{C} \{Q\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P * R\} \mathbb{C} \{Q * R\}} \text{FRAMEH} \\
\frac{\mathcal{A} \vDash P \text{ stable} \quad \mathcal{A} \vDash Q \text{ stable} \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \langle P \rangle \mathbb{C} \langle Q \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \mathbb{C} \{Q\}} \text{ATOMW} \\
\\
\frac{\lambda < \lambda' \quad r \notin \text{dom}(\mathcal{A}) \quad \mathcal{A}' = \mathcal{A}[r \mapsto (X, k, X', T)] \quad T \subseteq \mathcal{T}_1(G) \quad R = \text{io}(T) \quad m; \lambda'; \mathcal{A}' \vdash_{\Phi} \{\exists x \in X. t_r^\lambda(x) * r \Rightarrow \blacklozenge\} \mathbb{C} \{\exists x, y. R(x, y) \wedge r \Rightarrow (x, y)\}}{m; \lambda'; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle t_r^\lambda(x) * \lceil G \rceil_r \rangle \mathbb{C} \langle \exists y. t_r^\lambda(y) * \lceil G \rceil_r \wedge R(x, y) \rangle} \text{MKATOM} \\
\\
\frac{\mathcal{A}' = \mathcal{A}[r \mapsto \perp] \quad r \in \text{dom}(\mathcal{A}) \quad \mathcal{A} \vDash I(t_r^\lambda(z)) * Q_1(x, z) \wedge R(x, z) \xrightarrow{\lambda \Rightarrow \lambda+1} Q'_1(x, z) \quad \mathcal{A} \vDash I(t_r^\lambda(z)) * Q_2(x, z) \wedge x = z \xrightarrow{\lambda \Rightarrow \lambda+1} Q'_2(x, z) \quad \vdash_{\mathcal{A}} P(x) \ni [O_0(x)]_r^{\perp} \quad \vdash_{\mathcal{A}} Q_1(x, z) \ni [O_1(x, z)]_r^{\perp} \quad \vdash_{\mathcal{A}} Q_2(x, z) \ni [O_2(x)]_r^{\perp} \quad \{((x, O_0(x)), (z, O_1(x, z))) \mid x \in X \wedge R(x, z)\} \cup \{((x, O_0(x)), (x, O_2(x))) \mid x \in X\} \subseteq \text{tr}(\mathcal{A}, r)}{m; \lambda; \mathcal{A}' \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle I(t_r^\lambda(x)) * P(x) \rangle \mathbb{C} \langle \exists z. I(t_r^\lambda(z)) * \left(\begin{array}{l} R(x, z) \wedge Q_1(x, z) \\ \vee x = z \wedge Q_2(x) \end{array} \right) \rangle} \text{UPDREG} \\
\\
\frac{\vdash_{\mathcal{A}} P(x) \ni [O_1]_r^{\perp} \quad \vdash_{\mathcal{A}} Q(x, z) \ni [O_2(x, z)]_r^{\perp} \quad \{((x, O_1(x)), (z, O_2(x, z))) \mid x \in X \wedge R(x, z)\} \subseteq \mathcal{T}_1(G) \quad r \in \text{dom}(\mathcal{A}) \Rightarrow R = \text{id} \quad \mathcal{A} \vDash I(t_r^\lambda(z)) * Q(x, z) \wedge R(x, z) \xrightarrow{\lambda \Rightarrow \lambda+1} Q'(x, z)}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle I(t_r^\lambda(x)) * \lceil G \rceil_r * P(x) \rangle \mathbb{C} \langle \exists z. I(t_r^\lambda(z)) * Q(x, z) \wedge R(x, z) \rangle} \text{LIFTA} \\
\\
\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. z \in Z. \langle P(x, z) \rangle \mathbb{C} \langle Q(x, z) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle \exists z \in Z. P(x, z) \rangle \mathbb{C} \langle \exists z \in Z. Q(x, z) \rangle} \text{A}\exists\text{ELIM}
\end{array}$$

Fig. 6. TaDA Live rules. Abbreviations:

$\vdash_{\mathcal{A}} P \triangleright k$ means $\forall r \in \text{Rld}. \vdash_{\mathcal{A}} P \Rightarrow r \triangleright k$;

$\vdash_{\mathcal{A}} P \ni [O]_r^{\perp}$ means $\vdash_{\mathcal{A}} P \Rightarrow [O]_r^{\perp} * \text{emp}_{\text{Ob}}^{\perp}$.

$$\begin{array}{c}
\mathcal{A} \vDash L \text{ stable} \quad \vdash_{\mathcal{A}} L \Rightarrow L * \exists \alpha. M(\alpha) \\
\frac{m; \lambda; \mathcal{A} \vdash L * M(\alpha) : L * M(\alpha) \longrightarrow T}{m; \lambda; \mathcal{A} \vdash L \xrightarrow{M} T} \text{ENVLIVE} \\
\\
\frac{m; \lambda; \mathcal{A} \vdash L(\alpha) : L_1(\alpha) \longrightarrow T \quad m; \lambda; \mathcal{A} \vdash L(\alpha) : L_2(\alpha) \longrightarrow T}{m; \lambda; \mathcal{A} \vdash L(\alpha) : L_1(\alpha) \vee L_2(\alpha) \longrightarrow T} \text{ECASE} \quad \frac{\forall \alpha. \vdash_{\mathcal{A}} T'(\alpha) \Rightarrow T}{m; \lambda; \mathcal{A} \vdash L(\alpha) : T'(\alpha) \longrightarrow T} \text{LIVET} \\
\\
\frac{\text{impr}_{\mathcal{A}}(\mathfrak{t}_r^\lambda, L, L', \text{io}(R), R', T) \quad m > \text{lay}(O) \quad \forall \alpha. \vdash_{\mathcal{A}} L(\alpha) \triangleright \text{lay}(O) \quad O \in \text{AOB} \quad \lambda < \lambda' \quad R = \cup \{ \mathcal{T}_i(G') \mid G' \# G \} \quad R' = \{ (a_1, a_2) \mid ((a_1, O_1), (a_2, O_2)) \in R, O \sqsubseteq O_1, O \not\sqsubseteq O_2 \}}{m; \lambda'; \mathcal{A} \vdash L'(\alpha) : L(\alpha) * \exists x. \mathfrak{t}_r^\lambda(x) * [G]_r * [O]_r^E \longrightarrow T} \text{LIVEO} \\
\\
\frac{\text{impr}_{\mathcal{A}}(\mathfrak{t}_r^\lambda, L, L', R, R', T) \quad m > k \quad \forall \alpha. \vdash_{\mathcal{A}} L(\alpha) \triangleright k \quad \lambda < \lambda' \quad (X \twoheadrightarrow_k X') = \text{live}(\mathcal{A}, r) \quad R = \cup \{ \text{io}(\mathcal{T}_i(G')) \mid G' \# G \} \quad R' = \{ (a_1, a_2) \in R \mid a_2 \in X' \}}{m; \lambda'; \mathcal{A} \vdash L'(\alpha) : L(\alpha) * \exists x. \mathfrak{t}_r^\lambda(x) * [G]_r * r \Rightarrow \diamond \longrightarrow T} \text{LIVEA}
\end{array}$$

Fig. 7. Environment Liveness Condition Rules

The Atomicity Rules. The atomicity Rules **MkATOM**, **LIFTA** and **UPDREG** are standard TaDA rules, adapted to TaDA Live by simply propagating the liveness assumptions. The **MkATOM** rule says that a Hoare triple can be promoted to an atomic triple if it contains a “certificate” of atomicity for the region r , that is it goes from owning $r \Rightarrow \diamond$ to some $r \Rightarrow (x, y)$. The important aspect for TaDA Live is that the liveness assumption of the atomic triple is recorded in the atomicity context. This way it becomes available for supporting environment liveness condition proofs. The proof of spin lock and CLH lock in the appendix illustrate applications of **MkATOM**.

The rules **LIFTA** and **UPDREG** are used to lift atomic updates performed on the interpretation of a region to the region itself. Rule **UPDREG** is used to declare the update a linearization point (storing the fact that it happened in the atomicity tracking assertion). The side-conditions of these two rules check that the update is consistent with the transition system of the region (**LIFTA**) or the expected linearization point as recorded in \mathcal{A} (**UPDREG**).

5.0.1 The Environment Liveness Condition. The essence of the termination argument is captured by the conditions of the form $m; \lambda; \mathcal{A} \vdash L \xrightarrow{M} T$ in **LIVEC** and **WHILE**. They establish “always eventually T holds” facts. The condition is parametrised by L , an assertion that holds at any point in the traces we are considering, an assertion T , characterising the so-called *target* states, and an assertion $M(\alpha)$ parametric on some ordinal α , which represents the environment progress measure. Intuitively, the condition states that, from any state satisfying $L * M(\alpha)$, for some α , we can find an environment transition that *must* happen that would take us either to T , or to some state satisfying $L * M(\alpha')$ with $\alpha' < \alpha$. Additionally, any transition from L to L that *may* happen does not strictly increase the progress measure, unless they end in a target state. The transitions that must happen are characterised by being those that either: (1) fulfil some obligation known to be in the environment and with layer lower than the ones we may hold locally, (2) fulfil some environment liveness assumption stored in \mathcal{A} with layer lower than the ones we may hold locally.

Take the environment liveness condition required by the application of **LIVEC** in the proof of Example 3.1. There we have: $m = 1$, $M(\alpha) = (\alpha = 0)$ and

$$\begin{aligned}
L &= \exists l \in \{0, 1\}, d. \mathbf{c}_r(x, \text{done}, l, d) * l = 1 \Rightarrow \lfloor \mathbf{k} \rfloor_r^E \\
T &= \exists l \in \{0\}, d. \mathbf{c}_r(x, \text{done}, l, d) * l = 1 \Rightarrow \lfloor \mathbf{k} \rfloor_r^E
\end{aligned}$$

That is, during the interference phase, we know that at any point in time the atomic precondition will hold for some $l \in \{0, 1\}$; we want to prove that the environment will always eventually set l to 0. Here this is particularly easy to show: L states that when $l = 1$ the obligation \mathbf{k} is held by the environment; since $\text{lay}(\mathbf{k}) = \mathbf{0} < \mathbf{1}$ (and L does not hold obligations) we can assume the obligation will be eventually fulfilled; the only transition that can fulfil it is the one that sets $l = 0$, so in exactly one such step we reach T . This justifies the trivial definition of M : we do not need to keep track of progress towards T as we reach it in exactly one of the steps that *must* happen.

Technically, the environment liveness condition can be proven using the rules in Fig. 7. The only rule that applies directly is **ENVLIVE**, which checks that in a state satisfying L one can always measure progress (second premise), and then asks to discharge an auxiliary judgement of the form $m; \lambda; \mathcal{A} \vdash L(\alpha) : L(\alpha) \xrightarrow{M} T$ which allows case analysis on L by means of rule **ECASE**, until we reach one of the base cases. Rule **LIVET** is the case where we are already in T . Rule **LIVEO** is the case where we justify progress by appealing to an environment-owned atomic obligation O of some region with id r . The layer of O needs to be lower than the layer of any obligation we may be holding (premises two and three). We then compute the set of allowed transitions R for r , and the set of transitions R' that fulfil O (which must happen), and check that, when the environment will fulfil O , the progress measure will *improve*, as ensured by the $\text{impr}_{\mathcal{A}}$ condition defined as follows. The condition $\text{impr}_{\mathcal{A}}(\mathbf{t}_r^\lambda, L, R, R', T)$ holds if, $R' \neq \emptyset$, and for every α_1, α_2 , every $(x_1, x_2) \in R$ and every states w_1 and w_2 satisfying $L(\alpha) * \mathbf{t}_r^\lambda(x_1)$ and $L'(\alpha) * \mathbf{t}_r^\lambda(x_1)$ respectively, we have that either (a) w_2 satisfies $T * \text{True}$, or (b) $\alpha_1 \leq \alpha_2$ and, if $(x_1, x_2) \in R'$ then $\alpha_1 < \alpha_2$.

Rule **LIVEA** is the case where we justify progress by appealing to an environment liveness assumption stored in \mathcal{A} . Similarly to **LIVEO**, R and R' represent the allowed transitions and the transitions fulfilling the environment liveness assumption respectively. The layer of the assumption needs to be lower than any layer we may be holding. Since the environment liveness assumptions only hold in the interference phase of an update, the rule needs evidence that the linearisation point on r has not occurred yet, which is provided by $r \Rightarrow \diamond$.

In the proof of Example 3.1, the environment liveness condition is proved by:

$$\frac{\frac{\frac{\forall \alpha. \mathbf{t}_\emptyset L_0(\alpha) \Rightarrow T}{\mathbf{1}; \emptyset \vdash L(\alpha) : L_0(\alpha) \longrightarrow T} \text{LIVET} \quad \frac{\text{impr}_\emptyset(\mathbf{c}_r, L_1, L, R_c, R_{\mathbf{k}}), T}{\mathbf{1}; \emptyset \vdash L(\alpha) : L_1(\alpha) \longrightarrow T} \text{LIVEO}}{\mathbf{1}; \emptyset \vdash L(\alpha) : L_0(\alpha) \vee L_1(\alpha) \longrightarrow T} \text{ECASE}}{\mathbf{1}; \emptyset \vdash L \xrightarrow{M} T} \text{ENVLIVE}$$

Since L trivially implies $L * \exists \alpha. M(\alpha)$, we can apply **ENVLIVE**, setting $L(\alpha) = (L \wedge \alpha = 0)$. Then we apply **ECASE** to split on the value of l : $L(\alpha) = L_0(\alpha) \vee L_1(\alpha)$ where $L_0(\alpha) = \mathbf{c}_r(x, \text{done}, 0, _) \wedge \alpha = 0$ and $L_1(\alpha) = \mathbf{c}_r(x, \text{done}, 1, _) * \lfloor \mathbf{k} \rfloor_r^E \wedge \alpha = 0$. If $l = 0$ we can apply **LIVET** as we are already in T ; if $l = 1$, L_1 entails $\lfloor \mathbf{k} \rfloor_r^E$ so we can apply **LIVEO** with $G = \mathbf{0}$, $O = \mathbf{k}$, $R_c = \bigcup_G \mathcal{T}_c(G)$ (i.e. all the transitions of the interference protocol of \mathbf{c}), and $R_{\mathbf{k}} = \{((1, d), (0, d)) \mid d \in \text{Bool}\}$, that is, the only transition which fulfils \mathbf{k} . The $\text{impr}_{\mathcal{A}}$ condition is satisfied: every transition in R_c does not increase α , and any transition in $R_{\mathbf{k}}$ takes us directly to T .

Soundness. Specifications \mathbb{S} are hybrid triples with the command omitted; our model, defined in Appendix, defines an (infinite) trace semantics for specifications $\llbracket \mathbb{S} \rrbracket$. The infix syntactic judgement used in rules can be written as $\vdash \mathbb{C} : \mathbb{S}$. The corresponding semantic judgement $\vDash \mathbb{C} : \mathbb{S}$ is defined as $\llbracket \mathbb{C} \rrbracket \subseteq \llbracket \mathbb{S} \rrbracket$.

THEOREM 5.1 (SOUNDNESS). *If $\vdash_{\Phi} \mathbb{C} : \mathbb{S}$ then $\vDash_{\Phi} \mathbb{C} : \mathbb{S}$.*

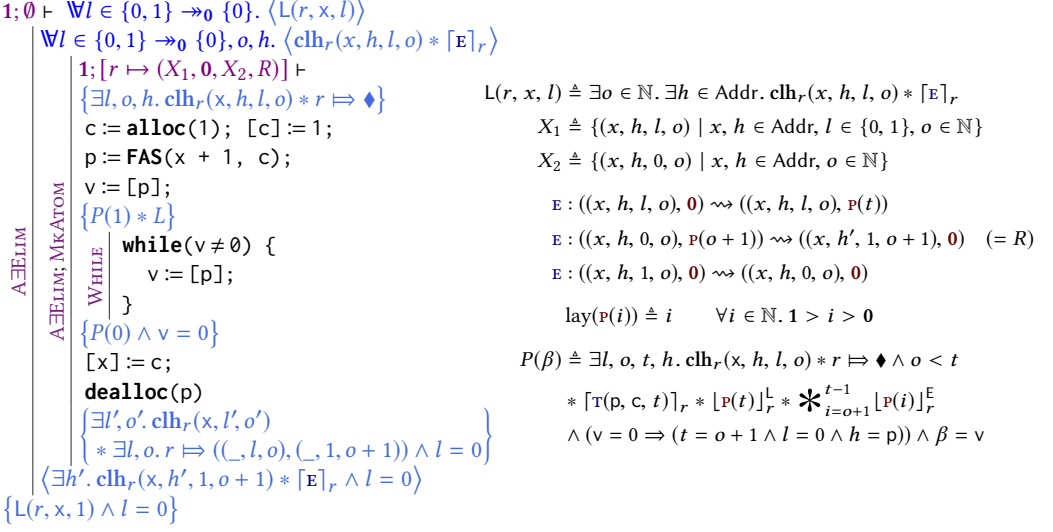


Fig. 8. Proof of CLH lock. On the right: auxiliary definitions, the transitions of \mathcal{T}_{clh} , the layers of obligations.

5.1 Proving CLH lock correct

The proof of CLH is sketched in Fig. 11. We explain here the important steps and refer to Appendix for the full details.

CLH locks work by maintaining an internal FIFO queue of threads requesting the lock. This queue is virtual, in that the next pointers are not kept in main memory: each requesting thread keeps a pointer of the previous thread in a local variable p . We maintain an abstract view of this queue, which precise enough to support the termination argument, but abstract enough to only see an update at the linearisation point of the lock operation. We associate, through ghost state, to each thread requesting the lock, a ticket number $t \in \mathbb{N}$ which corresponds to the order of arrival. Every time a thread joins the queue, it gets assigned the next available ticket. We also keep track of the ticket number o of the thread at the head of the queue, the owner. The region representing the shared lock state is thus $\text{clh}_r(x, h, l, o)$ where x is the address of the lock, h is the address of the head of the queue (important for verifying the `unlock` operation) $l \in \text{Bool}$ is the abstract state of the lock and $o \in \mathbb{N}$ is the current owner number. We record o in the region as this is global shared information about the lock's state, while the ticket number of the current thread is local, and stored as the argument of the obligation $p(t)$, which is obtained locally once a thread joins the queue.

The proof needs to show atomicity of the operation, which is done by applying the **MkATOM** rule. The rule requires proving a Hoare triple, thus allowing multiple steps to be taken, but with evidence (provided by the atomicity tracking component) of a single atomic update having taken place for some region r . In such case, the update declared in the atomicity tracking component is promoted to the atomic postcondition in the derived atomic triple. Note that the state of the region r might have been changed by the environment after the update, a fact that would be reflected in weak information about the state in the Hoare postcondition of the premise of **MkATOM**. The atomic postcondition is however derived from the information in $r \Rightarrow (x, y)$, which is stable, as it is information about an event in the past, and not the current state.

The only part of the proof where the termination argument is non-trivial, is the application of the **WHILE** rule. The loop invariant $P(\beta)$, the persistent invariant L , the environment progress

measure, and the target states are defined as follows:

$$\begin{aligned}
P(\beta) &\triangleq \exists l, o, t, h. \text{clh}_r(x, h, l, o) * r \Rightarrow \blacklozenge * [\mathsf{T}(p, c, t)]_r * [\mathsf{P}(t)]_r^L * \\
&\quad *_{i=o+1}^{t-1} [\mathsf{P}(i)]_r^E \wedge o < t \wedge (v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)) \wedge \beta = v \\
L &= \exists l, o, h. \text{clh}_r(x, h, l, o) * [\mathsf{P}(t)]_r^L * *_{i=o+1}^{t-1} [\mathsf{P}(i)]_r^E * r \Rightarrow \blacklozenge \wedge o < t \\
M(\alpha) &= \exists l, o, h. \text{clh}_r(x, h, l, o) \wedge \alpha = 2(t - o - 1) + l \\
T &= \exists l, o, h. \text{clh}_r(x, h, l, o) \wedge t = o + 1 \wedge l = 0 \wedge h = p
\end{aligned}$$

Here, $\mathsf{T}(p, c, t)$ is ghost state recording the current and previous pointers in the queue of threads waiting for the lock. The natural number t is the order of arrival at the queue: it is incremented every time a thread joins the queue and assigned to that thread. The obligation $\mathsf{P}(t)$, with layer t , represents the responsibility of effectively acquiring the lock, once a thread reaches the head of the queue. The liveness assumption $X_1 \rightarrow_0 X_2$ recorded in the atomicity context ensures the owner of the lock will eventually unlock the lock. The persistent invariant L knows that every thread with number i ahead in the queue (i.e. $o < i < t$) owns the obligation $\mathsf{P}(i)$. The while loop is unblocked if the lock is unlocked and the current thread is at the head of the queue, as formalised by T . The measure of environment progress is $2(t - o - 1) + l$. The number of threads ahead in the queue is $t - o - 1$. To show that the environment eventually takes us stably to T we can split into two cases (using **ECASE**): either the lock is unlocked, in which case we can invoke the liveness assumption in the atomicity context and deduce that l goes from 1 to 0 without changing the owner number, which decreases the measure by 1 (using **LIVEA**). In the other case, the lock is unlocked but t is not the head of the queue. In that case there is some thread with number $t' < t$ holding $\mathsf{P}(t')$ which has layer strictly lower than t and we can apply **LIVEO**. The only transition that can fulfill such obligation is the one that increments the owner number while setting l from 0 to 1, bringing the measure down by 1.

5.2 Evaluation

We have used TaDA Live to verify a number of representative examples. In Appendix, we present detailed proofs of spin lock, CLH lock, our distinguishing client, a linearizable counter module using multiple locks and a lock-coupling set. Spin lock illustrates blocking in the presence of impedance. CLH lock requires a more interesting environment liveness condition that captures why each thread will eventually reach the head of the queue, by combining liveness assumptions from pseudo-quantifiers and obligations for internal blocking. The distinguishing client shows how TaDA Live’s specifications lead to simple and modular client proofs. It also illustrates how obligations and layers allow for sound reasoning with a mix of blocking operations and busy-waiting. The counter module shows that for simple common programming patterns, the layer system leads to natural and modular client proofs. Lock-coupling set is a challenging example with a dynamic number of locks, which shows the flexibility and power of the layer system. These examples cover all the proof patterns needed to prove all the examples of the LiLi papers [20, 21]. Notably, proofs in LiLi involving modules that use locks, which require in-lining some non-atomic implementation of the lock operations in the client, resulting in far less modular proofs and unnecessarily intertwined termination arguments.

5.3 Limitations

Non-local linearization points. As with other total program logics, TaDA Live does not support helping/speculation. Such patterns are challenging for the identification of the linearization point, which is entirely a safety property. 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 the related literature, to explore termination in a simpler logic.

Non-structural thread creation. TaDA Live currently supports only structural parallel. We believe the support of non-structural fork/join would not require substantial new ideas. For comparison, LiLi does not support parallel nor fork/join.

Scheduling non-determinism. A more interesting limitation comes from our approach to specifying impedance. For non-blocking programs, the ordinal-based approach is complete. It is not complete for blocking programs. Consider $\mathbb{C}_2 \triangleq (\mathbb{C}_1 \parallel [\text{done}] := \text{true})$ where \mathbb{C}_1 is the distinguishing client *with a spin lock*. Scheduler fairness guarantees \mathbb{C}_2 will be eventually executed. 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 $\alpha(S)$ to represent a prophecy of the number of steps it will take to fulfil *live* obligation s . This would solve the problem for \mathbb{C}_2 , because $\alpha(s) + 1$ (where s is fulfilled by setting done to true) would be the required budget. To the best of our knowledge, none of the approaches in the literature can handle this example.

Loop body specifications. 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 $\text{lay}(\kappa)$; doing otherwise would be unsound in the case when κ is held continuously. A solution would be to introduce an assertion $\text{live}(\kappa)$, certifying that an obligation is fulfilled at some point in a block of code. It 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 $\text{live}(\kappa)$.

More Expressive Layers. It might be possible to shorten some program proofs if the lay function is constrained through assertions, rather than statically specified. This would potentially enable layers to change as result of interference, provided their relative order is preserved. We are not aware of any example that cannot be proved using static layers and critically requires more expressive layers. It is not clear how to ensure soundness if interference on layers is allowed.

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 [17], 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, 18]. Similar techniques have been used in [12] to prove global deadlock-freedom (a *safety* property requiring that at least some thread can take a step), and [14] 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 abstract blocking patterns arising from busy waiting and shared memory interference. Our solution uniformly handles programs that mix blocking primitives and ad-hoc synchronisation patterns. The notion of “obligations” found in [1, 12, 14, 18] is only superficially related to our obligations.

First, obligations found in the literature represent primitive blocking events (like the acquisition of a lock). They are also typically equipped with a structure to represent causal dependencies between these events, to detect deadlocks. Our layered obligations are associated with arbitrary *abstract* state changes, removing the need for ad-hoc treatment of primitives, and supporting abstraction and abstract atomicity. Moreover, our layers do not represent causal dependencies between events, but rather dependencies between liveness assumptions in a termination argument. This reflects in our specifications, e.g. a lock operation does not return an obligation in its post-condition. Whether there is a need for an obligation linked to that lock is entirely dependent on how the client will decide to use the lock. Nevertheless, the specification precisely captures the termination guarantees of lock operations. Finally, obligations in the literature have a purely safety semantics, from which one can only derive safety properties as non-blocking or deadlock-freedom. Our obligations explain how to express proper liveness invariants, how to blend them with the layers, and how to use them for proving termination.

History-based methods. The CertiKOS project [11, 16] 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 specification with the corresponding 30-line specification from Fig. 7 in [16]), and our reasoning is simpler (the layered obligation system leads to a more intuitive proof compared to the proof of MCS locks in [16]).⁷

Higher-order logics. Iris [15] has been extended to reason about termination [26, 27]. The idea is to establish a non-contextual refinement between two programs, one acting as a specification and one as an implementation. A crucial shortcoming is that the approach is not modular: the refinement concerns two closed programs and cannot be reused as part of a 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. [10]. This work only supports atomic specifications for non-blocking operations. Liang et al. [22] studies the exact relationship between common progress properties of fine-grained operations and contextual refinement. The study of the contextual refinement induced by our triple semantics is future work.

LiLi. The work closest to ours is LiLi [20, 21]. LiLi was the first program logic to prove total specifications for linearizable concurrent objects with internal blocking [20]. LiLi is also a concurrent separation logic, but proves linearizability via contextual refinement: specifications are expressed as atomic programs and verification proves a refinement relation between these programs and their implementations. Recently, LiLi was extended to handle external blocking [21]. Although we share most of our goals with LiLi, our approach is radically different. LiLi’s specifications use a primitive blocking operation to represent abstract blocking operations, and appeal to scheduler fairness assumptions to communicate sensitivity to impedance. LiLi’s verification is limited to proving a

⁷The proof is a variation of the one for CLH.

module correct with respect to its specification but it does not support parallel composition and does not directly support client verification. Our environment liveness condition was informed by LiLi’s *definite progress* condition. Our obligations are similar in spirit to LiLi’s definite actions, but attain higher locality of the argument thanks to subjectivity and layers. 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. Our key contribution is our approach to abstract atomic blocking as reliance of termination on liveness properties of the environment. We have illustrated the subtlety of our reasoning using a spin lock and a CLH lock, and a client. We have given many other examples in the Appendix. We believe our work provides a substantial contribution to the understanding of compositional reasoning about progress for fine-grained concurrent algorithms.

In future, we will 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 [24]. This should give us mechanisms for extending our compositional reasoning towards general liveness for reactive systems. We would, eventually, like to provide a semi-automatic implementation of TaDA Live based on CAPER [7].

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] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *ESOP (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 363–377.
- [10] 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.
- [11] 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.
- [12] Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 415–441.
- [13] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
- [14] 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.

- [15] 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.
- [16] 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.
- [17] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 4137. Springer, 233–247.
- [18] 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.
- [19] Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *POPL*. ACM, 561–574.
- [20] 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.
- [21] Hongjin Liang and Xinyu Feng. 2018. Progress of concurrent objects with partial methods. *PACMPL* 2, POPL (2018), 20:1–20:31.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] Peter W. O’Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR (Lecture Notes in Computer Science)*, Vol. 3170. Springer, 49–67.
- [26] 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.
- [27] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. *CoRR* abs/1701.05888 (2017).

Appendix

A PROGRAM PROOFS CONVENTIONS

A.1 Specification abbreviations

Here is a summary of all the abbreviations we use in writing specifications. The full hybrid specification format is

$$m; \lambda; \mathcal{A} \vDash \mathbb{W}x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle$$

The $\exists y$ quantification is a normal existential quantification but its scope extends over both the Hoare and the atomic postconditions. We omit it when y does not occur in the triple.

$$\mathbb{W}x \triangleq \mathbb{W}x \in \text{Val}$$

$$\mathbb{W}x \in X \triangleq \mathbb{W}x \in X \rightarrow_{\perp} X$$

$$\mathbb{W}x_1 \in X_1 \rightarrow_k X'_1, x_2 \in X_2 \rightarrow_k X'_2 \triangleq \mathbb{W}(x_1, x_2) \in (X_1 \times X_2) \rightarrow_k (X'_1 \times X'_2).$$

An omitted pseudo-quantifier is to be understood as the trivial pseudo-quantifier $\mathbb{W}x \in \text{AVal} \rightarrow_{\perp} \text{AVal}$, for an unused x .

The triples

$$m, \lambda, \mathcal{A} \vdash \{P\} \mathbb{C} \{Q\}$$

$$m, \lambda, \mathcal{A} \vdash \mathbb{W}x \in X \rightarrow_k X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle$$

are abbreviated with

$$m; \lambda; \mathcal{A} \vdash \langle P \mid \text{emp} \rangle \mathbb{C} \langle Q \mid \text{emp} \rangle$$

$$\forall \vec{v}_0. m; \lambda; \mathcal{A} \vdash \mathbb{W}x \in X \rightarrow_k X'. \langle \vec{v}_0 \doteq \vec{v}_0 \mid P'(x) \rangle \mathbb{C} \exists \vec{v}_1. \langle \vec{v}_0 \doteq \vec{v}_0 \wedge \vec{v}_1 \doteq \vec{v}_1 \mid Q'(x) \rangle$$

respectively, where $\vec{v}_0 = \text{pv}(P(x))$, $\vec{v}_1 = \text{pv}(Q(x)) \setminus \vec{v}_0$, $P'(x) = P(x)[\vec{v}_0/\vec{v}_0]$ and $Q'(x) = Q(x)[\vec{v}_0/\vec{v}_0, \vec{v}_1/\vec{v}_1]$ (for technical reasons the atomic pre/post-conditions in the general triples cannot contain program variables). In other words, the program variables mentioned in the atomic pre/post-conditions refer to the value stored in them *at the beginning* of the execution of the command. Most commonly, the program variables used this way are actually not modified by the command.

A.2 Guard and Obligation Algebras

Defining a guard algebra can be tedious. In program proofs, we will define guard algebras by generating them from some *guard constructors* and some axioms defining the guard operation. We explain this construction by introducing some unsurprising auxiliary definitions.

Definition A.1. Given a set X , the set $\mathcal{M}(X) \triangleq X \rightarrow \mathbb{N}$ is the set of *multisets* over X ; \emptyset is the empty multiset (i.e. the function mapping every element to 0) and $\oplus: \mathcal{M}(X) \times \mathcal{M}(X) \rightarrow \mathcal{M}(X)$ is multiset union (i.e. the pointwise lifting of $+$). The expression $\langle x_1, \dots, x_n \rangle$ denotes the multiset containing the elements x_1, \dots, x_n . Given a set X , the *free commutative monoid* over X is the monoid $(\mathcal{M}(X), \oplus, \emptyset)$. Given a commutative monoid $(X, \bullet, \mathbf{0})$ and a congruence relation $\cong \subseteq X \times X$, the *quotient* $(X/\cong, \bullet/\cong, [\mathbf{0}]_{\cong})$ is a commutative monoid. Given a commutative monoid $(X, \bullet, \mathbf{0})$ and a set $U \subseteq X$ with $\mathbf{0} \notin U$, the *PCM over X induced by U* is $(X|_U, \bullet_U, \mathbf{0})$ where $X|_U = \{x \in X \mid \forall u \in U. \nexists y \in X. x = u \bullet y\}$ and for $x, y \in X|_U$, $x \bullet_U y = x \bullet y$ if $x \bullet y \in X|_U$, otherwise undefined.

For each guard algebra to be defined, we will introduce a number of symbols G_1, \dots, G_n , called *guard constructors* each with some *guard domain* $\text{dom}(G_i) \subseteq \text{AVal}^{k_i}$ for some $k_i \in \mathbb{N}$. They induce the set of *guard terms* $\text{GT} \triangleq \bigcup_{i=1}^n \{G_i(\vec{a}) \mid \vec{a} \in \text{dom}(G_i)\}$. By specifying some guard constructors, a congruence $\cong \subseteq \mathcal{M}(\text{GT}) \times \mathcal{M}(\text{GT})$ and a set $U \subseteq \mathcal{M}(\text{GT})/\cong$ one obtains the guard algebra $((\mathcal{M}(\text{GT})/\cong)|_U, (\oplus/\cong)_U, [\emptyset]_\cong)$.

The guard constructors are specified by listing their domains, writing $G: D$ to mean $\text{dom}(G) = D$.

The congruence \cong is specified as the smallest congruence satisfying given axioms of the form

$$\langle G_{i_1}(\vec{a}_{i_1}), \dots, G_{i_k}(\vec{a}_{i_k}) \rangle \cong \langle G_{j_1}(\vec{a}_{j_1}), \dots, G_{j_{k'}}(\vec{a}_{j_{k'}}) \rangle$$

which we write using the syntax

$$G_{i_1}(\vec{a}_{i_1}) \bullet \dots \bullet G_{i_k}(\vec{a}_{i_k}) = G_{j_1}(\vec{a}_{j_1}) \bullet \dots \bullet G_{j_{k'}}(\vec{a}_{j_{k'}})$$

The set U is specified as the smallest set satisfying given axioms of the form

$$\left[\langle G_{i_1}(\vec{a}_{i_1}), \dots, G_{i_k}(\vec{a}_{i_k}) \rangle \right]_\cong \in U$$

which we write using the syntax

$$G_{i_1}(\vec{a}_{i_1}) \bullet \dots \bullet G_{i_k}(\vec{a}_{i_k}) = \perp$$

Example A.2. The guard algebra used in Example 3.1, is expressed by using two guard constructors with empty domain, \mathbf{k} and \mathbf{d} , and axioms: $\mathbf{k} \bullet \mathbf{k} = \perp$, $\mathbf{d} \bullet \mathbf{d} = \perp$. Note that with no congruence axioms, the induced congruence relation is equality. These induce the guard algebra with elements $\{\emptyset, \langle \mathbf{k} \rangle, \langle \mathbf{d} \rangle, \langle \mathbf{k}, \mathbf{d} \rangle\}$.

A.3 Levels

Region levels are used to remove the possibility of unsound duplication of resources by opening regions. To see the problem consider a generic region $\mathbf{t}_r^\lambda(a)$; we have $\mathbf{t}_r^\lambda(a) \equiv \mathbf{t}_r^\lambda(a) * \mathbf{t}_r^\lambda(a)$: this is the essence of what it means for a region to be a shared resource. When we open a region however, we obtain ownership of the contents of its interpretation $\mathcal{I}(\mathbf{t}_r^\lambda(a))$; the interpretation can contain resources that are not shared, for example heap assertions, in which case we have $\mathcal{I}(\mathbf{t}_r^\lambda(a)) \neq \mathcal{I}(\mathbf{t}_r^\lambda(a)) * \mathcal{I}(\mathbf{t}_r^\lambda(a)) \equiv \text{False}$. Without constraining levels, one could start with $\mathbf{t}_r^\lambda(a)$, produce the equivalent $\mathbf{t}_r^\lambda(a) * \mathbf{t}_r^\lambda(a)$, open the first region assertion with **UPDREG** or **LIFTA**, then open the second region assertion and end up with **False**. Levels are a mean to avoid unsound derivations that use the above chain of implications. A level λ in the context of a judgement records that all the regions of level λ or higher might have been already opened and should not be opened again. The rules that do open regions (rules **UPDREG** and **LIFTA**) can only open a region of level λ if the level in the context is $\lambda + 1$, and they record the operation by setting the context level to λ , so that the region cannot be opened again.

The presentation of the program proofs omits the level annotations to ease readability. The levels can be unambiguously derived from the sequence of application of the **UPDREG** and **LIFTA** rules.

A.4 Layers

TaDA Live specifications include two layers: one in the context, and one decorating the liveness assumption. These layers are the only ones that “leak” in the abstract specifications. To make the specifications as reusable as possible, a common pattern is to allow instantiating the layers of a specification with different layers per instance of the module. For example, the fair lock specifications are

$$\begin{aligned} 1_r \vdash \forall l \in \{0, 1\} \rightarrow_{0_r} \{0\}. \langle L(r, x, l) \rangle \text{lock}(x) \langle L(r, x, 1) \wedge l = 0 \rangle \\ 0_r \vdash \langle L(r, x, 1) \rangle \text{unlock}(x) \langle L(r, x, 0) \rangle \end{aligned}$$

where 1_r and 0_r are layers parametrised on r , which uniquely identifies the shared lock at x . In the proof of the implementation, one may have internal layers which do not leak, but that need to be kept parametric on r . To be able to associate unambiguously a layer to obligations, the obligations need to be parametric on r as well, leading to the pattern $\text{lay}(\mathbf{O}(r, \vec{a})) = m_r$. To remove the noise generated by this uniform parametrisation on the region identifiers, we omit it from the proofs of the implementation, writing simply $\text{lay}(\mathbf{O}(\vec{a})) = m$, with the understanding that the proof can be unambiguously parametrised.

A.5 (Meta-)Quantification

Following standard conventions, free meta-variables are implicitly universally quantified. For example, we may write

$$\mathbf{A}(x) \bullet \mathbf{B}(y) \neq \perp \Leftrightarrow x = y$$

to denote the set of axioms

$$\forall x \in \text{dom}(\mathbf{A}), y \in \text{dom}(\mathbf{B}). \mathbf{A}(x) \bullet \mathbf{B}(y) \neq \perp \Leftrightarrow x = y$$

A.6 Region type specifications

– *Abstract state domain.* It can be tedious (and detrimental to readability) to always explicitly write the domains of quantified variables in the assertions of program proofs, especially when they can be easily inferred from context. Consider the case of regions. Some of the rules, for example **MkATOM**, need the precise domain of the abstract state ($\exists x \in X$) because it needs to match the pseudo-quantifier’s domain ($\forall x \in X$). To improve readability, we adopt the following strategy. Suppose the region type t has abstract state in the domain A . We can define the interpretation function so that it constrains the domain of the abstract state accordingly: $\mathcal{I}(t_r^\lambda(a)) = a \in A \wedge \dots$. Then we trivially have that $\lambda'; \mathcal{A} \vDash \exists a. t_r^\lambda(a) \Rightarrow \exists a \in A. t_r^\lambda(a)$. We thus can omit the domains from existential quantification and implicitly apply rule **CONS** whenever the domain information is needed in the proof.

To ease even further the specification of region types, when defining a new region type we will introduce, once and for all, the domain of the corresponding abstract state, and omit the obvious constraint from the interpretation definition.

– *Fixed parameters.* It is very common to have a product domain as abstract state of regions, as one needs to assemble in an abstract state many bits of information that characterise region’s state. Typically, the abstract state domain A can be seen as the product of two domains $F \times S$, the domain of the *fixed parameters* F and the domain of the *non-fixed parameters* S . (Both F and S can be themselves products of simpler domains.) The fixed parameters are set at the point of creation of the region, and can never be updated; they typically define the “interface” of the region. For example, the address of a lock module instance x is the fixed parameter of a region $\text{lock}_r(x, l)$ and $l \in \{0, 1\}$ is the non-fixed parameter representing the state of the lock. When introducing a new region type we will specify which parameters are fixed, and they will be omitted from the region interference specification, as they are left untouched by every transition. For example, for the hypothetical region $\text{lock}_r(x, l)$ above, we may write $\mathbf{G} : (0, \mathbf{0}) \rightsquigarrow (1, \mathbf{k})$ and $\mathbf{G} : (1, \mathbf{k}) \rightsquigarrow (0, \mathbf{0})$ to denote $\mathbf{G} : ((x, 0), \mathbf{0}) \rightsquigarrow ((x, 1), \mathbf{k})$ and $\mathbf{G} : ((x, 1), \mathbf{k}) \rightsquigarrow ((x, 0), \mathbf{0})$.

– *Interference protocols and atomicity contexts.* Definition E.8 requires \mathcal{T}_t to be monotone in the guards, reflexive and closed under obligation frames. Since writing the whole function can be tedious and redundant, we will only write a number of expressions of the form

$$\mathbf{G} : (a_1, O_1) \rightsquigarrow (a_2, O_2) \tag{7}$$

which will set $\mathcal{T}_t(G) \ni \{(a_1, O_1), (a_2, O_2)\}$, and implicitly complete the function by closing \mathcal{T}_t under the properties above.

Similarly, atomicity contexts associate to some region identifier records $\mathcal{A}(r) = (X, k, X', R)$ that have (unguarded) transition relations as their last component R . We therefore borrow the syntax from (7), and write $R = (a_1, O_1) \rightsquigarrow (a_2, O_2)$ to specify R as the minimal relation that include such relations and is closed under obligation frames.

A.7 Proof patterns

There are some recurring patterns in TaDA Live proofs, which we summarise here to help the reader navigate the examples.

– *The exclusive guard.* It is very common to express some exclusive permission on some shared resource by means of some guard \mathbf{E} that cannot be composed with itself: i.e. $\mathbf{E} \bullet \mathbf{E} = \perp$. Local ownership of \mathbf{E} is exclusive in that no other thread can at the same time assert ownership of \mathbf{E} . An ubiquitous use of this guard is in representing the resource offered by a module. Take for example a concurrent counter module. Abstractly we have a (fixed) location x for the module instance and an abstract state $n \in \mathbb{N}$ representing the current value of the counter. Since this is a concurrent counter it uses internally shared resources. We therefore have a region $\mathbf{cnt}_r(x, n)$ encapsulating the shared internal resources of the counter. From the perspective of the client, however, at the moment of creation of the counter with, say, an operation $\text{makeCounter}()$, the counter is exclusively owned by the client. This, for example, is reflected in the fact that, until the client shares the counter or invokes operations on it, the value of the counter will be stably 0. To represent this fact, one typically defines an exclusive guard \mathbf{E} guarding each transition of the region interference: e.g. $\mathbf{E} : (n, O_1) \rightsquigarrow (m, O_2)$. Then the $\text{makeCounter}()$ operation can be given the specification

$$\vdash \{ \mathbf{emp} \} x := \text{makeCounter}() \{ \mathbf{cnt}_r(x, 0) * \lceil \mathbf{E} \rceil_r \}$$

which gives to the client the stable assertion $\mathbf{cnt}_r(x, 0) * \lceil \mathbf{E} \rceil_r$. (Note how $\mathbf{cnt}_r(x, 0)$ is not stable.) To re-share the counter, the client will create its own region encoding the invariants governing the interaction over the counter (and the other resources of the client) the interpretation of which will contain $\mathbf{cnt}_r(x, 0) * \lceil \mathbf{E} \rceil_r$.

Note that the assertion $\mathbf{cnt}_r(x, 0) * \lceil \mathbf{E} \rceil_r$ has a very different meaning if occurring in the *atomic* precondition of a triple, as opposed to the *Hoare* precondition: the resources in the atomic precondition are not owned by the local thread, but only acquired instantaneously at the linearisation point. For example, in the triple

$$\vdash \forall n \in \mathbb{N}. \langle \mathbf{cnt}_r(x, n) * \lceil \mathbf{E} \rceil_r \rangle \text{incr}(x) \langle \mathbf{cnt}_r(x, n+1) * \lceil \mathbf{E} \rceil_r \rangle$$

the exclusivity of \mathbf{E} is only granted *instantaneously* to the thread acting on it atomically, i.e. either the environment during the interference phase as allowed by the pseudo-quantifier, or the local thread at the linearisation point.

Since this pattern is ubiquitous, we reserve the \mathbf{E} guard constructor for this use, and will omit the $\mathbf{E} \bullet \mathbf{E} = \perp$ axiom when specifying guard algebras.

A.8 Modules

TaDA Live is a logics that emphasizes modularity of the proofs. One aspect of this is that when a program is naturally structured as a collection of modules, one would want the proof of correctness to be decomposed into independent proofs of each module exporting some specifications for the externally accessible operations, and a proof that the client of these modules is correct, which depends only on these abstract module specifications.

In our model, a module is nothing but a conceptually related set of operations f_1, \dots, f_n that are defined in a **let** statement: **let** $f_1(\vec{x}_1) = \mathbb{C}_1$ **in** \dots **let** $f_n(\vec{x}_n) = \mathbb{C}_n$ **in** \mathbb{C} . Here \mathbb{C} is what we call “client” of a module offering operations f_1, \dots, f_n . The operation deals with let statements by populating a function φ associating each function name f_i to its formal parameters \vec{x}_i and its implementation \mathbb{C}_i .

Similarly, the proof of correctness of \mathbb{C} , will need to fetch the abstract specifications of the functions (which appear as free names in \mathbb{C}) from some mapping Φ from function names to their specifications. The fact that the implementation of each operation satisfies its specification is checked in the proof derivation for the let statement (rule **LET**) but then the proof of the client and of the module are done separately.

For this reason, we present proofs of just a module against its abstract specifications, which can be used as if they were axioms in the proof of any client using them. To talk about modules independently of their clients we introduce the notation **def** $f(x) \{ \mathbb{C} \}$ which can be understood as populating an entry of φ for f . We will then prove some specification for f which will populate an entry of Φ for f .

In the proof of some client, we will recall the module specifications that are assumed in Φ , and use rule **CALL** to handle the calls to the operations of the module. We will omit from the proof outlines Φ and the applications of rule **CALL** for readability.

A.9 Proof outlines

In program proof outlines, we adopt a number of notational conventions. First, unless it involves a viewshift or we want to highlight it, we will apply rule **CONS** without mentioning it. Similarly, we omit the obvious applications of rules **VAR**, **CALL** and **SUBPQ** and the axioms (i.e. the rules associated with primitive commands).

Next, in outline such as

$$\begin{array}{c}
 \mathcal{A}; k \vdash \\
 \forall x \in X \rightarrow X'. \\
 \langle P(x) \rangle \\
 \text{OUTER} \left| \begin{array}{c} \langle P'(x) \rangle \\ \vdots \\ \langle Q'(x) \rangle \end{array} \right. \\
 \langle Q(x) \rangle
 \end{array}
 \quad
 \frac{\begin{array}{c} \vdots \\ \mathcal{A}; k \vdash \forall x \in X \rightarrow X'. \langle P'(x) \rangle \mathbb{C} \langle Q'(x) \rangle \end{array}}{\mathcal{A}; k \vdash \forall x \in X \rightarrow X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle}
 \begin{array}{l}
 \text{INNER} \\
 \text{OUTER}
 \end{array}$$

the specification of the inner step inherits the context and the pseudo-quantifier of the specification of the outer step, as in the derivation on the right.

B THE TADA LIVE PROOF SYSTEM

In this section, we present the full proof system of TaDA Live.

For brevity we use the metavariable \vec{X} to range over expressions of the form $X_1 \rightarrow_k X_2$ and is used in rules when the pseudo-quantification is simply copied verbatim from premise to conclusion.

In the rules we use the following abbreviations:

$$\begin{aligned} \vdash_{\mathcal{A}} P \triangleright k &\triangleq \forall r \in \text{Rld}. \vdash_{\mathcal{A}} P \Rightarrow r \triangleright k \\ \vdash_{\mathcal{A}} P \ni [O]_r^L &\triangleq \vdash_{\mathcal{A}} P \Rightarrow [O]_r^L * \text{emp}_{\text{Ob}}^r \end{aligned}$$

B.1 Liveness rules

For reference we reproduce the liveness-related rules.

$$\begin{aligned} &\frac{n; \lambda; \mathcal{A} \vdash \exists x \in X. P(x) \xrightarrow{M} \exists x' \in X'. P(x') \quad m, k \geq n \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle} \text{LIVEC} \\ &\frac{\forall \beta \leq \beta_0. m(\beta); \lambda; \mathcal{A} \vdash L \xrightarrow{M} T(\beta) \quad \forall \beta \leq \beta_0. \vdash_{\mathcal{A}} P(\beta) \triangleright m(\beta) \geq m \quad \forall \alpha. \mathcal{A} \models \exists \alpha'. L * M(\alpha') \wedge \alpha' \leq \alpha \text{ stable} \quad \text{pv}(T, L, M) \cap \text{mod}(\mathbb{C}) = \emptyset \quad \forall \beta \leq \beta_0. \forall b \in \text{Bool}. m; \lambda; \mathcal{A} \vdash_{\Phi} \{P(\beta) * (b \Rightarrow T(\beta)) \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma. P(\gamma) \wedge \gamma \leq \beta * (b \Rightarrow \gamma < \beta)\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P(\beta_0) * L\} \text{while}(\mathbb{B})\{\mathbb{C}\} \{\exists \beta. P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta \leq \beta_0\}} \text{WHILE} \\ &\frac{m_1; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1\} \mathbb{C}_1 \{Q_1\} \quad \vdash_{\mathcal{A}} Q_1 \triangleright m_2 \leq m \quad m_2; \lambda; \mathcal{A} \vdash_{\Phi} \{P_2\} \mathbb{C}_2 \{Q_2\} \quad \vdash_{\mathcal{A}} Q_2 \triangleright m_1 \leq m}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1 * P_2\} \mathbb{C}_1 \parallel \mathbb{C}_2 \{Q_1 * Q_2\}} \text{PAR} \end{aligned}$$

B.1.1— The Environment liveness rules. The Environment liveness rules use the $\text{impr}_{\mathcal{A}}$ condition defined as follows:

Definition B.1 ($\text{impr}_{\mathcal{A}}$). Given assertions $L(\alpha)$, $L'(\alpha)$ and T , and relations $R, R' \subseteq \text{AState} \times \text{AState}$, the condition $\text{impr}_{\mathcal{A}}(\mathbf{t}_r^{\lambda}, L, L', R, R', T)$ holds if and only if, $R' \neq \emptyset$ and for all $w_1, w_2 \in \text{World}_{\mathcal{A}}$, α_1, α_2 , and $(x_1, x_2) \in R$:

$$\begin{aligned} (w_1 \models_{\mathcal{A}} L(\alpha_1) * \mathbf{t}_r^{\lambda}(x_1) \wedge w_2 \models_{\mathcal{A}} L'(\alpha_2) * \mathbf{t}_r^{\lambda}(x_2)) \\ \Rightarrow ((\alpha_2 \leq \alpha_1 \wedge ((x_1, x_2) \in R' \Rightarrow \alpha_2 < \alpha_1)) \vee w_2 \models_{\mathcal{A}} T * \text{True}) \end{aligned}$$

Intuitively, $L(\alpha) * \mathbf{t}_r^{\lambda}(x_1)$ describes configurations with abstract state x_1 and environment progress measure α , corresponding to the current subset of possible configurations taken as starting point. $L'(\alpha) * \mathbf{t}_r^{\lambda}(x_1)$ describes the same, but for the possible end-point configurations (the whole of the invariant). R describes the possible abstract state transitions, R' describes the possible abstract state transitions to a good state, T describes the target configurations. The $\text{impr}_{\mathcal{A}}(\mathbf{t}_r^{\lambda}, L, R, R', T)$ condition then encodes the check that

- (1) there is a transition to a good state possible ($R' \neq \emptyset$)
- (2) if we compare the measure α_1 and α_2 , taken before and after a transition in R occurred, respectively, then we obtain that either (a) the measure did not increase and it decreased if it was a transition to a good state, or (b) we reached the target T

$$\frac{\mathcal{A} \models L \text{ stable} \quad \vdash_{\mathcal{A}} L \Rightarrow L * \exists \alpha. M(\alpha) \quad m; \lambda; \mathcal{A} \vdash L * M(\alpha) : L * M(\alpha) \xrightarrow{T}}{m; \lambda; \mathcal{A} \vdash L \xrightarrow{M} T} \text{ENVLIVE}$$

$$\begin{array}{c}
\frac{m; \lambda; \mathcal{A} \vdash L(\alpha) : L_1(\alpha) \rightarrow T \quad m; \lambda; \mathcal{A} \vdash L(\alpha) : L_2(\alpha) \rightarrow T}{m; \lambda; \mathcal{A} \vdash L(\alpha) : L_1(\alpha) \vee L_2(\alpha) \rightarrow T} \text{ECASE} \quad \frac{\forall \alpha. \vdash_{\mathcal{A}} T'(\alpha) \Rightarrow T}{m; \lambda; \mathcal{A} \vdash L(\alpha) : T'(\alpha) \rightarrow T} \text{LIVET} \\
\\
\frac{\text{impr}_{\mathcal{A}}(\mathfrak{t}_r^\lambda, L, L', \text{io}(R), R', T) \quad m > \text{lay}(O) \quad \forall \alpha. \vdash_{\mathcal{A}} L(\alpha) \triangleright \text{lay}(O) \quad O \in \text{AOB} \quad \lambda < \lambda' \quad R = \bigcup \{ \mathcal{T}_i(G') \mid G' \# G \} \quad R' = \{ (a_1, a_2) \mid ((a_1, O_1), (a_2, O_2)) \in R, O \sqsubseteq O_1, O \not\sqsubseteq O_2 \}}{m; \lambda'; \mathcal{A} \vdash L'(\alpha) : L(\alpha) * \exists x. \mathfrak{t}_r^\lambda(x) * [G]_r * [O]_r^E \rightarrow T} \text{LIVEO} \\
\\
\frac{\text{impr}_{\mathcal{A}}(\mathfrak{t}_r^\lambda, L, L', R, R', T) \quad m > k \quad \forall \alpha. \vdash_{\mathcal{A}} L(\alpha) \triangleright k \quad \lambda < \lambda' \quad (X \rightarrow_k X') = \text{live}(\mathcal{A}, r) \quad R = \bigcup \{ \text{io}(\mathcal{T}_i(G')) \mid G' \# G \} \quad R' = \{ (a_1, a_2) \in R \mid a_2 \in X' \}}{m; \lambda'; \mathcal{A} \vdash L'(\alpha) : L(\alpha) * \exists x. \mathfrak{t}_r^\lambda(x) * [G]_r * r \Rightarrow \diamond \rightarrow T} \text{LIVEA}
\end{array}$$

B.2 General forms

The following rules are the general forms of some of the rules in Fig. 6.

$$\begin{array}{c}
\frac{\forall x \in X. \vdash_{\mathcal{A}} R_h * R_a(x) \triangleright m \quad \text{fv}(R_h, R_a(x)) \cap \text{mod}(\mathbb{C}) = \emptyset \quad \mathcal{A} \vDash R_h \text{ stable} \quad \forall x \in X. \mathcal{A} \vDash R_a(x) \text{ stable}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{FRAME} \\
\\
\frac{\lambda < \lambda' \quad r \notin \text{dom}(\mathcal{A}) \quad \mathcal{A}' = \mathcal{A}[r \mapsto (X, k, X', T)] \quad T \subseteq \mathcal{T}_i(G) \quad R = \text{io}(T) \quad m; \lambda'; \mathcal{A}' \vdash_{\Phi} \{ P_h * \exists x \in X. \mathfrak{t}_r^\lambda(x) * r \Rightarrow \diamond \} \mathbb{C} \{ \exists x, y. R(x, y) * Q_h(x, y) * r \Rightarrow (x, y) \}}{m; \lambda'; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P_h \mid \mathfrak{t}_r^\lambda(x) * [G]_r \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid \mathfrak{t}_r^\lambda(y) * [G]_r * R(x, y) \rangle} \text{MkATOMG} \\
\\
\frac{r \in \text{dom}(\mathcal{A}) \quad \mathcal{A}' = \mathcal{A}[r \mapsto \perp] \quad \mathcal{A} \vDash Q_h(x, y) \xrightarrow{\lambda} Q'_h(x, y) \quad \mathcal{A} \vDash I(\mathfrak{t}_r^\lambda(z)) * Q_1(x, z) \wedge R(x, z) \xrightarrow{\lambda} Q'_1(x, z) \quad \mathcal{A} \vDash I(\mathfrak{t}_r^\lambda(z)) * Q_2(x, z) \wedge x = z \xrightarrow{\lambda} Q'_2(x, z) \quad \vdash_{\mathcal{A}} P_h \Rightarrow \text{emp}_{\text{Ob}}^r \quad \vdash_{\mathcal{A}} Q_h(x, y) \Rightarrow \text{emp}_{\text{Ob}}^r \quad \vdash_{\mathcal{A}} P(x) \ni [O_0(x)]_r^{\perp} \quad \vdash_{\mathcal{A}} Q_1(x, z) \ni [O_1(x, z)]_r^{\perp} \quad \vdash_{\mathcal{A}} Q_2(x, z) \ni [O_2(x, z)]_r^{\perp} \quad \{ ((x, O_0(x)), (z, O_1(x, z))) \mid x \in X \wedge R(x, z) \} \cup \{ ((x, O_0(x)), (x, O_2(x))) \mid x \in X \} \subseteq \text{tr}(\mathcal{A}, r)}{m; \lambda; \mathcal{A}' \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid I(\mathfrak{t}_r^\lambda(x)) * P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid \exists z. I(\mathfrak{t}_r^\lambda(z)) * \left(\begin{array}{l} R(x, z) \wedge Q_1(x, y, z) \\ \vee x = z \wedge Q_2(x, y) \end{array} \right) \rangle} \text{UpdREGG} \\
\\
\frac{m; \lambda+1; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid \mathfrak{t}_r^\lambda(x) * P_a(x) * r \Rightarrow \diamond \rangle \mathbb{C} \exists y. \langle Q'_h(x, y) \mid \exists z. \mathfrak{t}_r^\lambda(z) * \left(\begin{array}{l} Q'_1(x, y, z) * r \Rightarrow (x, z) \\ \vee Q'_2(x, y) * r \Rightarrow \diamond \end{array} \right) \rangle}{\mathcal{A} \vDash Q_h(x, y) \xrightarrow{\lambda} Q'_h(x, y) \quad \mathcal{A} \vDash I(\mathfrak{t}_r^\lambda(z)) * Q_a(x, y, z) \wedge R(x, y, z) \xrightarrow{\lambda} Q'_a(x, y, z) \quad \vdash_{\mathcal{A}} P_h \Rightarrow \text{emp}_{\text{Ob}}^r \quad \vdash_{\mathcal{A}} Q_h(x, y) \Rightarrow \text{emp}_{\text{Ob}}^r \quad \vdash_{\mathcal{A}} P_a(x) \ni [O_1(x)]_r^{\perp} \quad \vdash_{\mathcal{A}} Q_a(x, y, z) \ni [O_2(x, z)]_r^{\perp} \quad r \in \text{dom}(\mathcal{A}) \Rightarrow R = \text{id} \quad R(x, z) \Rightarrow ((x, O_1(x)), (z, O_2(x, z))) \in \mathcal{T}_i(G)}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid I(\mathfrak{t}_r^\lambda(x)) * P_a(x) * [G]_r \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid \exists z. I(\mathfrak{t}_r^\lambda(z)) * Q_a(x, y, z) \wedge R(x, z) \rangle} \text{LIFTAG} \\
\\
\frac{m; \lambda+1; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid \mathfrak{t}_r^\lambda(x) * P_a(x) * [G]_r \rangle \mathbb{C} \exists y. \langle Q'_h(x, y) \mid \exists z. \mathfrak{t}_r^\lambda(z) * Q'_a(x, y, z) \rangle}{\mathcal{A} \vDash P_h * P \text{ stable} \quad \forall x \in X, y. \mathcal{A} \vDash Q_h(x, y) * Q(x, y) \text{ stable}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P * P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q(x, y) * Q_a(x, y) \rangle} \text{ATOMWG} \\
\\
\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h * P \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) * Q(x, y) \mid Q_a(x, y) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}, z \in Z. \langle P_h \mid P_a(x, z) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y, z) \rangle} \text{A}\exists\text{ELIMG} \\
\\
\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid \exists z \in Z. P_a(x, z) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid \exists z \in Z. Q_a(x, y, z) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid \exists z \in Z. P_a(x, z) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid \exists z \in Z. Q_a(x, y, z) \rangle} \text{A}\exists\text{ELIMG}
\end{array}$$

$$\frac{k_1; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle \quad k_1 \leq k_2}{k_2; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{LAYWG}$$

B.3 Logical manipulation rules

The rules below allow for basic logical manipulation.

$$\frac{\begin{array}{l} \lambda; \mathcal{A} \vDash P_h \Rightarrow P'_h \quad \forall x \in X, y. \lambda; \mathcal{A} \vDash Q'_h(x, y) \Rightarrow Q_h(x, y) \\ \forall x \in X. \lambda; \mathcal{A} \vDash P_a(x) \Rightarrow P'_a(x) \quad \forall x \in X, y. \lambda; \mathcal{A} \vDash Q'_a(x, y) \Rightarrow Q_a(x, y) \\ m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P'_h \mid P'_a(x) \rangle \mathbb{C} \exists y. \langle Q'_h(x, y) \mid Q'_a(x, y) \rangle \end{array}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{CONS}$$

$$\frac{\forall v \in X. m; \lambda; \mathcal{A} \vdash_{\Phi} \{P(v)\} \mathbb{C} \{Q\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{\exists x \in X. P(x)\} \mathbb{C} \{Q\}} \text{\exists ELIM}$$

$$\frac{\forall k \leq m. k; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h(k) \wedge k \leq m \mid P_a(k, x) \rangle \mathbb{C} \exists y. \langle Q_h(k, x, y) \mid Q_a(k, x, y) \rangle}{\forall k \leq m. m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h(k) \mid P_a(k, x) \rangle \mathbb{C} \exists y. \langle Q_h(k, x, y) \mid Q_a(k, x, y) \rangle} \text{QL}$$

$$\frac{\begin{array}{l} f: X \rightarrow Y \quad Y' = f(X') \quad \forall x \in X. \vdash_{\mathcal{A}} P'_a(x) \Rightarrow P_a(f(x)) \\ \forall x \in X, z. \vdash_{\mathcal{A}} Q_h(f(x), z) \Rightarrow Q'_h(x, z) \quad \forall x \in X, z. \vdash_{\mathcal{A}} Q_a(f(x), z) \Rightarrow Q'_a(x, z) \\ m; \lambda; \mathcal{A} \vdash_{\Phi} \forall y \in Y \rightarrow_k Y'. \langle P_h \mid P_a(y) \rangle \mathbb{C} \exists z. \langle Q_h(y, z) \mid Q_a(y, z) \rangle \end{array}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P_h \mid P'_a(x) \rangle \mathbb{C} \exists z. \langle Q'_h(x, z) \mid Q'_a(x, z) \rangle} \text{SUBPQ}$$

$$\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle}{m; \lambda; \mathcal{A} \cup \mathcal{A}' \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{ACEXT}$$

$$\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X''. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle \quad X' \subseteq X'' \subseteq X}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \mathbb{C} \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{LIVEW}$$

B.4 Axioms

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{\mathbb{E} \geq 0\} \ x := \mathbf{alloc}(\mathbb{E}) \ \left\{ \ast_{i=0}^{\mathbb{E}-1} x + i \mapsto _ \right\}} \text{ALLOC}$$

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{\mathbb{E} \mapsto _ \} \ \mathbf{dealloc}(\mathbb{E}) \ \{\mathbf{emp}\}} \text{DEALLOC}$$

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall v. \langle \mathbb{E} \mapsto v \rangle \ x := [\mathbb{E}] \ \langle \mathbb{E} \mapsto v \wedge x = v \rangle} \text{READ}$$

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall v. \langle \mathbb{E}_1 \mapsto v \rangle \ [\mathbb{E}_1] := \mathbb{E}_2 \ \langle \mathbb{E}_1 \mapsto \mathbb{E}_2 \rangle} \text{MUTATE}$$

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall v. \langle \mathbb{E}_1 \mapsto v \rangle \ x := \mathbf{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \ \left\langle \begin{array}{l} (x = 1 \wedge \mathbb{E}_1 \mapsto \mathbb{E}_3 \wedge v = \mathbb{E}_2) \vee \\ (x = 0 \wedge \mathbb{E}_1 \mapsto v \wedge v \neq \mathbb{E}_2) \end{array} \right\rangle} \text{CAS}$$

$$\frac{}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall v. \langle \mathbb{E}_1 \mapsto v \rangle \ x := \mathbf{FAS}(\mathbb{E}_1, \mathbb{E}_2) \ \langle \mathbb{E}_1 \mapsto \mathbb{E}_2 \wedge x = v \rangle} \text{FAS}$$

B.5 Standard Hoare rules

$$\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \mathbb{C}_1 \{R\} \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \{R\} \mathbb{C}_2 \{Q\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \mathbb{C}_1; \mathbb{C}_2 \{Q\}} \text{SEQ}$$

$$\frac{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P \wedge \mathbb{B}\} \mathbb{C}_1 \{Q\} \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \{P \wedge \neg \mathbb{B}\} \mathbb{C}_2 \{Q\}}{m; \lambda; \mathcal{A} \vdash_{\Phi} \{P\} \ \mathbf{if}(\mathbb{B}) \{ \mathbb{C}_1 \} \mathbf{else} \{ \mathbb{C}_2 \} \{Q\}} \text{IF}$$

$$\begin{array}{c}
\frac{x \notin \text{fv}(P_h) \cup \text{fv}(Q_h) \cup \text{fv}(\mathbb{E}) \quad m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \wedge x = \mathbb{E} \mid P_a(x) \rangle \mathbb{C} \langle Q_h(x, y) \mid Q_a(x, y) \rangle}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \text{ var } x = \mathbb{E} \text{ in } \mathbb{C} \langle Q_h(x, y) \mid Q_a(x, y) \rangle} \text{VAR} \\
\frac{(\vec{x}, \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y, \text{ret}) \mid Q_a(x, y) \rangle)_{m; \lambda; \mathcal{A}} \in \Phi(f)}{m; \lambda; \mathcal{A} \vdash_{\Phi} \forall x \in \vec{X}. \langle P_h[\vec{\mathbb{E}}/\vec{x}] \mid P_a(x) \rangle \text{ z} := f(\vec{\mathbb{E}}) \exists y. \langle Q_h(x, y, z) \mid Q_a(x, y) \rangle} \text{CALL} \\
\frac{\text{pv}(\mathbb{S}) \subseteq \vec{x} \cup \{\text{ret}\} \quad f \notin \text{dom}(\Phi) \quad \Phi' = \Phi[f \mapsto (\vec{x}, \mathbb{S})] \quad \vdash_{\Phi} \mathbb{C}_1 : \mathbb{S}_1 \quad \vdash_{\Phi'} \mathbb{C}_2 : \mathbb{S}_2}{\vdash_{\Phi} \text{let } f(\vec{x}) = \mathbb{C}_1 \text{ in } \mathbb{C}_2 : \mathbb{S}_2} \text{LET} \\
\frac{P, Q \in \text{SL} \quad \forall x \in X. \perp; 0; \emptyset \vdash_{\Phi} \{P(x)\} \mathbb{C} \{Q(x)\}}{\perp; 0; \emptyset \vdash_{\Phi} \forall x \in X. \langle P(x) \rangle \langle \mathbb{C} \rangle \langle Q(x) \rangle} \text{PRAT}
\end{array}$$

B.6 On Stability Checks

A triple is well-defined, according to Definition E.19, if the Hoare pre- and post-conditions are both stable assertions. The rules all assume the triples in the premises are well-defined and ensure that the triple in the conclusion is well-defined as well. The only exceptions are rules **MkATOMG**, **CONS**, **SUBPQ**, and **ΞELIM**, where the Hoare pre-/post-conditions should be checked for stability to ensure the conclusion is a well-defined triple. We omitted these stability checks from these rules to improve readability.

In practice, however, this way of handling stability has a drawback: if one starts with a goal that has unstable pre-/post-conditions, one would only see the mistake much further up in the proof, typically at applications of **ATOMW** or **FRAME** (which requires stability of the frames) just before applications of the axioms. Therefore, in practice, to make the proof fail early in case of mistakes, one would want to additionally check stability at the top-level goal, and applications of **PAR**.

C EXAMPLES

This section contains the full details of the proofs of total correctness of some representative examples. We start with the spin lock and CLH lock implementations, followed by Example 3.1. Finally, we consider a bigger more challenging example: a concurrent set implemented using the lock-coupling protocol.

Note that the presentation of the program proofs follows the notation and conventions introduced in Appendix A.

C.1 Spin lock

– *Code.* The spin lock module implements a lock by storing a single bit in a heap cell; locking is implemented by trying to CAS the bit from 0 to 1 until the CAS succeeded, unlocking simply sets the bit back to 0.

```

def makeLock() {
  ret := alloc(1);
  [ret] := 0;
}

def lock(x) {
  var d = 0 in
  while(d = 0){
    d := CAS(x, 0, 1);
  }
}

def unlock(x) {
  [x] := 0;
}

```

– *Specifications.* We will prove the module satisfies the specifications:

$$\begin{aligned}
&\forall \alpha. \mathbf{0} \vdash \{\text{emp}\} \text{makeLock}() \{\exists r. \mathbf{L}(r, \text{ret}, 0, \alpha)\} \\
&\forall \phi. \mathbf{1} \vdash \mathbf{W}l \in \{0, 1\} \rightarrow_0 \{0\}, \alpha. \langle \mathbf{L}(r, x, l, \alpha) \wedge \alpha > \phi(\alpha) \rangle \text{lock}(x) \langle \mathbf{L}(r, x, 1, \phi(\alpha)) \wedge l = 0 \rangle \\
&\mathbf{0} \vdash \langle \mathbf{L}(r, x, 1, \alpha) \rangle \text{unlock}(x) \langle \mathbf{L}(r, x, 0, \alpha) \rangle
\end{aligned}$$

where $\mathbf{L}(r, x, l, \alpha)$ abstractly represents the lock resource at abstract location r and concrete address x , with abstract state $l \in \{0, 1\}$ and impedance budget α (an ordinal). Here $\phi: \mathbb{O} \rightarrow \mathbb{O}$ is a function that can be freely instantiated by the client upon usage of the specification, and it indicates precisely how much the budget will decrease after this call (which is client dependent information). Note how the specification of `makeLock` allows the client to pick an arbitrary ordinal as the initial budget.

– *Region Types.* The abstract shared lock resource will be represented by a region $\text{spin}_r(x, l, \alpha)$ where $x \in \text{Addr}$, $l \in \{0, 1\}$, $\alpha \in \mathbb{O}$. Here x is a fixed parameter of the region. The lock resource is abstractly represented by the predicate $\mathbf{L}(r, x, l, \alpha) \triangleq \text{spin}_r(x, l, \alpha) * [\mathbf{E}]_r$.

– *Guards and Obligations.* For the `spin` region we only the exclusive guard \mathbf{E} , and no obligation constructors, as the implementation has no internal blocking. All the blocking behaviour is represented by the liveness assumption in the pseudo-quantifier of the specification of `lock`.

– *Interference protocol.* The interference protocol for `spin` is very simple:

$$\begin{aligned}
\mathbf{E}: ((0, \alpha), \mathbf{0}) &\rightsquigarrow ((1, \beta), \mathbf{0}) \text{ only if } \beta < \alpha \\
\mathbf{E}: ((1, \alpha), \mathbf{0}) &\rightsquigarrow ((0, \alpha), \mathbf{0})
\end{aligned}$$

It states that whoever owns \mathbf{E} can freely acquire or release the lock, provided that at each acquisition, some budget is spent ($\beta < \alpha$).

– *Region interpretation.* The implementation uses a single bit stored in the heap, and we have no non-trivial guards/obligations; the interpretation is thus straightforward:

$$I(\text{spin}_r(x, l, \alpha)) \triangleq x \mapsto l$$

PROOF OF lock(x):	
	$1; \emptyset \vdash \forall l \in \{0, 1\} \rightarrow_0 \{0\}, \alpha.$
	$\langle L(r, x, l, \alpha) \wedge \alpha > \phi(\alpha) \rangle$
	$\langle \text{spin}_r(x, l, \alpha) * [E]_r \wedge \alpha > \phi(\alpha) \rangle$
	$1; [r \mapsto \{0, 1\} \times \mathbf{0}, \{0\} \times \mathbf{0}, ((0, \alpha), \mathbf{0}) \rightsquigarrow ((1, \phi(\alpha)), \mathbf{0})] \vdash$
	$\langle \exists l, \alpha. \text{spin}_r(x, l, \alpha) \wedge \alpha > \phi(\alpha) * r \Rightarrow \blacklozenge \rangle$
	var d=0 in
	$\langle \exists l, \alpha. \text{spin}_r(x, l, \alpha) \wedge \alpha > \phi(\alpha) * r \Rightarrow \blacklozenge \wedge d = 0 \rangle$
	while (d=0){
	$\forall b \in \{0, 1\}, \beta.$
	$\langle \exists l, \alpha. \text{spin}_r(x, l, \alpha) \wedge \alpha > \phi(\alpha) \wedge \beta \geq \alpha \wedge$
	$\wedge b \Rightarrow (l = 0 \vee \beta > \alpha) * r \Rightarrow \blacklozenge \wedge d = 0 \rangle$
	$\forall l \in \{0, 1\}, \alpha.$
	$\langle \text{spin}_r(x, l, \alpha) \wedge \alpha > \phi(\alpha) \wedge \beta \geq \alpha \wedge$
	$b \Rightarrow (l = 0 \vee \beta > \alpha) * r \Rightarrow \blacklozenge \wedge d = 0 \rangle$
	$\langle x \mapsto l \wedge \alpha > \phi(\alpha) \wedge \beta \geq \alpha \rangle$
	$\langle b \Rightarrow (l = 0 \vee \beta > \alpha) \rangle$
	$\langle x \mapsto l \rangle$
	$d := \text{CAS}(x, \emptyset, 1);$
	$\langle x \mapsto 1 \wedge ((d = 0 \wedge l = 1) \vee (d = 1 \wedge l = 0)) \rangle$
	$\langle \exists \delta. x \mapsto 1 \wedge \left(\begin{array}{l} (d = 0 \wedge l = 1 \wedge \delta = \alpha \wedge \alpha > \phi(\alpha) \wedge b \Rightarrow \beta > \alpha) \\ \vee (d = 1 \wedge l = 0 \wedge \delta = \phi(\alpha) \wedge \beta > \phi(\alpha)) \end{array} \right) \rangle$
	$\langle \exists \delta. \text{spin}_r(x, 1, \delta) * \left(\begin{array}{l} (d = 0 \wedge l = 1 \wedge \delta > \phi(\delta) \wedge b \Rightarrow \beta > \delta \wedge r \Rightarrow \blacklozenge) \\ \vee (d = 1 \wedge l = 0 \wedge \beta > \delta \wedge r \Rightarrow ((l, \alpha), (1, \phi(\alpha)))) \end{array} \right) \rangle$
	$\langle \exists \gamma, \delta. \text{spin}_r(x, 1, \delta) \wedge \gamma \geq \delta \wedge b \Rightarrow \gamma < \beta$
	$* \left(\begin{array}{l} (d = 0 \wedge \delta > \phi(\delta) \wedge r \Rightarrow \blacklozenge) \\ \vee (d = 1 \wedge l = 0 \wedge r \Rightarrow ((l, \alpha), (1, \phi(\alpha)))) \end{array} \right) \rangle$
	$\langle \exists l', \alpha', \gamma. \text{spin}_r(x, l', \alpha') \wedge \gamma \geq \alpha' \wedge b \Rightarrow \gamma < \beta$
	$* \left(\begin{array}{l} (d = 0 \wedge \alpha' > \phi(\alpha') \wedge r \Rightarrow \blacklozenge) \\ \vee (\exists l, \alpha. d = 1 \wedge r \Rightarrow ((l, \alpha), (1, \phi(\alpha))) \wedge l = 0) \end{array} \right) \rangle$
	\rangle
	$\langle \exists l, \alpha. \text{spin}_r(x, _ , _) * r \Rightarrow ((l, \alpha), (1, \phi(\alpha))) \wedge l = 0 \rangle$
	$\langle \text{spin}_r(x, 1, \phi(\alpha)) * [E]_r \wedge l = 0 \rangle$
	$\langle L(r, x, 1, \phi(\alpha)) \wedge l = 0 \rangle$

Fig. 9. Spin lock: proof of lock.

PROOF OF makeLock():	PROOF OF unlock(x):
$0; \emptyset \vdash$	$0; \emptyset \vdash$
$\{\text{emp}\}$	$\langle L(r, x, 1, \alpha) \rangle$
$\{\text{emp}\}$	$\langle \text{spin}_r(x, 1, \alpha) * [E]_r \rangle$
$\text{ret} := \text{alloc}(1);$	$\langle x \mapsto 1 \rangle$
$[\text{ret}] := \emptyset;$	$[x] := \emptyset;$
$\{\text{ret} \mapsto 0\}$	$\langle x \mapsto 0 \rangle$
$\{\exists r. L(r, \text{ret}, 0, \alpha)\}$	$\langle \text{spin}_r(x, 0, \alpha) * [E]_r \rangle$
	$\langle L(r, x, 0, \alpha) \rangle$

Fig. 10. Spin lock: proof of makeLock and unlock. Here STEP 4 is LIFTA, FRAME, SUBPQ.

Note how α is pure ghost state in that it is not linked to any information in the concrete memory. This interpretation is trivially valid, as the only obligation is $\mathbf{0}$.

– *Proof of lock.* Figure 9 shows the outline of the proof of the lock operation. The only step that involves reasoning about termination is **STEP 3**, which applies rules **CONS**, **∃ELIM**, **WHILE** as justified below.

The loop invariant is

$$P(\beta) \triangleq \exists l', \alpha'. \mathbf{spin}_r(x, l', \alpha') \wedge \beta \geq \alpha' * \left(\begin{array}{l} (d = 0 \wedge r \Rightarrow \blacklozenge \wedge \alpha' > \phi(\alpha')) \\ \vee (\exists l, \alpha. d = 1 \wedge r \Rightarrow ((l, \alpha), (1, \phi(\alpha))) \wedge l = 0) \end{array} \right)$$

which contains:

- the safety information to prove the linearisation point, namely that if the **CAS** failed ($d = 0$) then we have not touched the resource yet and we still have permission to perform the linearisation point ($r \Rightarrow \blacklozenge$); whereas if the **CAS** succeeded ($d = 1$) then we did perform the linearisation point with the expected effect.
- the definition of the local variant β as an upper bound on the impedance budget α .

Indeed, whenever some budget is spent, the loop is getting closer to termination as eventually an exhausted budget means no more interference is possible. Without additional information however, we cannot show the local variant will strictly decrease after every iteration: in the case $l = 1$ we cannot exit the loop and the environment is not forced to spend budget. Therefore, the termination argument will need the assumption that the environment always eventually unlocks the lock, which is available in the atomicity context $\mathcal{A} = [r \mapsto (\{0, 1\} \times \mathbb{O}, \mathbf{0}, \{0\} \times \mathbb{O}, R)]$ with $R = ((0, \alpha), \mathbf{0}) \rightsquigarrow ((1, \phi(\alpha)), \mathbf{0})$.

We therefore declare the target states as the ones where, either the linearisation point has been performed, or the lock is unlocked, or some budget was spent:

$$T(\beta) \triangleq \exists l, \alpha. \mathbf{spin}_r(x, l, \alpha) \wedge (r \Rightarrow (_, _) \vee l = 0 \vee \beta > \alpha)$$

The persistent loop invariant here is simply $L = \mathbf{spin}_r(x, _, _)$, which is a valid stable frame of the loop.

Given these parameters, **STEP 3** first establishes the loop invariant holds at the beginning for some β_0 , by applying **CONS**:

$$\begin{aligned} \exists l, \alpha. \mathbf{spin}_r(x, l, \alpha) \wedge \alpha > \phi(\alpha) * r \Rightarrow \blacklozenge \wedge d = 0 &\implies \exists \beta_0. P(\beta_0) * L \\ \exists \beta_0, \beta. P(\beta) * L \wedge d \neq 0 \wedge \beta_0 \geq \beta &\implies \exists \alpha. \mathbf{spin}_r(x, _, _) * r \Rightarrow ((0, \alpha), (1, \phi(\alpha))) \wedge l = 0 \end{aligned}$$

Then **∃ELIM** on β_0 gets rid of the existential quantification, so we are ready to apply **WHILE**.

To apply **WHILE** we need to specify $m(\beta)$, which in this case is simply **1** which satisfies the layer constraints of the rule; and the environment progress measure M :

$$M(\alpha_e) \triangleq \exists l, \alpha. \mathbf{spin}_r(x, l, \alpha) \wedge \alpha_e = 2\alpha + l$$

(here we use the variable α_e for the environment progress measure variable, to avoid clashes with the impedance budget α .)

To complete the application of the rule we need to show

$$\forall \alpha. \mathcal{A} \vDash \exists \alpha'. L * M(\alpha') \wedge \alpha' \leq \alpha \text{ stable} \quad (8)$$

$$\mathbf{1}; \mathcal{A} \vdash L \xrightarrow{M} T(\beta) \quad (9)$$

Condition (8) is easily seen to hold: Suppose, for an arbitrary α , that $\exists \alpha_0. L * M(\alpha_0) \wedge \alpha_0 \leq \alpha$ holds for some world w and consider a world w' such that $w \mathbf{R}_{\mathcal{A}} w'$; certainly w' would satisfy $\exists \alpha_1. L * M(\alpha_1)$ and such α_1 would be such that $\alpha_1 \leq \alpha_0 \leq \alpha$.

Finally, condition (9) is proven as follows. We observe that:

$$\begin{aligned}
L(\alpha_e) &= L * M(\alpha_e) \equiv (\exists l, \alpha. \mathbf{spin}_r(x, l, \alpha) \wedge \alpha_e = 2\alpha + l \wedge (r \Rightarrow (_, _) \vee l = 0)) & (L_1(\alpha_e)) \\
&\vee (\exists \alpha. \mathbf{spin}_r(x, 1, \alpha) \wedge \alpha_e = 2\alpha + 1 \wedge r \Rightarrow \diamond) & (L_2(\alpha_e)) \\
L_2(\alpha_e) &\equiv L(\alpha_e) * [\mathbf{0}]_r * r \Rightarrow \diamond
\end{aligned}$$

We can then derive the environment liveness condition:

$$\frac{\frac{\forall \alpha_e. \vdash_{\mathcal{A}} L_1(\alpha_e) \Rightarrow T(\beta)}{\mathbf{1}; \mathcal{A} \vdash L(\alpha_e) : L_1(\alpha_e) \rightarrow T(\beta)} \text{LIVET} \quad \frac{\text{impr}_{\mathcal{A}}(\mathbf{spin}_r, L_2, L, R, R', T(\beta))}{\mathbf{1}; \mathcal{A} \vdash L(\alpha_e) : L_2(\alpha_e) \rightarrow T(\beta)} \text{LIVEA}}{\frac{\mathbf{1}; \mathcal{A} \vdash L(\alpha_e) : L(\alpha_e) \rightarrow T(\beta)}{\mathbf{1}; \mathcal{A} \vdash L \xrightarrow{M} T(\beta)} \text{ECASE}} \text{ENVLIVE}$$

Intuitively, L_1 encodes the case where we performed the linearisation point or the lock is unlocked, while L_2 the case where we still have not performed the linearisation point and the lock is locked. If L_1 holds then T holds, so no progress of the environment is needed. This is used in the application of rule **LIVET**. If L_2 holds then we can apply rule **LIVEA** to invoke the liveness assumption stored in \mathcal{A} : if the lock is unlocked, α_e strictly decreases.

Formally, the application of **ENVLIVE** requires to prove $\vdash_{\mathcal{A}} L \Rightarrow L * \exists \alpha_e. M(\alpha_e)$ which is trivial. For the application of **LIVEA** we have

$$\begin{aligned}
R &= \{((1, \alpha), (0, \alpha)) \mid \alpha \in \mathbb{O}\} \cup \{((0, \alpha), (1, \beta)) \mid \alpha > \beta\} \\
R' &= \{((1, \alpha), (0, \alpha)) \mid \alpha \in \mathbb{O}\}
\end{aligned}$$

with which it is easy to see that $\text{impr}_{\mathcal{A}}(\mathbf{spin}_r, L_2, L, R, R', T(\beta))$ holds: the transitions in R make α_e strictly decrease in all cases.

To conclude the argument, we briefly comment on the proof of the body of the loop. The applications of rules **UPDREG** and **FRAME** lift the concrete atomic **CAS** to a (potential) update to the \mathbf{spin}_r region. An application of **CONS** allows us to introduce γ as an upper bound to the impedance budget from now on.

Then, we apply rule **AΞELIM** to remove the pseudo-quantification on l and α . At this point, the abstract state l', α' of the region is weakened to any state that might be reached before or after the linearisation point as modified by the environment. We however keep record of what happened exactly at the linearisation point because of the $r \Rightarrow _$ assertions. The later application of **MkATOM** will be able to fetch the atomic update witness $r \Rightarrow ((l, \alpha), (1, \phi(\alpha)))$ and declare the appropriate atomic update in the overall specification. Note that the overall Hoare postcondition after the application of **ATOMW** is stable.

– *Proof of makeLock and unlock.* Figure 10 shows the proof outlines for the makeLock and unlock operations. The only notable step of the proof of makeLock is the last application of **CONS** to viewshift the postcondition from $\text{ret} \mapsto 0$ to $\exists r. \mathbf{spin}_r(x, l, \alpha) * [\mathbf{E}]_r$, which is possible because the interpretation of the region matches with the initial resource, so the reification of the two assertions coincide, and because rule **WR₃** of the world rely allows arbitrary creation of regions to be frame-preserving.

The proof of unlock is a straightforward lifting of the atomic reset of the cell at x to the region \mathbf{spin}_r . Neither proof involves liveness arguments.

C.2 CLH lock

— *Code.* A CLH lock is an implementation of a fair lock module that guarantees fairness by queuing the threads that are waiting to take its possession. This queue is represented by associating each thread queuing for the lock with a cell in memory and having each queuing thread hold a pointer to the cell associated with the thread ahead of it in the queue. The local variables p and c in the lock operation are the addresses of the cell associated with the previous and current threads respectively.

To implement this queue, a tail pointer for the queue is required. When a thread wishes to enqueue itself, a **FAS** operation is performed on the tail pointer, placing the current thread’s cell at the tail of the queue, and returning the address of its predecessor to the current thread.

Each thread’s cell is initially set to the value 1. Once a thread possesses the lock, if it wishes to release it, it may set its associated cell to the value 0, signaling the thread behind it in the queue that it has relinquished the lock. Once signaled, the next thread may take possession of the lock. The while loop in the lock implementation repeatedly reads the current thread’s predecessor’s cell, waiting for it to signal that it has relinquished the lock.

As in most practical implementations, the lock stores, in addition to the tail pointer, a pointer to the current lock holder’s cell. The content of this cell, 1 or 0, indicates the state of the lock, as it is set to 0 once the lock’s owner relinquishes it. By storing a head pointer, the `unlock` operation can access the cell associated with the head to change its value to 0.

```

1 def makeLock() {
2   var x, h in
3   h := alloc(1); [h] := 0;
4   x := alloc(2);
5   [x] := h;
6   [x + 1] := h;
7   ret := x;
8 }

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

1 def unlock(x) {
2   var h in
3   h := [x];
4   [h] := 0;
5 }

```

An interesting aspect of this example is that it features a combination of internal and external blocking: the client needs to always eventually unlock the lock — external blocking, leaks in the pseudo-quantifier — and the lock operation needs to finally take possession of the lock once the previous thread signalled the release of the lock — internal blocking, dealt with using obligations not exposed in the specification. The proof will therefore involve an environment liveness condition proved by using both **LIVEO** and **LIVEA**.

— *Specifications.* We will prove the standard fair lock module specifications:

$$\begin{aligned}
1 &\vdash \forall l \in \{0, 1\} \rightarrow_0 \{0\}. \langle L(r, x, l) \rangle \text{lock}(x) \langle L(r, x, 1) \wedge l = 0 \rangle \\
0 &\vdash \langle L(r, x, 1) \rangle \text{unlock}(x) \langle L(r, x, 0) \rangle
\end{aligned}$$

where $L(r, x, l)$ abstractly represents the lock resource at abstract location r and concrete address x , with abstract state $l \in \{0, 1\}$.

— *Notation.* Given $n \in X$ and $ns, ns' \in X^*$, we write $n \oplus ns$, $ns \oplus n$, and $ns \oplus ns'$ for prepend, append, and concatenation, respectively; $|ns|$ is the length of ns , and $ns(i) = n$ states that the i -th element (from 0) in ns is n and $i < |ns|$; $\text{fst}(ns)$ and $\text{last}(ns)$ are the first and the last element of ns , respectively.

— *Region Types.* The abstract shared lock resource will be represented by a region $\text{clh}_r(x, h, l, o)$ where $x, h \in \text{Addr}$, $l \in \{0, 1\}$, $o \in \mathbb{N}$. Here x , the address of the lock, is the fixed parameter of the

region: The abstract state of the region includes l , which represents its state, o , which is the *ticket number*, explained later, of the lock's current owner, and h is the address of the cell associated with the owner. The lock resource is abstractly represented by the predicate:

$$L(r, x, l) \triangleq \exists o \in \mathbb{N}. \exists h \in \text{Addr}. \text{clh}_r(x, h, l, o) * [\mathbf{E}]_r$$

– *Guard algebra*: Take $p, c \in \text{Addr}, ns \in \text{Addr}^*, o, t \in \mathbb{N}$ arbitrary. For this proof, two guards will be necessary. First $\mathbf{T}(p, c, t)$, which represents the ownership of a position in the queue. The parameters $c, p \in \text{Addr}$ are pointers to the cell associated with the current thread and its predecessor respectively. Here, $t \in \mathbb{N}$ is the ticket number associated with the thread owning the $\mathbf{T}(p, c, t)$ guard. The second guard we need is $\mathbf{Q}(ns, o)$, which is used to track the overall queue, by tracking the cells associated with enqueued threads, $ns \in \text{Addr}^*$, and the ticket number of the current owner, $o \in \mathbb{N}$.

To use this as intended, a few axioms on the guard algebra will be required. First, an axiom to create new tickets, adding a new cell to the queue and associated a new, unique ticket number to the thread:

$$\mathbf{Q}(ns \oplus [p], o) = \mathbf{Q}(ns \oplus [p, c], o) \bullet \mathbf{T}(p, c, o + |ns| + 1)$$

This will be used to create the relevant guard resources \mathbf{T} , when a lock operation enqueues itself on line 4. Similarly, an axiom to remove a ticket from the queue once it can take possession of the lock:

$$\mathbf{Q}([p, c] \oplus ns, o) \bullet \mathbf{T}(p, c, o + 1) = \mathbf{Q}(c \oplus ns, o + 1)$$

This will be used to update the relevant guard resources \mathbf{Q} with the relevant \mathbf{T} , when a lock operation takes possession of the lock on line 7. Finally, an axiom to guarantee that a ticket guard, \mathbf{T} is well-formed with respect to the queue in a guard \mathbf{Q} :

$$\mathbf{Q}(ns, o) \bullet \mathbf{T}(p, c, t) \neq \perp \Leftrightarrow ns(t - o - 1) = p \wedge ns(t - o) = c$$

– *Obligation algebra*: Take $o, o', t, t' \in \mathbb{N}$ arbitrary. To verify the totality of the CLH lock operation, once a thread is enqueued, if its predecessor gains and then relinquishes possession of the lock, it must eventually take possession of it (and in fact unlock it, as indicated by the lock specification's pseudoquantifier) or subsequent enqueued threads will not be able to guarantee they terminate, as the lock operation's while loop will never terminate and so they will never get the opportunity to take possession of the lock.

To do this, we associate an obligation $\mathbf{P}(t)$ with the ownership of the ticket $t \in \mathbb{N}$. The region's transition system then requires that a thread with ticket $o \in \mathbb{N}$ must return the obligation $\mathbf{P}(o)$ to the region when taking possession of the lock. The layer associated with $\mathbf{P}(t)$ is then t , so that these obligations are resolved in the order the associated threads are enqueued. Finally, as with the guard algebra, we have an obligation $\mathbf{O}(o, t)$, which will remain in the shared region's state and track the owner's ticket, o , and the next ticket to be handed out, t , associated with the obligation \mathbf{P} via the obvious axioms.

$$\mathbf{O}(o, t) = \mathbf{O}(o, t + 1) \bullet \mathbf{P}(t) \quad \mathbf{O}(o + 1, t) = \mathbf{O}(o, t) \bullet \mathbf{P}(o + 1)$$

$$\mathbf{O}(o, t) \bullet \mathbf{P}(t') \neq \perp \Leftrightarrow o \leq t' < t$$

$$\mathcal{L} \triangleq \mathbb{N} \cup \{\mathbf{1}, \mathbf{0}\} \quad \forall i \in \mathbb{N}. \mathbf{1} > i > \mathbf{0} \quad \text{lay}(\mathbf{O}(o, o')) = 0 \quad \text{lay}(\mathbf{P}(t)) = t$$

– *Interference protocol*. The interference protocol for the CLH lock is as follows:

$$\mathbf{E} : ((h, l, o), \mathbf{0}) \rightsquigarrow ((h, l, o), \mathbf{P}(t))$$

$$\mathbf{E} : ((h, 0, o), \mathbf{P}(o + 1)) \rightsquigarrow ((h', 1, o + 1), \mathbf{0})$$

$$\mathbf{E} : ((h, 1, o), \mathbf{0}) \rightsquigarrow ((h, 0, o), \mathbf{0})$$

The first transition allows a thread to place itself in the queue waiting to obtain the CLH lock. While doing so, the threads acquires an obligation, $\mathbf{p}(t)$, requiring it to eventually take possession of the lock once it is at the head of the queue. The second, allows the thread at the head of the queue to take possession of the lock, by changing the state, l , incrementing the owner ticket, o , to its own (tracked by the thread’s obligation) and changing the owner pointer of the lock. This discharges the obligation $\mathbf{p}(o + 1)$, as the thread then leaves the queue, to take possession of the lock. Finally, the third transition allows the lock to be unlocked.

– *Region interpretation.* As explained above, the thread queue is represented by an abstract queue of the addresses associated with each thread, ns . While threads are queuing, the associated cells must have value 1; this is represented using the predicate ones:

$$\text{ones}(ns) \triangleq ns(1) \mapsto 1 * \dots * ns(|ns| - 1) \mapsto 1$$

The implementation holds the cells associated with each queued thread, this is represented by the resource $\text{ones}(ns)$ in the region interpretation.

The implementation also holds a pointer to the tail of the queue, ns , as well as a pointer to the cell associated with the thread possessing the lock, whose state is the the state of the lock, as described above. This is described by the resource:

$$x \mapsto h, \text{last}(ns) * h \mapsto l$$

This is then tied together with the ghost state, using t , the next available ticket number and environmental obligations are kept for each obligation \mathbf{p} held by the environment, from $\mathbf{p}(o + 1)$, the next ticket to possess the lock, to $\mathbf{p}(t - 1)$, the last ticket that has been handed to an enqueued thread:

$$[\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E$$

Finally, the invariant $t - o = |ns|$ is used to guarantee that each thread that holds a ticket is associated with a cell in the queue ns . All of this ties together to give the following region interpretation:

$$\mathcal{I}(\text{clh}_r(x, h, l, o)) \triangleq \exists ns \in \text{Addr}^*, t \in \mathbb{N}. x \mapsto h, \text{last}(ns) *$$

$$h \mapsto l * \text{ones}(ns) * [\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E \wedge t - o = |ns|$$

– *Proof of lock.* Figure 11 gives a high-level outline of the clh lock operation implementation, the definition of the loop invariant $P(\beta)$ will be given later. The steps involving liveness are the **FAS** operation, the while loop and setting the owner of the lock, at line 7. First the details of the **FAS** operation’s proof are as follows:

$$\begin{array}{l} 1; [r \mapsto (X_1, 0, X_2, R)] \vdash \\ \left\{ \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, l, o) * r \Rightarrow \blacklozenge * c \mapsto 1 \right\} \\ \left| \begin{array}{l} 1; [r \mapsto (X_1, 0, X_2, R)] \vdash \\ \mathbb{W}l \in \{0, 1\}, o, t \in \mathbb{N}, h \in \text{Addr}, ns \in \text{Addr}^*. \\ \left\langle x \mapsto h, \text{last}(ns) * h \mapsto l * \text{ones}(ns) * [\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E \wedge t - o = |ns| * c \mapsto 1 \right\rangle \\ \mathbf{0}; [r \mapsto (X_1, 0, X_2, R)] \vdash \\ \left\langle x + 1 \mapsto \text{last}(ns) \right\rangle \\ \mathbf{p} := \mathbf{FAS}(x + 1, c); \\ \left\langle x + 1 \mapsto c \wedge \mathbf{p} = \text{last}(ns) \right\rangle \\ \left\langle \exists ns' \in \text{Addr}^*. x \mapsto h, \text{last}(ns') * h \mapsto l * \text{ones}(ns') * [\mathbf{Q}(ns', o)]_r * [\mathbf{O}(o, t + 1)]_r^L * \right. \\ \left. \bigstar_{i=o+1}^t [\mathbf{P}(i)]_r^E \wedge (t + 1) - o = |ns'| \wedge o < t \wedge ns' = ns \oplus c * ([\mathbf{T}(\mathbf{p}, c, t)]_r * [\mathbf{P}(t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E) \right\rangle \\ \left\{ \exists l \in \{0, 1\}, o, t \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, l, o) * r \Rightarrow \blacklozenge * [\mathbf{T}(\mathbf{p}, c, t)]_r * [\mathbf{P}(t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E \wedge o < t \right\} \end{array} \right. \end{array}$$

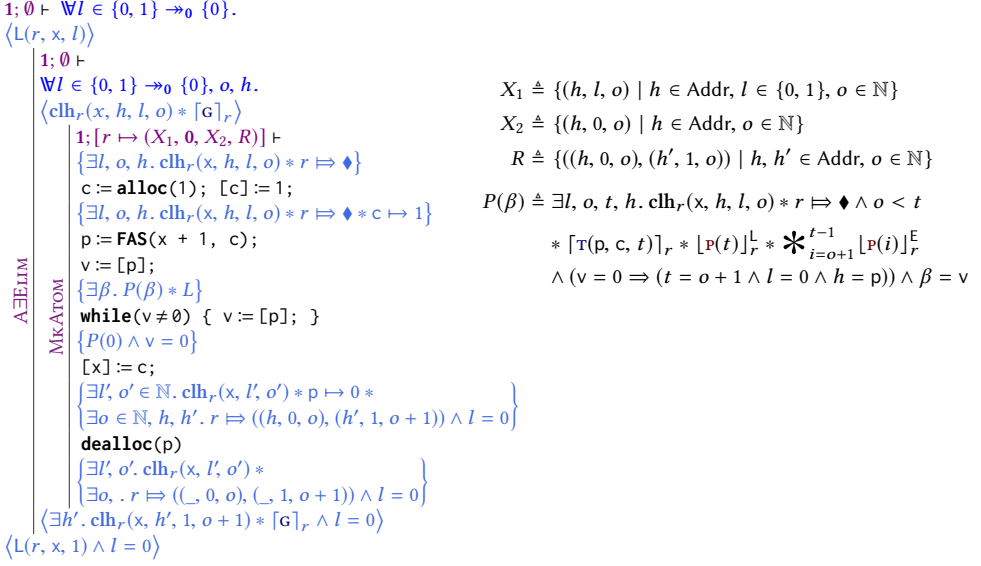


Fig. 11. Outline of CLH lock proof.

where STEP 5 is composed of the rules: **FRAME**, **ATOMW**, **A \exists ELIM**, **LIFTA**, **A \exists ELIM**. The application of the **FRAME** rule frames of the view $r \Rightarrow \blacklozenge$, the **ATOMW** rule then transfers all the remaining resources to the atomic precondition and postcondition, the **A \exists ELIM** rule then pseudoquantifies l, o and h , **LIFTA** then opens up the region and the final application of **A \exists ELIM** rule pseudoquantifies ns .

The consequence rule is implicitly applied to the postcondition of the antecedent of the inner rule application, using the relevant guard and obligation axioms to reestablish the region interpretation, consequently retaining the guard assertion $\lceil \text{T}(p, c, t) \rceil_r$ and the obligation assertion $\lfloor \text{P}(t) \rfloor_r^L$ outside of the region. A stable copy of the environment assertions within the region are also retained locally. These environmental assertions will be necessary for the application of the **WHILE** rule.

Next, consider the proof of the **while** loop. The loop invariant is:

$$\begin{aligned}
P(\beta) \triangleq \exists l, o, h. \text{clh}_r(x, h, l, o) * \lceil \text{T}(p, c, t) \rceil_r \wedge o < t \\
\wedge (v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)) \wedge \beta = v
\end{aligned}$$

which asserts that:

- $\lceil \text{T}(p, c, t) \rceil_r$, the local thread is queueing for the lock, with predecessor cell p , current cell c , and ticket t .
- $o < t$, the current owner must come before the local thread with ticket t . This is stable due to the **T** guard.
- $v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)$, if v , the last read of the value of the predecessor cell is 0, then the owner is the predecessor of the current thread, therefore $t = o + 1$, and the lock is unlocked, $l = 0$. The owner's cell, h , will also take the value of that of the predecessor.

- β is 0 once the thread has observed that its predecessor has taken possession of and relinquished the lock (by reading the cell at address p). β will have value 1 otherwise.

A thread with ticket t can take possession of a CLH lock once its predecessor has taken possession of and relinquished the lock. Once the lock reaches this state, $o = t - 1$ and $l = 0$, it holds stably as all transitions from this state would set the owner of the lock to t , which can only be done by a thread holding the appropriate \mathbf{T} guard.

The intent of this loop is to wait till this condition holds, allowing the thread to safely take possession of the lock once the loop terminates. Hence, the goal state, is:

$$T = \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, l, o) \wedge t = o + 1 \wedge l = 0 \wedge h = p$$

Once the lock reaches this state, a subsequent iteration of this **while** loop will terminate with $v = 0$, breaking the loop. To reach the goal state, threads that come before the current thread must both take possession and relinquish the lock. The first is guaranteed due to obligations $\mathbf{P}(t')$ for $t' < t$ and the second due to the pseudoquantifier, guaranteeing that the lock must always eventually be released. The progress measure

$$M(\alpha) = \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, l, o) \wedge \alpha = 2(t - o - 1) + l$$

is decreased by both of these actions, and as $t > o$, $2(t - o - 1) + l \geq 0$, the progress measure is well-founded. To support this argument, the persistent loop invariant, L , must contain the resource $r \Rightarrow \blacklozenge$ to make use of the liveness assumptions of the pseudoquantifier, and the relevant environmental liveness assertions for threads queued before the current thread:

$$L = \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, l, o) * [\mathbf{P}(t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E * r \Rightarrow \blacklozenge \wedge o < t$$

The **WHILE** rule is applied as follows:

$$\begin{array}{c} \mathbf{1}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\ \{ \exists t \in \mathbb{N}. \exists \beta_0. P(\beta_0) * L \} \\ \left. \begin{array}{l} \forall \beta_0, t \in \mathbb{N}. \\ \mathbf{1}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\ \{ P(\beta_0) * L \} \\ \text{CONS; } \exists\text{ELIM} \quad \text{WHILE} \quad \begin{array}{l} \mathbf{while}(v \neq 0) \{ \\ \quad \forall \beta \leq \beta_0, b \in \mathbb{B}. \\ \quad \{ P(\beta) * b \Rightarrow T(\beta) \wedge v \neq 0 \} \\ \quad v := [p]; \\ \quad \{ \exists \gamma. P(\gamma) \wedge \gamma \leq \beta \wedge b \Rightarrow \gamma < \beta \} \\ \quad \} \\ \quad \{ \exists \beta. P(\beta) * L \wedge \beta \leq \beta_0 \wedge v = 0 \} \end{array} \\ \{ \exists o \in \mathbb{N}. \text{clh}_r(x, p, o, o) * r \Rightarrow \blacklozenge * [\mathbf{T}(p, c, o + 1)]_r * [\mathbf{P}(o + 1)]_r^L \} \end{array} \right\} \end{array}$$

The rule $\exists\text{ELIM}$ is applied to quantify t and β_0 over the antecedent.

To complete the application of the rule we need to show

$$\forall \alpha. \mathcal{A} \vDash \exists \alpha'. L * M(\alpha') \wedge \alpha' \leq \alpha \text{ stable} \quad (10)$$

$$\mathbf{1}; \mathcal{A} \vdash L \xrightarrow{M} T(\beta) \quad (11)$$

Condition (10) holds trivially. Suppose, for an arbitrary α , that $\exists \alpha_0. L * M(\alpha_0) \wedge \alpha_0 \leq \alpha$ holds for some world w and consider a world w' such that $w \mathbf{R}_{\mathcal{A}} w'$; certainly w' would satisfy $\exists \alpha_1. L * M(\alpha_1)$ and such α_1 would be such that $\alpha_1 \leq \alpha_0 \leq \alpha$.

To prove (11), take

$$\begin{aligned}
L'(\alpha) &= \left(\begin{array}{c} \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr. } \text{clh}_r(x, h, l, o) * \lfloor \mathbf{P}(t) \rfloor_r^L * \\ \star_{i=o+1}^{t-1} \lfloor \mathbf{P}(i) \rfloor_r^E * r \Rightarrow \blacklozenge \wedge ((l = 0 \wedge o + 1 < t) \vee (l = 1)) \end{array} \right) * M(\alpha) \\
L'_0(\alpha) &= \left(\begin{array}{c} \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr. } \text{clh}_r(x, h, l, o) * \lfloor \mathbf{P}(t) \rfloor_r^L * \\ \star_{i=o+2}^{t-1} \lfloor \mathbf{P}(i) \rfloor_r^E * r \Rightarrow \blacklozenge \wedge l = 0 \wedge o + 1 < t \end{array} \right) * M(\alpha) \\
L'_1(\alpha) &= \left(\begin{array}{c} \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr. } \text{clh}_r(x, h, l, o) * \lfloor \mathbf{P}(t) \rfloor_r^L * \\ \star_{i=o+1}^{t-1} \lfloor \mathbf{P}(i) \rfloor_r^E * r \Rightarrow \blacklozenge \wedge l = 1 \end{array} \right) * M(\alpha) \\
L(\alpha) &= L * M(\alpha)
\end{aligned}$$

First split on $\alpha = 0 \vee \alpha > 0$:

$$\frac{\frac{\frac{}{\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L(0) \xrightarrow{M} T} \text{LIVE T}}{\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L(\alpha) \xrightarrow{M} T} \text{ECASE}}{\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L(\alpha) \xrightarrow{M} T} \text{ENVLIVE}}$$

In the case $\alpha = 0$, the rule **LIVE T** applies directly. To show $\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L'(\alpha) \wedge \alpha > 0 \xrightarrow{M} T$ holds, split on the state of the lock, $l = 0 \vee l = 1$. In either case, the set of permitted transitions on the state of the clh lock region is:

$$R_{\text{clh}} = \{((h, 0, o), (h', 1, o + 1)) \mid h, h' \in \text{Addr}, o \in \mathbb{N}\} \cup \{((h, 1, o), (h, 0, o)) \mid h \in \text{Addr}, o \in \mathbb{N}\}$$

In the case $l = 0$, the environment is guaranteed to eventually take possession of the lock due to the environmental obligation assertion $\lfloor \mathbf{P}(o + 1) \rfloor_r^E$, so the **LIVE O** rule is applied:

$$\frac{\text{impr}_{\mathcal{A}}(\text{clh}_r, \exists o \in \mathbb{N}. L'_0(\alpha, o), L, R_{\text{clh}}, R_p, T)}{\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L'_0(\alpha) * \exists o. \text{clh}_r(x, _, _, o) * \lfloor \mathbf{P}(o + 1) \rfloor_r^E \xrightarrow{M} T} \text{LIVE O}$$

where

$$R_p = \{((h, 0, o), (h', 1, o + 1)) \mid h \in \text{Addr}, o \in \mathbb{N}\}$$

is the set of transitions that return an obligation of the form $\mathbf{P}(o + 1)$ to the shared region. As none of the transitions in R_{clh} can increase the progress measure, and all of the transitions in R_p clearly decrease it, $\text{impr}_{\mathcal{A}}(\text{clh}_r, \exists o \in \mathbb{N}. L'_0(\alpha, o), L, R_{\text{clh}}, R_p, T)$ holds, as required.

In the case $l = 1$, progress is guaranteed due to the assumptions in the atomicity context, \mathcal{A} , that eventually, the lock must be released, so the **LIVE A** rule is applied:

$$\frac{\text{impr}_{\mathcal{A}}(\text{clh}_r, L'_1, L, R_{\text{clh}}, R_{pq}, T)}{\mathbf{1}; \mathcal{A} \vdash L(\alpha) : L'_1(\alpha) \xrightarrow{M} T} \text{LIVE A}$$

where

$$R_{pq} = \{((h, 1, o), (h, 0, o)) \mid h \in \text{Addr}, o \in \mathbb{N}\}$$

As before, since none of the transitions in R_{clh} can increase the progress measure, and all of the transitions in R_{pq} clearly decrease it, $\text{impr}_{\mathcal{A}}(\text{clh}_r, L'_1, L, R_{\text{clh}}, R_{pq}, T)$ holds, as required.

The argument for the body of the **while** loop's proof is purely a safety argument, the full proof is as follows:

$\forall \beta_0, t \in \mathbb{N}, \beta, b \in \mathbb{B}$.

$$\begin{array}{c}
 \mathbf{1}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\
 \left\{ \begin{array}{l}
 (\exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr. } \text{clh}_r(x, h, l, o) * [\mathbf{T}(p, c, t)]_r \wedge \\
 o < t \wedge (v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)) \wedge \beta = v \wedge b \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p) \wedge (v \neq 0))
 \end{array} \right\} \\
 \left| \begin{array}{l}
 \mathbf{1}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\
 \forall l \in \{0, 1\}, h \in \text{Addr}, ns \in \text{Addr}^*, o, nt \in \mathbb{N}. \\
 \left\langle \begin{array}{l}
 x \mapsto h, \text{last}(ns) * h \mapsto l * \text{ones}(ns) * [\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, nt)]_r^L * \bigstar_{i=o+1}^{nt-1} [\mathbf{P}(i)]_r^E \wedge nt - o = |ns| * \\
 ([\mathbf{T}(p, c, t)]_r \wedge o < t \wedge \beta \geq 1 \wedge b \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p) \wedge p \in ns)
 \end{array} \right\rangle \\
 \mathbf{0}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\
 \forall v \in \{0, 1\}. \\
 \langle p \mapsto v \rangle \\
 v := [p]; \\
 \langle p \mapsto v \wedge v = v \rangle \\
 \left\langle \begin{array}{l}
 x \mapsto h, \text{last}(ns) * h \mapsto l * \text{ones}(ns) * [\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, nt)]_r^L * \bigstar_{i=o+1}^{nt-1} [\mathbf{P}(i)]_r^E \wedge nt - o = |ns| * \\
 ([\mathbf{T}(p, c, t)]_r \wedge \beta = 1 \wedge \exists v \in \{0, 1\}. v = v \wedge b \Rightarrow v = 0 \wedge (v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)))
 \end{array} \right\rangle \\
 \left\{ \begin{array}{l}
 \exists l \in \{0, 1\}, o \in \mathbb{N}, h \in \text{Addr}, \gamma. \text{clh}_r(x, h, l, o) * [\mathbf{T}(p, c, t)]_r \wedge \\
 o < t \wedge (v = 0 \Rightarrow (t = o + 1 \wedge l = 0 \wedge h = p)) \wedge \gamma = v \wedge \gamma \leq \beta \wedge b \Rightarrow \gamma = 0
 \end{array} \right\}
 \end{array}
 \right.
 \end{array}$$

Finally, we consider the details of the linearization point, when the lock operation takes possession of the lock.

$$\begin{array}{c}
 \mathbf{1}; [r \mapsto (X_1, \mathbf{0}, X_2, R)] \vdash \\
 \left\{ \exists o \in \mathbb{N}. \text{clh}_r(x, p, o, 0) * r \Rightarrow \blacklozenge * [\mathbf{T}(p, c, o + 1)]_r * [\mathbf{P}(o + 1)]_r^L \right\} \\
 \left| \begin{array}{l}
 \mathbf{1}; \emptyset \vdash \\
 \forall t \in \mathbb{N}, ns \in \text{Addr}^*. \\
 \left\langle \begin{array}{l}
 x \mapsto p, \text{last}(ns) * p \mapsto 0 * \text{ones}(ns) * [\mathbf{Q}(ns, o)]_r * [\mathbf{O}(o, t)]_r^L * \bigstar_{i=o+1}^{t-1} [\mathbf{P}(i)]_r^E \wedge t - o = |ns| * \\
 ([\mathbf{T}(p, c, o + 1)]_r * [\mathbf{P}(o + 1)]_r^L \wedge ns(1) = c)
 \end{array} \right\rangle \\
 \mathbf{0}; \emptyset \vdash \\
 \langle x \mapsto p \rangle \\
 [x] := c; \\
 \langle x \mapsto c \rangle \\
 \left\langle \begin{array}{l}
 \exists ns' \in \text{Addr}^*. x \mapsto c, \text{last}(ns') * c \mapsto 1 * \text{ones}(ns') * [\mathbf{Q}(ns', o + 1)]_r * [\mathbf{O}(o + 1, t)]_r^L * \\
 \bigstar_{i=o+2}^{t-1} [\mathbf{P}(i)]_r^E \wedge t - (o + 1) = |ns'| * (p \mapsto 0 * ns = p \oplus ns')
 \end{array} \right\rangle \\
 \left\{ \exists l' \in \{0, 1\}, o' \in \mathbb{N}. \text{clh}_r(x, l', o') * p \mapsto 0 * \exists o \in \mathbb{N}, h, h' \in \text{Addr}. r \Rightarrow ((h, 0, o), (h', 1, o + 1)) \wedge l = 0 \right\}
 \end{array}
 \right.
 \end{array}$$

Where **STEP 6** is $\exists\text{ELIM}$, **ATOMW**, **UPDREG**, **A \exists ELIM**, **CONS**. First $\exists\text{ELIM}$ is applied to quantify the ticket of the current owner, o , (the predecessor of the current thread) over the antecedent. Then the **ATOMW** and **UPDREG** rules are applied to atomically update the region state by acting on its invariant. The **A \exists ELIM** is then applied to pseudoquantify t and ns , the two variables that are existentially quantified within the region invariant and finally the **CONS** rule is applied to re-establish the invariant in the postcondition by adjusting the ghost state. Specifically, the guard **T** and the obligation **P** are reabsorbed into **Q** and **O** respectively, to update the list of threads waiting on the lock and increment the owner. The inner part of the proof then decreases the layer and frames of unnecessary resources to apply the update.

This ends the proof of the lock operation.

– *Proof of unlock.* Let

$$X = \{(h, 1, o) \mid h \in \text{Addr}, o \in \mathbb{N}\} \quad R = \{((h, 1, o), (h, 0, o)) \mid h \in \text{Addr}, o \in \mathbb{N}\}$$

The proof of the unlock operation is as follows:

$$\begin{array}{l}
 \mathbf{0}; \emptyset \vdash \\
 \langle L(r, x, 1) \rangle \\
 \left. \begin{array}{l}
 \mathbf{0}; [r \mapsto (X, \mathbf{0}, X, R)] \vdash \\
 \{ \exists o \in \mathbb{N}, h \in \text{Addr}. \text{clh}_r(x, h, 1, o) * r \Rightarrow \blacklozenge \} \\
 h := [x]; \\
 \{ \exists o \in \mathbb{N}. \text{clh}_r(x, h, 1, o) * r \Rightarrow \blacklozenge \} \\
 [h] := \emptyset; \\
 \{ \exists o \in \mathbb{N}. r \Rightarrow ((h, 1, o), (h, 0, o)) \}
 \end{array} \right\} \text{STEP 7} \\
 \langle L(r, x, 0) \rangle
 \end{array}$$

C.3 Distinguishing Client

Here we expand on the proof of the distinguishing client presented in Section 3, particularly focusing on the details of the application of **LIVEC** in the left-hand thread and the application of **WHILE** in the right-hand thread.

– *Code.*

$$\begin{array}{l} \text{lock}(x); \\ \text{[done]} := \text{true}; \\ \text{unlock}(x); \end{array} \quad \parallel \quad \begin{array}{l} \mathbf{var} \ d = \text{false} \ \mathbf{in} \\ \mathbf{while}(\neg d)\{ \\ \quad \text{lock}(x); \ d := \text{[done]}; \ \text{unlock}(x); \\ \} \end{array}$$

– *Specifications.* As above, we assume the following lock specifications for starvation-free locks:

$$\begin{array}{l} \mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \rightarrow \mathbf{0} \cdot \langle L(s, x, l) \rangle \text{lock}(x) \langle L(s, x, 1) \wedge l = 0 \rangle \\ \mathbf{0} \vdash \langle L(s, x, 1) \rangle \text{unlock}(x) \langle L(s, x, 0) \rangle \end{array}$$

– *Region Types.* The two threads share the variables x and done ; we will encode their state as the region $c_r(x, \text{done}, l, d)$ for $l \in \{0, 1\}$ (the state of the lock at x) and $d \in \text{Bool}$ (the value stored at done).

– *Guards and Obligations.* We use obligations \mathbf{K} and \mathbf{D} which double as guards, using the same composition operator defined by

$$\mathbf{K} \bullet \mathbf{K} = \perp \qquad \mathbf{D} \bullet \mathbf{D} = \perp \qquad \text{lay}(\mathbf{K}) = \mathbf{0} < \mathbf{1} = \text{lay}(\mathbf{D})$$

– *Interference protocol.*

$$\begin{array}{l} \mathbf{D} : ((l, \text{false}), \mathbf{D}) \rightsquigarrow ((l, \text{true}), \mathbf{0}) \\ \mathbf{0} : ((0, d), \mathbf{0}) \rightsquigarrow ((1, d), \mathbf{K}) \\ \mathbf{K} : ((1, d), \mathbf{K}) \rightsquigarrow ((0, d), \mathbf{0}) \end{array}$$

– *Region Interpretation.*

$$\begin{aligned} \mathcal{I}(c_r(s, x, \text{done}, l, d)) &\triangleq L(s, x, l) * \text{done} \mapsto d \\ &* ((([\mathbf{K}]_r^L \wedge l = 0) \vee ([\mathbf{K}]_r^E \wedge l = 1)) \\ &* ((([\mathbf{D}]_r^L \wedge d) \vee ([\mathbf{D}]_r^E \wedge \neg d)) \end{aligned}$$

– *Proof of Distinguishing Client.* The proof outline is reproduced in Fig. 12. **STEP 1** follows a standard TaDA proof pattern:

$$\frac{\frac{\frac{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \cdot \langle \exists d. c_r(s, x, \text{done}, l, d) * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle \exists d. c_r(s, x, \text{done}, 1, d) * [\mathbf{K}]_r^L \rangle}{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \cdot \langle c_r(s, x, \text{done}, l, \text{false}) * [\mathbf{D}]_r^L * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle c_r(s, x, \text{done}, 1, \text{false}) * [\mathbf{D}]_r^L * [\mathbf{K}]_r^L \rangle}}{\mathbf{1} \vdash \langle \exists l. c_r(s, x, \text{done}, l, \text{false}) * [\mathbf{D}]_r^L * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle c_r(s, x, \text{done}, 1, \text{false}) * [\mathbf{D}]_r^L * [\mathbf{K}]_r^L \rangle}}{\mathbf{1} \vdash \langle \exists l. c_r(s, x, \text{done}, l, \text{false}) * [\mathbf{D}]_r^L * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle c_r(s, x, \text{done}, 1, \text{false}) * [\mathbf{D}]_r^L * [\mathbf{K}]_r^L \rangle}} \text{FRAMEA} \\ \text{A}\exists\text{ELIM} \\ \text{ATOMW}$$

The derivation for **STEP 9** is as follows:

$$\frac{\frac{\frac{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \rightarrow \{0\} \cdot \langle L(s, x, l) \rangle \text{lock}(x) \langle L(s, x, 1) \wedge l = 0 \rangle}{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \rightarrow \{0\}, d \in \text{Bool} \cdot \langle \mathcal{I}(c_r(s, x, \text{done}, l, d)) * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle \mathcal{I}(c_r(s, x, \text{done}, 1, d)) * [\mathbf{K}]_r^L \rangle}}{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \rightarrow \{0\}, d \in \text{Bool} \cdot \langle c_r(s, x, \text{done}, l, d) * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle c_r(s, x, \text{done}, 1, d) * [\mathbf{K}]_r^L \rangle}}{\mathbf{1} \vdash \mathbf{WL} \in \{0, 1\} \rightarrow \{0\} \cdot \langle \exists d. c_r(s, x, \text{done}, l, d) * l = 1 \Rightarrow [\mathbf{K}]_r^E \rangle \text{lock}(x) \langle \exists d. c_r(s, x, \text{done}, 1, d) * [\mathbf{K}]_r^L \rangle}} \text{SUBPQ} \\ \text{FRAMEA} \\ \text{LIFTA} \\ \text{A}\exists\text{ELIM}$$

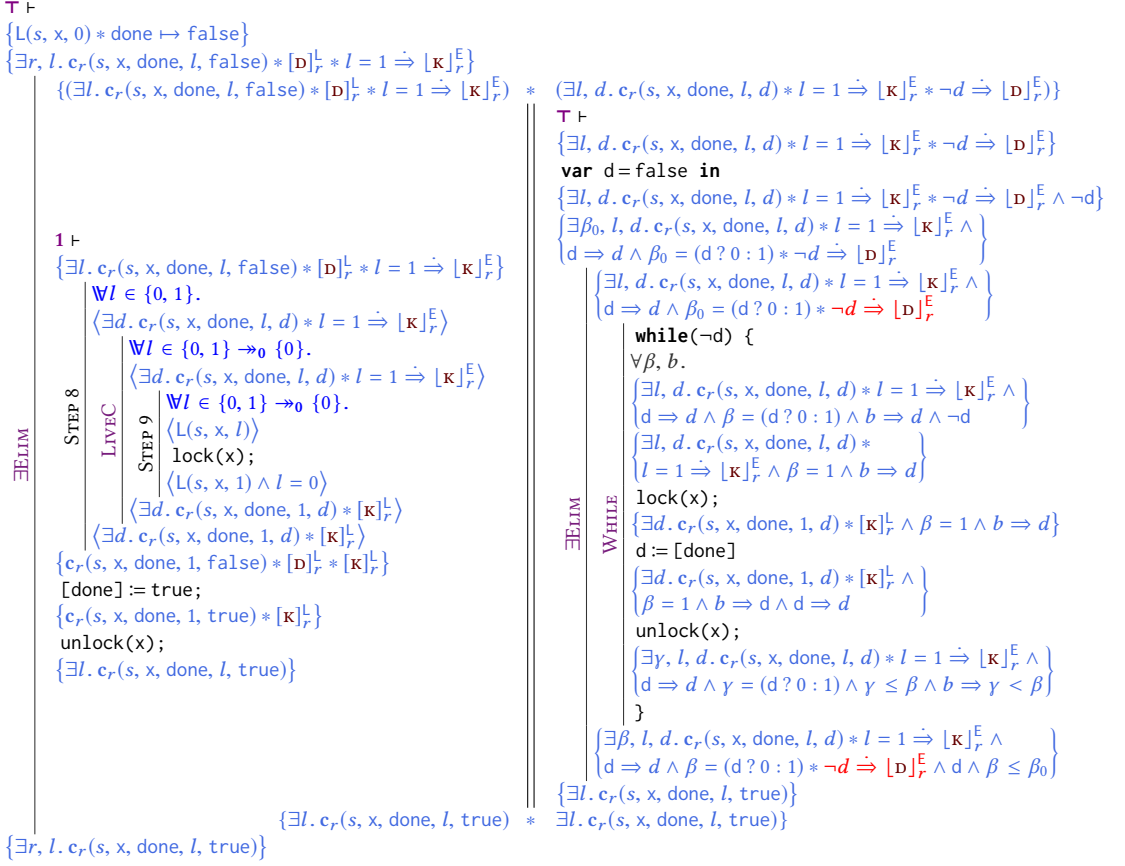


Fig. 12. Proof outline for the distinguishing client

The application of **LIVEC**. asks us to prove the environmental liveness condition $\mathsf{1}; \emptyset \vdash L \xrightarrow{M} T$ with

$$L = \exists l \in \{0, 1\}, d. c_r(s, x, \text{done}, l, d) * l = 1 \dot{\Rightarrow} [\mathsf{K}]_r^E$$

$$T = \exists l \in \{0\}, d. c_r(s, x, \text{done}, l, d) * l = 1 \dot{\Rightarrow} [\mathsf{K}]_r^E \equiv c_r(s, x, \text{done}, 0, _)$$

Here we can choose a trivial progress measure $M(\alpha) = (\alpha = 0)$ and since L trivially implies $L * \exists \alpha. M(\alpha)$, we can set $L(\alpha) = (L \wedge \alpha = 0)$ and apply **ENVLIVE**. The derivation of the environment liveness condition goes as follows:

$$\frac{\frac{\forall \alpha. \vdash_0 L_0(\alpha) \Rightarrow T}{\mathsf{1}; \emptyset \vdash L(\alpha) : L_0(\alpha) \rightarrow T} \text{LIVET} \quad \frac{\text{impr}_0(c_r, L_1, L, R_c, R_k, T)}{\mathsf{1}; \emptyset \vdash L(\alpha) : L_1(\alpha) \rightarrow T} \text{LIVEO}}{\mathsf{1}; \emptyset \vdash L(\alpha) : L_0(\alpha) \vee L_1(\alpha) \rightarrow T} \text{ECASE} \quad \text{ENVLIVE}$$

$$\mathsf{1}; \emptyset \vdash L \xrightarrow{M} T$$

To apply **ECASE** we split on the value of l : $L(\alpha) = L_0(\alpha) \vee L_1(\alpha)$ where $L_0(\alpha) = c_r(s, x, \text{done}, 0, _) \wedge \alpha = 0$ and $L_1(\alpha) = c_r(s, x, \text{done}, 1, _) * [\mathsf{K}]_r^E \wedge \alpha = 0$. If $l = 0$ we can apply **LIVET** as we are already in T ;

if $l = 1$, L_1 entails $\lfloor \mathbf{k} \rfloor_r^E$ so we can apply **LIVEO** with $G = \mathbf{0}$, $O = \mathbf{k}$, and

$$R_c \triangleq \mathcal{T}_c(_) = \bigcup_{l,d} \{((0, d), (1, d)), ((1, d), (0, d)), ((l, \text{false}), (l, \text{true}))\}$$

$$R_{\mathbf{k}} \triangleq \{((1, d), (0, d)) \mid d \in \text{Bool}\}$$

that is, all the transitions of the interference protocol of \mathbf{c} , and the only transition which fulfils \mathbf{k} , respectively. The $\text{impr}_{\mathcal{A}}$ condition is satisfied: every transition in R_c does not increase α , and any transition in $R_{\mathbf{k}}$ takes us directly to T .

The application of **WHILE** also requires an environment liveness condition check $\mathbf{1}; \mathbf{0} \vdash L' \xrightarrow{M} T'$, where:

$$T' \triangleq \mathbf{c}_r(s, x, \text{done}, _, \text{true}) \quad L' \triangleq T' \vee \mathbf{c}_r(s, x, \text{done}, _, \text{false}) * \lfloor \mathbf{D} \rfloor_r^E$$

Here M is again trivial, $M(\alpha) = (\alpha = 0)$, and $L'(\alpha) = L' * \alpha = 0$. Note how, in this case, the target states T' do not need to depend on the loop variant β , which itself is rather trivial.

Using these parameters, the following derivation tree for the environmental liveness check holds:

$$\frac{\frac{\frac{}{\mathbf{T}; \lambda; \mathcal{A} \vdash L'(\alpha) : T' \longrightarrow T'}{\mathbf{T}; \lambda; \mathcal{A} \vdash L'(\alpha) : T' \longrightarrow T'} \text{LIVET} \quad \frac{\text{impr}_{\mathcal{A}}(\mathbf{c}_r, \mathbf{c}_r(s, x, \text{done}, _, \text{false}), L', R_{i_0}, R_{\mathbf{D}}, T)}{\mathbf{T}; \lambda; \mathcal{A} \vdash L'(\alpha) : \mathbf{c}_r(s, x, \text{done}, _, \text{false}) * \lfloor \mathbf{D} \rfloor_r^E \longrightarrow T'} \text{LIVEO}}{\mathbf{T}; \lambda; \mathcal{A} \vdash L'(\alpha) : L'(\alpha) \longrightarrow T'} \text{ECASE}}{\mathbf{T}; \lambda; \mathcal{A} \vdash L' \xrightarrow{M} T'} \text{ENVLIVE}$$

Here $R_c = \mathcal{T}_c(_)$ as before and $R_{\mathbf{D}} \triangleq \{((0, d), (0, d)) \mid d \in \{\text{true}, \text{false}\}\}$ is the set of transitions fulfilling \mathbf{D} .

We split L' into two cases using **ECASE**. In the first case, T' holds, so the rule **LIVET** discharges the goal. In the second, since $\text{lay}(\mathbf{D}) = \mathbf{1}_s < \mathbf{T}$, \mathbf{D} is a live obligation. Trivially, none of the transitions in R_c increase the progress measure M . The transitions in $R_{\mathbf{D}}$ take us directly to T' thus satisfying the $\text{impr}_{\mathcal{A}}$ condition.

C.4 Blocking Counter

We briefly sketch the proof of a blocking counter module: a single natural number protected by a lock for concurrent access. The example illustrates how the TaDA Live specifications and proofs neatly support hiding blocking when it is unobservable by the client. It also shows a common proof pattern of TaDA Live: there is an inner region that exposes all the information needed for the termination argument (here both the counter value *and* the lock state) and an outer one that hides enough information to make the operation abstractly atomic: here we need to hide the state of the lock, or the region would be updated three times instead of only once. This pattern nicely separates the concerns in the proof: proving atomicity is done via the outer region, termination via the inner one.

– *Code.* The implementation of the module’s operations are:

```

1 def makeCounter(x) {           1 def incr(x) {                 1 def read(x) {
2   var x, l in                 2   l := [x];                   2   ret := [x + 1];
3   x := alloc(2);              3   lock(l);                    3 }
4   l := newLock();             4   v := [x + 1];
5   [x] := 1;                   5   [x + 1] := v + 1;
6   [x + 1] := 0;               6   unlock(l);
7   ret := x;                   7   ret := v;
8 }                               8 }

```

– *Specifications.* The abstract predicate $C(s, x, n)$ represents a blocking counter at address x with value n .

$$\begin{aligned}
& 1 \vdash \{\text{emp}\} \text{makeCounter}() \{\exists s. C(s, \text{ret}, 0)\} \\
& 1 \vdash \forall n \in \mathbb{N}. \langle \text{emp} \mid C(s, x, n) \rangle \text{incr}(x) \langle \text{ret} = n \mid C(s, x, n + 1) \rangle \\
& 1 \vdash \forall n \in \mathbb{N}. \langle \text{emp} \mid C(s, x, n) \rangle \text{read}(x) \langle \text{ret} = n \mid C(s, x, n) \rangle
\end{aligned}$$

– *Region Types.* This proof will use two region types: $\text{cnt}_r(r', x, la, n)$ and $\text{lcnt}_{r'}(x, la, l, n)$ where $r, r' \in \text{RId}$, $x, la \in \text{Addr}$, $l \in \{0, 1\}$ and $n \in \mathbb{N}$. Here r', x and la are the fixed parameters of the regions. The blocking counter resource is abstractly represented by the predicate $C((r, r', la), x, n) \triangleq \text{cnt}_r(r', x, la, n) * [\mathbf{E}]_r$.

– *Guards and Obligations.* We associate a single obligation \mathbf{k} with the region type lcnt . This obligation encodes ownership of the blocking counter’s lock, as well as the obligation to unlock it. We set $\text{lay}(\mathbf{k}) = \mathbf{0}$.

– *Interference Specification.* The guard-labeled transition system of the region cnt is:

$$\mathbf{E} : (n, \mathbf{0}) \rightsquigarrow (n + 1, \mathbf{0})$$

and the guard-labeled transition system of the region lcnt is:

$$\begin{aligned}
& \mathbf{E} : ((0, n), \mathbf{0}) \rightsquigarrow ((1, n), \mathbf{k}) \\
& \mathbf{E} : ((1, n), \mathbf{0}) \rightsquigarrow ((1, n + 1), \mathbf{0}) \\
& \mathbf{E} : ((1, n), \mathbf{k}) \rightsquigarrow ((0, n), \mathbf{0})
\end{aligned}$$

PROOF OF $\text{incr}(x)$:

$\text{CONS}; s = (r, r', la)$	MKATOM	$\mathbf{1}; \emptyset \vdash \forall n \in \mathbb{N}.$ $\langle \text{emp} \mid C(s, x, n) \rangle$ $\langle \text{emp} \mid \text{cnt}_r(r', x, la, n) * \lceil \mathbf{E} \rceil_r \rangle$ $\mathbf{1}; [r \mapsto (\mathbb{N}, \mathbf{0}, \mathbb{N}, \{(n, n+1) \mid n \in \mathbb{N}\})] \vdash$ $\{ \exists n. \text{cnt}_r(r', x, la, n) * r \Rightarrow \blacklozenge \}$ $l := [x];$ $\{ \exists n, l. \text{cnt}_r(r', x, l, n) * \text{lcnt}_{r'}(x, l, n) * r \Rightarrow \blacklozenge * l = 1 \Rightarrow \lfloor \mathbf{K} \rfloor_r^E \}$ $\text{lock}(1);$ $\{ \exists n. \text{cnt}_r(r', x, l, n) * r \Rightarrow \blacklozenge * \lfloor \mathbf{K} \rfloor_r^L \}$ $v := [x + 1];$ $\{ \exists n. \text{cnt}_r(r', x, l, n) * r \Rightarrow \blacklozenge * \lfloor \mathbf{K} \rfloor_r^L \wedge v = n \}$ $[x + 1] := v + 1;$ $\{ \exists n. \text{cnt}_r(r', x, l, n+1) * r \Rightarrow \langle n, n+1 \rangle * \lfloor \mathbf{K} \rfloor_r^L \wedge v = n \}$ $\text{unlock}(1);$ $\{ \exists n. r \Rightarrow \langle n, n+1 \rangle \wedge v = n \}$ $\text{ret} := v;$ $\{ \exists n. r \Rightarrow \langle n, n+1 \rangle \wedge \text{ret} = n \}$ $\langle \text{ret} = n \mid \text{cnt}_r(r', x, la, n+1) * \lceil \mathbf{E} \rceil_r \rangle$ $\langle \text{ret} = n \mid C(s, x, n+1) \rangle$
--------------------------------	-----------------	--

Fig. 13. Blocking counter: proof of incr .

– *Region Interpretations.* The interpretation of the locked counter region lcnt links the state of the lock and counter to the abstract state of the region and the ownership of \mathbf{k} . The region cnt is a simple wrapper around the lcnt region that hides the state of the lock.

$$\mathcal{I}(\text{lcnt}_r(x, la, l, n)) \triangleq \exists s. x \mapsto la, n * L(s, la, l) * ((l = 0 \wedge \lfloor \mathbf{K} \rfloor_r^L) \vee (l = 1 \wedge \lfloor \mathbf{K} \rfloor_r^E))$$

$$\mathcal{I}(\text{cnt}_r(r', x, la, n)) \triangleq \exists l \in \{0, 1\}. \text{lcnt}_{r'}(x, la, l, n) * \lceil \mathbf{E} \rceil_{r'}$$

– *Proof of incr .* The proof of incr can be found in Fig. 13. The only step requiring liveness reasoning is the call $\text{lock}(x)$, which is handled very similarly to the same call in the left thread of the distinguishing client (Example 3.1) where the environment liveness condition of the LIVEC rule application is discharged using $l = 1 \Rightarrow \lfloor \mathbf{K} \rfloor_r^E$.

C.5 Double Blocking Counter

Here we develop the proof of a double blocking counter module, that is a module encapsulating two integers each protected by a lock. The module offers linearizable operations to increment/read each counter in isolation and an `incrBoth` operation to atomically increment both. The implementation of `incrBoth` needs to deal with the ubiquitous pattern of locking multiple locks in a nested fashion, which is one of the most common sources of deadlocks in coarse-grained concurrent programs. The example illustrates how the specification format and layer system of TaDA Live allow for modular proofs of deadlock-freedom. In particular, verifying the example in LiLi would require: (i) replacing the calls to the lock operations with some non-atomic abstract code (ii) build a termination argument that talks about the queues of the two fair locks; in particular the variant argument would need to consider both queues at the same time and argue about all the possible ways the threads in the environment may enter and exit both queues. We avoid these complications by: (i) reusing the (fair) lock specifications which are truly atomic and properly hide the queues (ii) arguing about termination by means of two obligations with layers the order of which reflect the order of acquisition of locks. These obligations only represent the liveness invariant that each lock is always eventually released, the layers represent the dependency between the two locks. The proof requires no detail about why, thanks to the internal queues, this is sufficient to ensure global progress: that part of the argument has already been made in proving the lock specifications!

— *Code.* The implementation of the module’s operations are:

```

1 def makeDCounter() {
2   var x,l1,l2 in
3   x := alloc(4);
4   l1 := newLock();
5   l2 := newLock();
6   x.lock1 := l1;
7   x.lock2 := l2;
8   x.cnt1 := 0;
9   x.cnt2 := 0;
10  ret := x
11 }

1 def incr1(x) {
2   var l,v in
3   l := x.lock1;
4   lock(l);
5   v := x.cnt1;
6   x.cnt1 := v + 1;
7   unlock(l);
8   ret := v
9 }

1 def read1(x) {
2   var l1 in
3   l1 := x.lock1;
4   lock(l1);
5   ret := x.cnt1;
6   unlock(l1)
7 }

1 def incr2(x) {
2   var l,v in
3   l := x.lock2;
4   lock(l);
5   v := x.cnt2;
6   x.cnt2 := v + 1;
7   unlock(l);
8   ret := v
9 }

1 def read2(x) {
2   var l2 in
3   l2 := x.lock2;
4   lock(l2);
5   ret := x.cnt2;
6   unlock(l2)
7 }

1 def incrBoth(x) {
2   var l1,l2,v in
3   l1 := x.lock1;
4   l2 := x.lock2;
5   lock(l1);
6   lock(l2);
7   v := x.cnt1;
8   x.cnt1 := v + 1;
9   v := x.cnt2;
10  x.cnt2 := v + 1;
11  unlock(l2);
12  unlock(l1)
13 }

```

using the following abbreviations for readability:

$$x.\text{lock1} \triangleq [x] \quad x.\text{lock2} \triangleq [x+1] \quad x.\text{cnt1} \triangleq [x+2] \quad x.\text{cnt2} \triangleq [x+3]$$

– *Specifications.* The abstract predicate $\text{DC}(t, x, n, m)$ represents a double counter at address x with values n and m respectively. The goal of the proof is proving the specifications

$$\begin{aligned}
& 1 \vdash \{\text{emp}\} \text{makeDCounter}() \{\exists s. \text{DC}(t, \text{ret}, 0, 0)\} \\
& 1 \vdash \forall n, m \in \mathbb{N}. \langle \text{DC}(t, x, n, m) \rangle \text{incrBoth}(x) \langle \text{DC}(t, x, n+1, m+1) \rangle \\
& 1 \vdash \forall n, m \in \mathbb{N}. \langle \text{emp} \mid \text{DC}(t, x, n, m) \rangle \text{incr1}(x) \langle \text{ret} = n \mid \text{DC}(t, x, n+1, m) \rangle \\
& 1 \vdash \forall n, m \in \mathbb{N}. \langle \text{emp} \mid \text{DC}(t, x, n, m) \rangle \text{incr2}(x) \langle \text{ret} = m \mid \text{DC}(t, x, n, m+1) \rangle \\
& 1 \vdash \forall n, m \in \mathbb{N}. \langle \text{emp} \mid \text{DC}(t, x, n, m) \rangle \text{read1}(x) \langle \text{ret} = n \mid \text{DC}(t, x, n, m) \rangle \\
& 1 \vdash \forall n, m \in \mathbb{N}. \langle \text{emp} \mid \text{DC}(t, x, n, m) \rangle \text{read2}(x) \langle \text{ret} = m \mid \text{DC}(t, x, n, m) \rangle
\end{aligned}$$

given the fair lock specifications:

$$\begin{aligned}
& 1_s \vdash \forall l \in \{0, 1\} \rightarrow_{0_s} \{0\}. \langle L(s, x, l) \rangle \text{lock}(x) \langle L(s, x, 1) \wedge l = 0 \rangle \\
& 0_s \vdash \langle L(s, x, 1) \rangle \text{unlock}(x) \langle L(s, x, 0) \rangle
\end{aligned}$$

It is important to note here that we are making explicit the parametrisation of the layers in the region identifiers s , because we will need to associate different layers with the two instances of the lock. See Appendix A.4 for details on this parametrisation. As we will see later, we will have two region identifiers s_1 and s_2 , one per lock, with associated layers $1_{s_1}, 0_{s_1}, 1_{s_2}, 0_{s_2}$. The lock specifications themselves only require $1_{s_1} > 0_{s_1}$ and $1_{s_2} > 0_{s_2}$ but we will additionally impose, for this client proof, $0_{s_1} > 1_{s_2}$. This represents the fact that, in this client, the release of lock 1 will depend on the release of lock 2.

– *Region Types.* As for the single counter example, we need two nested regions, one to prove the atomicity of the operation (dcnt) and an inner one to prove termination (ldcnt). They differ in that dcnt only records the abstract states of the counters, while ldcnt includes the abstract states of the locks. Formally: $\text{dcnt}_{r_1}((r_0, t_0), x, n, m)$ and $\text{ldcnt}_{r_0}(t_0, x, l_1, l_2, n, m)$ where $r_0, r_1 \in \text{Rld}$, $x \in \text{Addr}$, $l_1, l_2 \in \{0, 1\}$ and $n, m \in \mathbb{N}$, and t_0 is a tuple (la_1, la_2, s_1, s_2) with $la_1, la_2 \in \text{Addr}$ and $s_1, s_2 \in \text{Rld}$. Here $(r_0, t_0), x$, and t_0, x are the fixed parameters of the two regions respectively. The double blocking counter resource is abstractly represented by the predicate $\text{DC}((r_1, t_1), x, n, m) \triangleq \text{dcnt}_{r_1}(t_1, x, n, m) * \lceil E \rceil_{r_1}$.

– *Guards and Obligations.* We introduce the guard constructors B_i, C_i , and w_i , for $i \in \{1, 2\}$, for bookkeeping of the value of the counters. We need this ghost state because in incrBoth there is an intermediate state where one counter has been updated but the other hasn’t; we cannot update the abstract state in two steps because we are proving atomicity of the operation, so we need to update both counter values in the abstract state in one go. We record the intermediate concrete state in these guards so the information is there locally without affecting the shared abstract state prematurely. The guard composition satisfies the axioms

$$B_1 = C_1(n, n') \bullet w_1(n, n') \qquad B_2 = C_2(n, n') \bullet w_2(n, n')$$

Here $C_i(n, n')$ is the reference value (left in the region interpretation) for the i -th counter’s abstract (n) and concrete (n') value and w_i is a local “witness” for the same information about the i -th counter, which can only be obtained when locking the i -th lock (otherwise it would not be stable information). This is enforced by the interpretation given later.

We associate two obligations K_1 and K_2 with the region type ldcnt , encoding ownership of the double counter’s locks respectively, as well as the obligation to unlock them:

$$K_1 \bullet K_1 = \perp \qquad K_2 \bullet K_2 = \perp$$

As anticipated, we choose the layers of the lock specifications in a way that represents the dependency between the two locks. We have a (double-counter-local) top (**1**) and bottom (**0**) layers, and intermediate layers for the locks:⁸

$$\mathbf{0} = \mathbf{0}_{s_2} = \text{lay}(\mathbf{k}_2) < \mathbf{1}_{s_2} < \mathbf{0}_{s_1} = \text{lay}(\mathbf{k}_1) < \mathbf{1}_{s_1} = \mathbf{1}$$

– *Interference Specification.* The interference protocol of the region **dcnt** trivially allows for any change to the counter values:

$$\mathbf{E} : ((n, m), \mathbf{0}) \rightsquigarrow ((n', m'), \mathbf{0})$$

The interference protocol of the region **ldcnt** encodes the constraint that we can update a counter only by holding the corresponding lock:

$$\begin{array}{ll} \mathbf{E} : ((0, l, n, m), \mathbf{0}) \rightsquigarrow ((1, l, n, m), \mathbf{k}_1) & \mathbf{E} : ((l, 0, n, m), \mathbf{0}) \rightsquigarrow ((l, 1, n, m), \mathbf{k}_2) \\ \mathbf{E} : ((1, l, n, m), \mathbf{k}_1) \rightsquigarrow ((0, l, n, m), \mathbf{0}) & \mathbf{E} : ((l, 1, n, m), \mathbf{k}_2) \rightsquigarrow ((l, 0, n, m), \mathbf{0}) \\ \mathbf{E} : ((1, l, n, m), \mathbf{k}_1) \rightsquigarrow ((1, l, n', m), \mathbf{k}_1) & \mathbf{E} : ((l, 1, n, m), \mathbf{k}_2) \rightsquigarrow ((l, 1, n, m'), \mathbf{k}_2) \end{array}$$

– *Region Interpretations.* The interpretation of **dcnt** formalises the fact that the outer region simply hides the state of the locks for the atomicity argument, while the actual internal protocol of the module is encoded in the interpretation of the inner region **ldcnt**:

$$\begin{aligned} \mathcal{I}(\mathbf{dcnt}_{r_1}((r_0, t_0), x, n, m)) &\triangleq \exists l_1, l_2 \in \{0, 1\}. \mathbf{ldcnt}_{r_0}(t_0, x, l_1, l_2, n, m) * \lceil \mathbf{E} \rceil_{r_0} \\ \mathcal{I}(\mathbf{ldcnt}_{r_0}((la_1, la_2, s_1, s_2), x, l_1, l_2, n, m)) &\triangleq \exists n', m' \in \mathbb{N}. \end{aligned}$$

$$\begin{aligned} x &\mapsto la_1, la_2, n', m' * \mathbf{L}(s_1, la_1, l_1) * \mathbf{L}(s_2, la_2, l_2) \\ &* \left(\begin{array}{l} (l_1 = 0 \wedge \lceil \mathbf{k}_1 \rceil_{r_0}^{\mathbf{L}} * \lceil \mathbf{B}_1 \rceil_{r_0} \wedge n = n') \\ \vee (l_1 = 1 \wedge \lceil \mathbf{k}_1 \rceil_{r_0}^{\mathbf{E}} * \lceil \mathbf{C}_1(n, n') \rceil_{r_0}) \end{array} \right) \\ &* \left(\begin{array}{l} (l_2 = 0 \wedge \lceil \mathbf{k}_2 \rceil_{r_0}^{\mathbf{L}} * \lceil \mathbf{B}_2 \rceil_{r_0} \wedge m = m') \\ \vee (l_2 = 1 \wedge \lceil \mathbf{k}_2 \rceil_{r_0}^{\mathbf{E}} * \lceil \mathbf{C}_2(m, m') \rceil_{r_0}) \end{array} \right) \end{aligned}$$

– *Proof of incrBoth.* The proof outline of **incrBoth** is reproduced in Fig. 14. Most of the proof is routine; the derivation for the acquisition of the first lock follows closely the pattern we already explained in Appendices C.3 and C.4. We show the proof of the acquisition of the second lock in more detail, to show the interplay between the layers. At that point we are continuously holding the obligation of the first lock, with layer greater than $\mathbf{1}_{s_2}$, so apply **LAYW** to lower the layer to $\mathbf{1}_{s_2}$ enabling the application of **FRAME** to frame $r_1 \Rightarrow \blacklozenge * \lceil \mathbf{k}_1 \rceil_{r_0}^{\mathbf{L}} * \lceil \mathbf{W}_1(n, n) \rceil_{r_0}$. The obligation \mathbf{k}_2 has layer lower than $\mathbf{1}_{s_2}$ so we are allowed to invoke it to discharge the environment liveness condition of the **LIVEC** application, in a way that is analogous to the derivations of Appendices C.3 and C.4.

⁸The proof works with $\mathbf{1}_{s_2} = \mathbf{0}_{s_1}$ too, but the ordered version better emphasizes the dependency between the locks.

PROOF OF $\text{incrBoth}(x)$:

$\text{CONS}; \text{Sub } t = (r_1, t_1), t_1 = (r_0, t_0), t_0 = (l_{a_1}, l_{a_2}, s_1, s_2)$

MKATOM

$$\langle \text{dcnt}_{r_1}(t_1, x, n, m) * [\mathbf{E}]_{r_1} \rangle$$

LAW; FRAME; ATOMW; AEBLIM

$$1; \mathcal{A} \triangleq [r_1 \mapsto (\mathbb{N}^2, \mathbf{0}, \mathbb{N}^2, \{(n, m), (n+1, m+1)\} \mid n, m \in \mathbb{N})] \vdash$$

LIFTA

$$\langle \exists n, m. \text{dcnt}_{r_1}(t_1, x, n, m) * r_1 \Rightarrow \blacklozenge \rangle$$

LIVEC

$$l_1 := [x];$$

STEP 10

$$l_2 := [x + 1];$$

LIFTA

$$// \ t'_1 \triangleq (r_0, t'_0), \ t'_0 \triangleq (l_1, l_2, s_1, s_2)$$

LIVEC

$$\langle \exists n, m, l_1, l_2. \text{dcnt}_{r_1}(t'_1, x, n, m) * \text{ldcnt}_{r_0}(t'_0, x, l_1, l_2, n, m) * \rangle$$

LIVEC

$$\langle r_1 \Rightarrow \blacklozenge * l_1 = 1 \Rightarrow \lfloor \mathbf{K}_1 \rfloor_{r_0}^E * l_2 = 1 \Rightarrow \lfloor \mathbf{K}_2 \rfloor_{r_0}^E \rangle$$

LIVEC

$$\text{lock}(l_1);$$

LIVEC

$$\langle \exists n, m, l_2. \text{dcnt}_{r_1}(t'_1, x, n, m) * \text{ldcnt}_{r_0}(t'_0, x, l_1, l_2, n, m) * \rangle$$

LIVEC

$$\langle r_1 \Rightarrow \blacklozenge * \lfloor \mathbf{K}_1 \rfloor_{r_0}^L * [\mathbf{w}_1(n, n)]_{r_0} * l_2 = 1 \Rightarrow \lfloor \mathbf{K}_2 \rfloor_{r_0}^E \rangle$$

LIVEC

$$1_{s_2}; \mathcal{A} \vdash$$

LIVEC

$$\mathbb{W}n, m \in \mathbb{N}, l_2 \in \{0, 1\}.$$

LIVEC

$$\langle \text{dcnt}_{r_1}(t'_1, x, n, m) * \text{ldcnt}_{r_0}(t'_0, x, 1, l_2, n, m) * l_2 = 1 \Rightarrow \lfloor \mathbf{K}_2 \rfloor_{r_0}^E \rangle$$

LIVEC

$$\langle \text{ldcnt}_{r_0}(t'_0, x, 1, l_2, n, m) * [\mathbf{E}]_{r_0} * l_2 = 1 \Rightarrow \lfloor \mathbf{K}_2 \rfloor_{r_0}^E \rangle$$

LIVEC

$$1_{s_2}; \mathcal{A} \vdash$$

LIVEC

$$\mathbb{W}n, m \in \mathbb{N}, l_2 \in \{0, 1\} \rightarrow_{0_{s_2}} \{0\}.$$

LIVEC

$$\langle \text{ldcnt}_{r_0}(t'_0, x, 1, l_2, n, m) * [\mathbf{E}]_{r_0} * l_2 = 1 \Rightarrow \lfloor \mathbf{K}_2 \rfloor_{r_0}^E \rangle$$

LIVEC

$$1_{s_2}; \mathcal{A} \vdash$$

LIVEC

$$\mathbb{W}l_2 \in \{0, 1\} \rightarrow \{0\}.$$

LIVEC

$$\langle \text{L}(s_2, l_2, l_2) \rangle$$

LIVEC

$$\text{lock}(l_2);$$

LIVEC

$$\langle \text{L}(s_2, l_2, 1) \wedge l_2 = 0 \rangle$$

LIVEC

$$\langle \text{ldcnt}_{r_0}(t'_0, x, 1, 1, n, m) * [\mathbf{E}]_{r_0} * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m)]_{r_0} \rangle$$

LIVEC

$$\langle \text{ldcnt}_{r_0}(t'_0, x, 1, 1, n, m) * [\mathbf{E}]_{r_0} * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m)]_{r_0} \rangle$$

LIVEC

$$\langle \text{dcnt}_{r_1}(t'_1, x, n, m) * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m)]_{r_0} \rangle$$

LIVEC

$$\langle \exists n, m. \text{dcnt}_{r_1}(t'_1, x, n, m) * r_1 \Rightarrow \blacklozenge * \rangle$$

LIVEC

$$\langle \lfloor \mathbf{K}_1 \rfloor_{r_0}^L * [\mathbf{w}_1(n, n)]_{r_0} * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m)]_{r_0} \rangle$$

LIVEC

$$v := x.\text{cnt}1;$$

LIVEC

$$x.\text{cnt}1 := v + 1;$$

LIVEC

$$v := x.\text{cnt}2;$$

LIVEC

$$x.\text{cnt}2 := v + 1;$$

LIVEC

$$\langle \exists n, m. \text{dcnt}_{r_1}(t'_1, x, n, m) * r_1 \Rightarrow \blacklozenge * \rangle$$

LIVEC

$$\langle \lfloor \mathbf{K}_1 \rfloor_{r_0}^L * [\mathbf{w}_1(n, n+1)]_{r_0} * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m+1)]_{r_0} \rangle$$

LIVEC

$$\text{unlock}(l_2);$$

LIVEC

$$\langle \exists n, m. \text{dcnt}_{r_1}(t'_1, x, n, m) * r_1 \Rightarrow ((n, m), (n+1, m+1)) * \rangle$$

LIVEC

$$\langle \lfloor \mathbf{K}_1 \rfloor_{r_0}^L * [\mathbf{w}_1(n+1, n+1)]_{r_0} * \lfloor \mathbf{K}_2 \rfloor_{r_0}^L * [\mathbf{w}_2(m, m+1)]_{r_0} \rangle$$

LIVEC

$$\text{unlock}(l_1);$$

LIVEC

$$\langle \exists n, m. r_1 \Rightarrow ((n, m), (n+1, m+1)) \rangle$$

LIVEC

$$\langle \text{dcnt}_{r_1}(t_1, x, n+1, m+1) * [\mathbf{E}]_{r_1} \rangle$$

LIVEC

$$\langle \text{DC}(t, x, n+1, m+1) \rangle$$

Fig. 14. Double blocking counter: proof of incrBoth .

STEP 10 is LIFTA, FRAME.

C.6 Lock-coupling set

Here we develop the proof of a lock coupling set module.

– *Code.* The implementation of the module’s operations are:

```

1  def add(x, e) {
2    var p, c, v, n,
3      nl, pl, cl in
4    p := locate(x, e);
5    c := p.next;
6    v := c.val;
7
8    if(v ≠ e) {
9      n := alloc(3);
10     nl := newLock();
11     n.lock := nl;
12     n.val := e;
13     n.next := c;
14     p.next := n;
15     unlock(nl);
16   }
17
18   pl := p.lock;
19   cl := c.lock;
20   unlock(cl);
21   unlock(pl);
22 }

1  def remove(x, e) {
2    var p, c, v,
3      n, pl, cl in
4    p := locate(x, e);
5    c := p.next;
6    v := c.val;
7
8    if(v = e) {
9      n := c.next;
10     p.next = n;
11   }
12
13   pl := p.lock;
14   cl := c.lock;
15   unlock(pl);
16   unlock(cl);
17 }

1  def locate(x, e) {
2    var p, c, v,
3      n, pl, cl in
4    p := x;
5    pl := p.lock;
6    lock(pl);
7    c := p.next;
8    cl := c.lock;
9    lock(cl);
10   v := c.value;
11   unlock(hl);
12
13   while(v < e) {
14     pl := p.lock;
15     c' := c.next;
16     cl' := c'.lock;
17     lock(cl');
18     v := c'.val;
19     unlock(pl);
20     p := c;
21     c := c';
22   }
23
24   ret := p;
25 }

```

The auxiliary operation `locate` is meant to only be used internally. The code uses a “record” syntax for readability, desugared as follows:

$$x.\text{lock} \triangleq [x] \qquad x.\text{val} \triangleq [x + 1] \qquad x.\text{next} \triangleq [x + 2]$$

– *Specifications.* The abstract predicate $\text{LCSet}(s, x, S)$ represents a lock-coupling set at address x abstractly representing the set S .

$$\begin{aligned} \top \vdash \forall S \in \wp(\mathbb{Z}). \langle \text{LCSet}(s, x, S) \wedge e \in \mathbb{N} \rangle \text{ add}(x, e) \langle \text{LCSet}(s, x, S \cup \{e\}) \rangle \\ \top \vdash \forall S \in \wp(\mathbb{Z}). \langle \text{LCSet}(s, x, S) \wedge e \in \mathbb{N} \rangle \text{ remove}(x, e) \langle \text{LCSet}(s, x, S \setminus \{e\}) \rangle \end{aligned}$$

– *Notation.*

$$\text{ord}(ls) \triangleq \begin{cases} \text{True} & \text{if } ls = [] \vee ls = [_] \\ v < v' \wedge \text{ord}((h', la', v', l') : ls') & \text{if } ls = (_, _, v, _) : (h', la', v', l') : ls' \end{cases}$$

$$\text{vals}(ls) \triangleq \begin{cases} \emptyset & \text{if } ls = [] \\ \{v\} \uplus \text{vals}(ls') & \text{if } ls = (_, _, v, _) : ls' \end{cases}$$

– *Region Types*. This proof will utilise two region types: $\text{lcset}_r(r', x, hl, S)$ and $\text{lclist}_r(x, h, ls)$ where $r' \in \text{RId}$, $x, hl \in \text{Addr}$, $S \in \wp(\mathbb{Z})$, $ls \in (\mathbb{Z} \cup \{\infty, -\infty\}) \times \{0, 1\}^*$. Here r' , x and hl are the fixed parameters of the region. The lock-coupling set resource is abstractly represented by the predicate

$$\text{LCSet}((r, r', hl), x, S) \triangleq \text{lcset}_r(r', x, hl, S) * [\mathbb{E}]_r$$

– *Guards and Obligations*. We will use an obligation $\mathbf{C}(ls, lay)$ to track ls and the maximal layer of any lock in the lock-coupling set, lay . The obligations $\mathbf{U}(h_1, la_1, v_1, lay_1)$, $\mathbf{L}(h_1, la_1, v_1, h_2, lay_1)$ and $\mathbf{K}(h_1, la_1, v_1, h_2, lay_1)$ track the state of the lock associated with the cell $h_1, h_2 \in \text{Addr}$, are the addresses of the associated cell and the next cell in the lock coupling set and $la_1 \in \text{Addr}$, $v_1 \in \mathbb{Z} \cup \{\infty, -\infty\}$ and lay are the address of the lock, the value and the layer associated with h_1 respectively. The obligation \mathbf{U} has as parameters all the fixed value associated with a cell in the lock coupling list and \mathbf{L} and \mathbf{K} also contain h_2 , the address of the next cell, which is fixed when the lock associated with h_1 is locked.

This axiom allows a lock to be locked:

$$\begin{aligned} \mathbf{C}(ls \oplus ((v, 0) : ls'), lay) \bullet \mathbf{U}(h_1, la_1, v_1, lay_1) = \\ \mathbf{C}(ls \oplus ((v, 1) : ls'), lay) \bullet \mathbf{L}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \end{aligned}$$

Once a thread holds the obligation $\mathbf{K}(h_1, la_1, v_1, h_2, lay_1)$, then, the list ls in the obligation $\mathbf{C}(ls, lay)$ must contain the element $(v_1, 1)$ and the cell h_1 ’s layer, lay_1 , must be smaller than or equal to the maximum layer of any cell lay . Furthermore, if a $\mathbf{L}(h_1, la_1, v_1, h_2, lay_1)$ obligation and a $\mathbf{K}(h'_1, la'_1, v'_1, h'_2, lay'_1)$ have the same cell address, then they must agree on all other elements which are invariant when the lock is held. This is expressed by the following axioms:

$$\mathbf{C}(ls, lay) \bullet \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \neq \perp \Leftrightarrow \exists ls', ls''. ls = ls' \oplus ((v_1, 1) : ls'') \wedge lay \geq lay_1$$

$$\mathbf{L}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{K}(h'_1, la'_1, v'_1, h'_2, lay'_1) \neq \perp \Leftrightarrow v_1 = v'_1 \Rightarrow \begin{pmatrix} h_1 = h'_1 \wedge la_1 = la'_1 \wedge \\ h_2 = h'_2 \wedge lay_1 = lay'_1 \end{pmatrix}$$

Next, we introduce the obligation $\mathbf{W}(h, la, v)$, which witness the fixed elements of a cell after a cell held by the local thread. As a node, h_1 , must be locked to remove the next node, h_2 from the list, this information is stable if the lock of h_1 is locked. The following axiom represent this fact:

$$\begin{aligned} \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{U}(h_2, la_2, v_2, lay_2) = \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{U}(h_2, la_2, v_2, lay_2) \bullet \mathbf{W}(h_2, la_2, v_2) \\ \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{L}(h_2, la_2, v_2, h_3, lay_2) = \mathbf{K}(h_1, la_1, v_1, h_2, lay_1) \bullet \mathbf{L}(h_2, la_2, v_2, h_3, lay_2) \bullet \mathbf{W}(h_2, la_2, v_2) \\ \mathbf{U}(h, l, v, _) \bullet \mathbf{W}(h', l', v') \neq \perp \Leftrightarrow (h = h' \Rightarrow (l = l' \wedge v = v')) \\ \mathbf{L}(h, l, v, _) \bullet \mathbf{W}(h', l', v') \neq \perp \Leftrightarrow (h = h' \Rightarrow (l = l' \wedge v = v')) \end{aligned}$$

While the `locate` operation performs the hand-over-hand locking to find the appropriate location in the lock-coupling set to act on. There also needs to be a “space” in the region’s layer structure to add in an obligation in the case of the add operation. To do this, we use an obligation, $\mathbf{FREE}(lay)$, which can be generated by increasing the maximal layer in $\mathbf{C}(ls, lay)$ to make space for an extra lock. This obligation represents an exclusive right to create a cell with the layer lay . This space

then needs to be moved along as the hand-over-hand locking occurs. These axioms allow this:

$$\begin{aligned}
& C((v_1, 1) : ls, lay) \bullet L(h_1, la_1, v_1, h_2, lay) \bullet K(h_1, la_1, v_1, h_2, lay) = \\
& \quad C((v_1, 1) : ls, lay + 1) \bullet L(h_1, la_1, v_1, h_2, lay + 1) \bullet K(h_1, la_1, v_1, h_2, lay + 1) \bullet \mathbf{FREE}(lay) \\
& \left(\begin{array}{c} L(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet K(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet \mathbf{FREE}(lay_1) \bullet \\ L(h_2, la_2, v_2, h_3, lay_2) \bullet K(h_2, la_2, v_2, h_3, lay_2) \end{array} \right) = \\
& \left(\begin{array}{c} L(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet K(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet \mathbf{FREE}(lay_2) \bullet \\ L(h_2, la_2, v_2, h_3, lay_2 + 1) \bullet K(h_2, la_2, v_2, h_3, lay_2 + 1) \end{array} \right) \\
& C(ls, lay) \bullet \mathbf{FREE}(lay_1) \neq \perp \Leftrightarrow lay > lay_1
\end{aligned}$$

Finally, these axioms allow a cell to be added or removed from the lock-coupling list when the appropriate locks are held using an appropriate obligation, $\mathbf{FREE}(lay)$:

$$\begin{aligned}
& \left(\begin{array}{c} C(ls \oplus ((v_1, 1) : (v_3, 1) : ls'), \overline{lay}) \bullet L(h_1, la_1, v_1, h_3, lay_2 + 1) \bullet K(h_1, la_1, v_1, h_3, lay_2 + 1) \bullet \\ L(h_3, la_3, v_3, h_4, lay_3) \bullet K(h_3, la_3, v_3, h_4, lay_3) \bullet \mathbf{FREE}(lay_2) \end{array} \right) = \\
& \left(\begin{array}{c} C(ls \oplus ((v_1, 1) : (v_2, 1) : (v_3, 1) : ls'), \overline{lay}) \bullet \\ L(h_1, la_1, v_1, h_2, lay_2 + 1) \bullet K(h_1, la_1, v_1, h_2, lay_2 + 1) \bullet \\ L(h_2, la_2, v_2, h_3, lay_2) \bullet K(h_2, la_2, v_2, h_3, lay_2) \bullet \\ L(h_3, la_3, v_3, h_4, lay_3) \bullet K(h_3, la_3, v_3, h_4, lay_3) \end{array} \right) \\
& \left(\begin{array}{c} C(ls \oplus ((v_1, 1) : (v_2, 1) : ls'), \overline{lay}) \bullet L(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet K(h_1, la_1, v_1, h_2, lay_1 + 1) \bullet \\ L(h_2, la_2, v_2, h_3, lay_2) \bullet K(h_2, la_2, v_2, h_3, lay_2) \bullet \mathbf{FREE}(lay_1) \\ C(ls \oplus ((v_1, 1) : ls'), \overline{lay}) \bullet L(h_1, la_1, v_1, h_3, lay_1 + 1) \bullet K(h_1, la_1, v_1, h_3, lay_1 + 1) \end{array} \right) =
\end{aligned}$$

The layer structure is $\mathbb{N} \cup \{1, \top, 0\}$, where $\forall n \in \mathbb{N}. 1 > \top > n > 0$. The obligation layers are:

$$\begin{aligned}
& \text{lay}(\mathbf{W}(h, l, v)) = \top \\
& \text{lay}(L(h_1, la_1, v_1, h_2, lay_1)) = lay_1 \\
& \text{lay}(\mathbf{FREE}(lay)) = lay
\end{aligned}$$

– *Interference protocol.* The guard-labeled transition system of the region $\text{lcset}_r(r', x, hl, S)$ is:

$$\begin{aligned}
\mathbf{E} & : \forall v. (S, \mathbf{0}) \rightsquigarrow (S \cup \{v\}, \mathbf{0}) \\
\mathbf{E} & : \forall v. (S, \mathbf{0}) \rightsquigarrow (S \setminus \{v\}, \mathbf{0})
\end{aligned}$$

and the guard-labeled transition system of the region $\text{lclist}_r(x, h, ls)$ is:

$$\begin{aligned}
\mathbf{E} & : \begin{array}{c} (ls \oplus ((v, 1) : (v'', 1) : ls'), \mathbf{0}) \rightsquigarrow \\ (ls \oplus ((v, 1) : (v', 1) : (v'', 1) : ls'), \mathbf{0}) \end{array} \quad v < v' < v'' \\
\mathbf{E} & : (ls \oplus ((v, 1) : (v', 1) : ls'), \mathbf{0}) \rightsquigarrow (ls \oplus ((v, 1) : ls'), \mathbf{0}) \\
\mathbf{E} & : (ls \oplus ((v, 0) : ls'), \mathbf{0}) \rightsquigarrow (ls \oplus ((v, 1) : ls'), L(h, la, v, h', lay)) \\
\mathbf{E} & : (ls \oplus ((v, 1) : ls'), L(h, la, v, h', lay)) \rightsquigarrow (ls \oplus ((v, 0) : ls'), \mathbf{0})
\end{aligned}$$

The first two interference specifications allow for elements to be added to and removed from the list when the correct locks are held and the last two allow the locks associated with the nodes of the lock-coupling set to be locked and unlocked.

– *Region interpretation.*

$$\mathcal{I}(\mathbf{lcsset}_r(r', x, hl, S)) \triangleq \exists ls. \mathbf{lclist}_{r'}((x, hl, -\infty, _) : ls) * [\mathbf{E}]_{r'} \wedge S \uplus \{\infty\} = \mathit{vals}(ls) \wedge \mathit{ord}(ls)$$

where $\forall n \in \mathbb{Z}. -\infty < n < \infty$.

$$\mathcal{I}(\mathbf{lclist}_r(x, hl, ls)) \triangleq \exists lay. \mathit{list}_r(ls, lay) * [\mathbf{C}(ls, lay)]_r$$

where

$$\begin{aligned} \mathit{list}_r([(h, la, v, l)], lay) &\triangleq \exists s, lay'. h \mapsto la, v, \mathbf{null} * \mathbf{Tlock}(s, la, l) * \\ &((l = 0 \wedge [\mathbf{U}(h, la, v, lay)]_r^L) \vee (l = 1 \wedge [\mathbf{L}(h, la, v, \mathbf{null}, lay)]_r^E)) \wedge lay > lay' \\ \mathit{list}_r((h, la, v, l) : (h', la', v', l') : ls, lay) &\triangleq \exists s, lay'. \\ &h \mapsto la, v, h' * \mathit{list}_r((h', la', v', l') : ls, lay') * \mathbf{Tlock}(s, la, l) * \\ &((l = 0 \wedge [\mathbf{U}(h, la, v, lay)]_r^L) \vee (l = 1 \wedge [\mathbf{L}(h, la, v, h', lay)]_r^E)) \wedge lay > lay' \end{aligned}$$

– *Proof of locate.* We detail the application of **LIVEC** in the proof of **locate** shown in Fig. 19. The associated environment liveness condition is proved by:

$$\frac{\frac{\frac{}{\mathbf{1}; \lambda; \mathcal{A} \vdash L(\alpha) : T \rightarrow T} \text{LIVET} \quad \frac{\mathit{impr}_{\mathcal{A}}(\mathbf{lclist}_r, L_1, L, R_{io}, R', T)}{\mathbf{1}; \lambda; \mathcal{A} \vdash L(\alpha) : L_1 \rightarrow T} \text{LIVEO}}{\mathbf{1}; \lambda; \mathcal{A} \vdash L(\alpha) : L(\alpha) \rightarrow T} \text{ECASE}}{\mathbf{1}; \lambda; \mathcal{A} \vdash L \xrightarrow{M} T} \text{ENVLIVE}$$

where $L(\alpha) \triangleq L * M(\alpha)$ and

$$\begin{aligned} M(\alpha) &\triangleq \exists ls, ls', nv, l. \mathbf{lclist}_{r'}(x, hl, ls \oplus ((nv, l) : ls')) \wedge \alpha = l \\ L &\triangleq \mathbf{lclist}_{r'}(x, hl, ls \oplus ((nv, l) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_r^L * [\mathbf{W}(c', cl', nv)]_r^L \\ &\quad * \exists lay'. l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _, lay')]_r^E \wedge \overline{lay} > lay' \\ L_1 &\triangleq \mathbf{lclist}_{r'}(x, hl, ls \oplus ((nv, 1) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_r^L * [\mathbf{W}(c', cl', nv)]_r^L \\ &\quad * \exists lay'. [\mathbf{K}(c', cl', nv, _, lay')]_r^E \wedge \overline{lay} > lay' \\ T &\triangleq \mathbf{lclist}_{r'}(x, hl, ls \oplus ((nv, 0) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_r^L * [\mathbf{W}(c', cl', nv)]_r^L \\ &\quad * \exists lay'. l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _, lay')]_r^E \wedge \overline{lay} > lay' \\ R_{io} &\triangleq \{((0, d), (1, d)) \mid d \in \text{Bool}\} \cup \{((1, d), (0, d)) \mid d \in \text{Bool}\} \cup \{((l, \mathbf{false}), (l, \mathbf{true})) \mid l \in \{0, 1\}\} \\ R' &\triangleq \{(((1, d), \mathbf{k}), ((0, d), \mathbf{0})) \mid d \in \text{Bool}\} \end{aligned}$$

 PROOF OF $\text{add}(x, e)$:

$1; \emptyset \vdash \forall S \in \mathcal{P}(\mathbb{Z}).$
 $\langle \text{LCSet}(s, x, S) \rangle$
 $\langle \text{lcset}_r(r', x, hl, S) * [\mathbb{E}]_r \rangle$
 $1; r : \forall S \in \mathcal{P}(\mathbb{Z}). S \rightarrow S \cup \{e\} \vdash$
 $\{ \exists S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond \}$
 $p := \text{locate}(x, e);$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, h, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(h, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay) \rfloor_r^{\perp} \wedge \\ v < e \leq \bar{v} \wedge lay > \bar{lay} \end{array} \right\}$
 $0; r : \forall S \in \mathcal{P}(\mathbb{Z}). S \rightarrow S \cup \{e\} \vdash$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, h, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(h, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay) \rfloor_r^{\perp} \wedge \\ v < e \leq \bar{v} \wedge lay > \bar{lay} \end{array} \right\}$
 $c := p.\text{next};$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, c, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay') \rfloor_r^{\perp} \wedge \\ v < e \leq \bar{v} \wedge lay > lay' > \bar{lay} \end{array} \right\}$
 $v := c.\text{val};$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, c, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay) \rfloor_r^{\perp} \wedge \\ v < e \leq \bar{v} \wedge lay > \bar{lay} \wedge v \neq e \Rightarrow e < \bar{v} \end{array} \right\}$
 $\text{if}(v \neq e) \{$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, c, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay) \rfloor_r^{\perp} \wedge \\ v < e < \bar{v} \wedge lay > \bar{lay} \end{array} \right\}$
 $n := \text{alloc}(3);$
 $n1 := \text{newLock}();$
 $n.\text{lock} := n1;$
 $n.\text{val} := e;$
 $n.\text{next} := c;$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, v, \bar{v}, h, lay, \bar{lay}, S. \text{lcset}_r(r', x, hl, S) * r \Rightarrow \diamond * \\ \lfloor \text{K}(p, la, v, c, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} * \lfloor \text{FREE}(lay) \rfloor_r^{\perp} * \\ (n \mapsto n1, e, c * \text{L}(s, n1, 0) \wedge v < e < \bar{v} \wedge lay > \bar{lay}) \end{array} \right\}$
 $p.\text{next} := n;$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, lay, \bar{lay}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \lfloor \text{K}(p, la, v, n, lay + 1) \rfloor_r^{\perp} * \lfloor \text{K}(n, n1, e, c, lay) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} \wedge lay > \bar{lay} \end{array} \right\}$
 $\text{unlock}(n1);$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, lay, \bar{lay}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \lfloor \text{K}(p, la, v, n, lay) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} \wedge lay > \bar{lay} \end{array} \right\}$
 $\}$
 $\left\{ \begin{array}{l} \exists la, \bar{la}, lay, \bar{lay}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \lfloor \text{K}(p, la, v, _ , lay) \rfloor_r^{\perp} * \lfloor \text{K}(c, \bar{la}, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} \wedge lay > \bar{lay} \end{array} \right\}$
 $p1 := p.\text{lock};$
 $c1 := c.\text{lock};$
 $\left\{ \begin{array}{l} \exists lay, \bar{lay}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \lfloor \text{K}(p, p1, v, _ , lay) \rfloor_r^{\perp} * \lfloor \text{K}(c, c1, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} \wedge lay > \bar{lay} \end{array} \right\}$
 $\left\{ \begin{array}{l} \exists lay, \bar{lay}, v, \bar{v}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \lfloor \text{K}(p, p1, v, _ , lay) \rfloor_r^{\perp} * \lfloor \text{K}(c, c1, \bar{v}, _ , \bar{lay}) \rfloor_r^{\perp} \wedge lay > \bar{lay} \end{array} \right\}$
 $\text{unlock}(c1);$
 $\left\{ \begin{array}{l} \exists lay, v, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \lfloor \text{K}(p, p1, v, _ , lay) \rfloor_r^{\perp} \end{array} \right\}$
 $\text{unlock}(p1);$
 $\left\{ \begin{array}{l} \exists S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) \end{array} \right\}$
 $\langle \text{lcset}_r(r', hl, x, S \cup \{e\}) * [\mathbb{G}]_r \rangle$
 $\langle \text{LCSet}(s, x, S \cup \{e\}) \rangle$

CONS: Sub s = (r, r', hl)
 MKATOM
 HELM: Q; CONS; FRAME

Fig. 15. Proof outline of add operation.

$$\left\{ \begin{array}{l} \exists lay, \overline{lay}, v, \overline{v}, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \\ \text{[K}(p, pl, v, _ \text{lay})]_{r'}^{\perp} * \text{[K}(c, cl, \overline{v}, _ \text{lay})]_{r'}^{\perp} \wedge lay > \overline{lay} \end{array} \right\} \\
\text{FRAME} \left\{ \begin{array}{l} \overline{lay}; r : \forall S \in \mathcal{P}(\mathbb{Z}). S \rightarrow S \cup \{e\} \vdash \\ \{ \exists \overline{v}, S'. \text{lcset}_r(r', x, hl, S') * \\ \text{[K}(c, cl, \overline{v}, _ \text{lay})]_{r'}^{\perp} * \\ \text{unlock}(cl); \\ \{ \exists \overline{v}, S'. \text{lcset}_r(r', x, hl, S') \} \end{array} \right\} \\
\left\{ \exists lay, v, S, S'. \text{lcset}_r(r', x, hl, S') * r \Rightarrow (S, S \cup \{e\}) * \text{[K}(p, pl, v, _ \text{lay})]_{r'}^{\perp} \right\}$$

Fig. 16. Details of unlock in add.

PROOF OF locate(x,e):

```

1; ∅ ⊢
{∃S. lcsetr(r', x, hl, S)}
p := x;
{∃S. lcsetr(r', x, hl, S) ∧ p = x}
pl := p.lock;
{∃S. lcsetr(r', pl, x, S) ∧ p = x}
lock(pl);
{∃lay, S. lcsetr(r', pl, x, S) * [K(p, pl, -∞, _ lay)]r'⊥}
c := p.next;
{∃lay, S. lcsetr(r', x, hl, S) * [K(p, pl, -∞, c, lay)]r'⊥}
cl := c.lock;
{∃lay, S. lcsetr(r', x, hl, S) * [K(p, pl, -∞, c, lay)]r'⊥ * [W(c, cl, _)]r'⊥}
lock(cl);
{∃lay,  $\overline{lay}$ , S. lcsetr(r', x, hl, S) * [K(p, pl, -∞, c, lay + 1)]r'⊥ * [K(c, cl, _ ,  $\overline{lay}$ )]r'⊥ * [FREE(lay)]r'⊥}
v := c.value;
{∃lay,  $\overline{lay}$ , S. lcsetr(r', x, hl, S) *
{ [K(p, pl, -∞, c, lay + 1)]r'⊥ * [K(c, cl, v, _ ,  $\overline{lay}$ )]r'⊥ * [FREE(lay)]r'⊥ ∧ lay >  $\overline{lay}$  }
{∃lay,  $\overline{lay}$ , pl, cl, lv, S. lcsetr(r', x, hl, S) *
{ [K(p, pl, lv, c, lay + 1)]r'⊥ * [K(c, cl, v, _ ,  $\overline{lay}$ )]r'⊥ * [FREE(lay)]r'⊥ ∧
lay >  $\overline{lay}$  ∧ lv < e }
while(v < e) {
  pl := p.lock;
  c' := c.next;
  cl' := c'.lock;
  lock(cl');
  v := c'.val;
  unlock(pl);
  p := c;
  c := c';
}
{∃l,  $\overline{l}$ , v,  $\overline{v}$ , h, lay,  $\overline{lay}$ , S. lcsetr(r', x, hl, S) *
{ [K(p, l, v, h, lay + 1)]r'⊥ * [K(h,  $\overline{l}$ ,  $\overline{v}$ , _ ,  $\overline{lay}$ )]r'⊥ *
[FREE(lay)]r'⊥ ∧ lay >  $\overline{lay}$  ∧ v < e ≤  $\overline{v}$  }
return p;
{∃l,  $\overline{l}$ , v,  $\overline{v}$ , h, lay,  $\overline{lay}$ , S. lcsetr(r', x, hl, S) *
{ [K(ret, l, v, h, lay + 1)]r'⊥ * [K(h,  $\overline{l}$ ,  $\overline{v}$ , _ ,  $\overline{lay}$ )]r'⊥ *
[FREE(lay)]r'⊥ ∧ lay >  $\overline{lay}$  ∧ v < e ≤  $\overline{v}$  }

```

Fig. 17. Proof outline of locate.

$$\begin{array}{l}
\mathbf{1; 0} \vdash \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, pl, cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, pl, lv, c, lay + 1)]_r^L * [\mathbf{K}(c, cl, v, _ \overline{lay})]_r^L * [\mathbf{FREE}(lay)]_r^L \wedge \\ \overline{lay} > lay \wedge lv < e \end{array} \right\} \\
\mathbf{while}(v < e) \{ \\
\forall \beta. \mathbf{1; 0} \vdash \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, pl, cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, pl, lv, c, lay + 1)]_r^L * [\mathbf{K}(c, cl, v, _ \overline{lay})]_r^L * [\mathbf{FREE}(lay)]_r^L * \\ lv < e \wedge v < e \wedge \beta \geq \overline{lay} > lay \end{array} \right\} \\
pl := p.lock; \\
c' := c.next; \\
cl' := c'.lock; \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, cl, lv, nv, S, ls, ls', l. \text{lcset}_r(r', x, hl, S) * \text{lclist}_r(x, hl, ls \oplus ((nv, l) : ls')) * \\ [\mathbf{K}(p, pl, lv, c, lay + 1)]_r^L * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_r^L * [\mathbf{W}(c', cl', nv)]_r^L * [\mathbf{FREE}(\overline{lay})]_r^L * \\ l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _)]_r^E \wedge lv < e \wedge v < e \wedge \beta \geq \overline{lay} > lay \end{array} \right\} \\
\text{lock}(cl'); \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, lay', cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, pl, lv, c, lay + 1)]_r^L * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_r^L * [\mathbf{K}(c', cl', _ _ , lay')]_r^L * \\ [\mathbf{FREE}(\overline{lay})]_r^L \wedge lv < e \wedge v < e \wedge \beta \geq \overline{lay} > lay > lay' \end{array} \right\} \\
v := c'.val; \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, lay', cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, pl, _ , c, lay + 1)]_r^L * [\mathbf{K}(c, cl, lv, c', \overline{lay} + 1)]_r^L * [\mathbf{K}(c', cl', v, _ , lay')]_r^L * \\ [\mathbf{FREE}(\overline{lay})]_r^L \wedge lv < e \wedge \beta \geq \overline{lay} > lay > lay' \end{array} \right\} \\
\text{unlock}(pl); \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(c, cl, lv, c', lay + 1)]_r^L * [\mathbf{K}(c', cl', v, _ , \overline{lay})]_r^L * \\ [\mathbf{FREE}(lay)]_r^L \wedge lv < e \wedge \beta > lay > \overline{lay} \end{array} \right\} \\
p := c; \\
c := c'; \\
\left\{ \begin{array}{l} \exists \overline{lay}, \overline{lay}, pl, cl, lv, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, pl, lv, c', lay + 1)]_r^L * [\mathbf{K}(c, cl, v, _ , \overline{lay})]_r^L * \\ [\mathbf{FREE}(lay)]_r^L \wedge lv < e \wedge \beta > lay > \overline{lay} \end{array} \right\} \\
\} \\
\left\{ \begin{array}{l} \exists l, \bar{l}, v, \bar{v}, h, lay, \overline{lay}, S. \text{lcset}_r(r', x, hl, S) * \\ [\mathbf{K}(p, l, v, h, lay + 1)]_r^L * [\mathbf{K}(h, \bar{l}, \bar{v}, _ , \overline{lay})]_r^L * \\ [\mathbf{FREE}(lay)]_r^L \wedge lay > \overline{lay} \wedge v < e \leq \bar{v} \end{array} \right\}
\end{array}$$

Fig. 18. Details of while loop in locate.

$$\left\{ \begin{array}{l}
\exists \overline{lay}, \overline{lay}', cl, lv, nv, S, ls, ls', l. \text{lcset}_r(r', x, \text{hl}, S) * \text{lclist}_{r'}(x, \text{hl}, ls \oplus ((nv, l) : ls')) * \\
[\mathbf{K}(p, pl, lv, c, lay + 1)]_{r'}^L * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{W}(c', cl', nv)]_{r'}^L * [\mathbf{FREE}(\overline{lay})]_{r'}^L * \\
l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _ , lay')]_{r'}^E \wedge lv < e \wedge v < e \wedge \beta \geq lay > \overline{lay} > lay'
\end{array} \right\}$$

$$\left\{ \begin{array}{l}
\overline{lay}; \emptyset \vdash \\
\mathbf{W}ls, ls' \in \mathbb{Z}^*, l \in \{0, 1\}. \\
\langle \text{lclist}_{r'}(x, \text{hl}, ls \oplus ((nv, l) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{W}(c', cl', nv)]_{r'}^L * \\
\exists lay'. l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _ , lay')]_{r'}^E \wedge \overline{lay} > lay' \rangle
\end{array} \right\}$$

$$\left\{ \begin{array}{l}
\overline{lay}; \emptyset \vdash \\
\mathbf{W}ls, ls' \in \mathbb{Z}^*, l \in \{0, 1\} \rightarrow \overline{lay} \{0\}. \\
\langle \text{lclist}_{r'}(x, \text{hl}, ls \oplus ((nv, l) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{W}(c', cl', nv)]_{r'}^L * \\
\exists lay'. l = 1 \Rightarrow [\mathbf{K}(c', cl', nv, _ , lay')]_{r'}^E \wedge \overline{lay} > lay' \rangle
\end{array} \right\}$$

$$\left\{ \begin{array}{l}
\mathbf{W}l \in \{0, 1\} \rightarrow \overline{lay} \{0\}. \\
\langle \text{Tlock}(s, cl', l) \rangle \\
\text{lock}(cl'); \\
\langle \text{Tlock}(s, cl', 1) \wedge l = 0 \rangle \\
\langle \text{lclist}_{r'}(x, \text{hl}, ls \oplus ((nv, 1) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{K}(c', cl', _ , _ , lay')]_{r'}^L \rangle \\
\langle \text{lclist}_{r'}(x, \text{hl}, ls \oplus ((nv, 1) : ls')) * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{K}(c', cl', _ , _ , lay')]_{r'}^L \rangle
\end{array} \right\}$$

$$\left\{ \begin{array}{l}
\exists \overline{lay}, \overline{lay}', lay', cl, lv, S. \text{lcset}_r(r', x, \text{hl}, S) * \\
[\mathbf{K}(p, pl, lv, c, lay + 1)]_{r'}^L * [\mathbf{K}(c, cl, v, c', \overline{lay} + 1)]_{r'}^L * [\mathbf{K}(c', cl', _ , _ , lay')]_{r'}^L * \\
[\mathbf{FREE}(lay)]_{r'}^L \wedge lv < e \wedge v < e \wedge \beta \geq lay > \overline{lay} > lay'
\end{array} \right\}$$

Fig. 19. Details of lock in while loop of locate.

STEP 11 is $\exists\text{ELIM}$, ATOMW , $\text{A}\exists\text{ELIM}$, LIFTA , QL , FRAME .STEP 12 is LIFTA , CONS , FRAME .

D LANGUAGE DEFINITION

We will make regular use of partial functions. We write $X \rightarrow Y$ for the set of partial function from X to Y , and $X \rightarrow_{\text{fin}} Y$ for the set of finite partial function. Given $f: X \rightarrow Y$, we write $f(x) = \perp$ if f is undefined on x , and $\text{dom}(f) \triangleq \{x \mid f(x) \neq \perp\}$. We will use the notation $[x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$ for the finite function that maps each of the x_i to y_i and is undefined on any other input. Given elements $x \in X$ and $y \in Y$, and functions $f: X \rightarrow Y$ and $g: X' \rightarrow Y'$, we define the following function $f[x \mapsto y]$ and $f \uplus g$ by:

$$(f[x \mapsto y])(z) \triangleq \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

$$(f \uplus g)(x) \triangleq \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ g(x) & \text{if } x \in \text{dom}(g) \end{cases} \quad \text{if } \text{dom}(f) \cap \text{dom}(g) = \emptyset$$

We write $f[x \mapsto \perp]$ for the partial function that is undefined on x but otherwise behaves like f . The union of two partial function $f \cup g$ is a well-defined partial function as long as $f(x) = g(x)$ where their domains overlap.

Definition D.1 (PCM). A (multi-unity) *partial commutative monoid* (PCM) is a tuple (X, \bullet, E) comprising a set X , a binary *partial* composition operation $\bullet: X \times X \rightarrow X$ and a set of unit elements E , such that the following axioms are satisfied (where either both sides are defined and equal, or both sides are undefined):

$$\begin{aligned} \forall x, y, z \in X. \quad (x \bullet y) \bullet z &= x \bullet (y \bullet z) && \text{(associativity)} \\ \forall x, y \in X. \quad x \bullet y &= y \bullet x && \text{(commutativity)} \\ \forall x \in X. \exists e \in E. \quad x \bullet e &= x && \text{(identity)} \end{aligned}$$

For $x, y \in X$, we write $x \# y$ if $x \bullet y \neq \perp$, and $x \sqsubseteq y$ if $\exists x_1. y = x \bullet x_1$.

We use the *set of Booleans*, $\text{Bool} \triangleq \{\text{True}, \text{False}\} \ni b, b_1, b_2$, a *set of values*, $\text{Val} \triangleq \mathbb{Z} \cup \text{Bool} \ni v, v_1, v_2, \dots$, a *set of program variables*, $\text{PVar} \ni x, y, \dots$, and a *set of function names*, $\text{FName} \ni f, g, \dots$. The set PVar contains a special element, ret , that holds a function's return value. Heap addresses are represented by natural numbers, $\text{Addr} \triangleq \mathbb{N}$. The natural numbers in Val represent both numeric values and heap addresses.

Definition D.2 (Numeric and Boolean Expressions). Let Vars be an arbitrary set of variables, and Values an arbitrary set of values. The *set of numerical expressions*, $\text{Exp}(\text{Vars}, \text{Values}) \ni \mathbb{E}, \mathbb{E}_1, \mathbb{E}_2, \dots$, and the *set of boolean expressions*, $\text{BExp}(\text{Vars}, \text{Values}) \ni \mathbb{B}, \mathbb{B}_1, \mathbb{B}_2, \dots$, are defined by the grammars:

$$\begin{aligned} \mathbb{E} ::= v \mid x \mid \mathbb{E} + \mathbb{E} \mid \mathbb{E} - \mathbb{E} \mid \mathbb{E} * \mathbb{E} \mid \dots & \quad \text{where } v \in \text{Values}, x \in \text{Vars} \\ \mathbb{B} ::= b \mid x \mid \neg \mathbb{B} \mid \mathbb{B} \wedge \mathbb{B} \mid \mathbb{E} = \mathbb{E} \mid \mathbb{E} < \mathbb{E} \mid \dots & \quad \text{where } b \in \text{Bool}, x \in \text{Vars} \end{aligned}$$

The numeric and Boolean program expressions are defined by the sets $\text{Exp}(\text{PVar}, \text{Val})$ and $\text{BExp}(\text{PVar}, \text{Val})$ respectively. In Appendix E, we also work with logical expressions built from both program and logical variables and values, hence the reason for the expression definition defined over an arbitrary variable and value sets.

The functions $\text{fv}_{\mathbb{E}}$ and $\text{fv}_{\mathbb{B}}$ provide the sets of free variables for the numeric and Boolean expressions respectively. They are defined inductively on the structure of expressions by:

$\begin{aligned} \text{fv}_{\mathcal{E}}(v) &= \emptyset & v \in \text{Values} \\ \text{fv}_{\mathcal{E}}(x) &= \{x\} & x \in \text{Vars} \\ \text{fv}_{\mathcal{E}}(\mathbb{E}_1 + \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \text{fv}_{\mathcal{E}}(\mathbb{E}_1 - \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \text{fv}_{\mathcal{E}}(\mathbb{E}_1 * \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \dots \end{aligned}$	$\begin{aligned} \text{fv}_{\mathcal{B}}(b) &= \emptyset & b \in \{\text{True}, \text{False}\} \\ \text{fv}_{\mathcal{B}}(x) &= \{x\} & x \in \text{Vars} \\ \text{fv}_{\mathcal{B}}(\neg \mathbb{B}) &= \text{fv}_{\mathcal{B}}(\mathbb{B}) \\ \text{fv}_{\mathcal{B}}(\mathbb{B}_1 \wedge \mathbb{B}_2) &= \text{fv}_{\mathcal{B}}(\mathbb{B}_1) \cup \text{fv}_{\mathcal{B}}(\mathbb{B}_2) \\ \text{fv}_{\mathcal{B}}(\mathbb{E}_1 = \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \text{fv}_{\mathcal{B}}(\mathbb{E}_1 < \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \dots \end{aligned}$
---	---

Definition D.3 (Commands). The set of commands, $\text{Cmd} \ni \mathbb{C}$, is defined by the grammar:

$\begin{aligned} \mathbb{C} &::= \text{skip} \\ & x := \mathbb{E} \\ & x := [\mathbb{E}] \\ & [\mathbb{E}] := \mathbb{E} \\ & x := \text{CAS}(\mathbb{E}, \mathbb{E}, \mathbb{E}) \\ & x := \text{FAS}(\mathbb{E}, \mathbb{E}, \mathbb{E}) \\ & x := \text{alloc}(\mathbb{E}) \\ & \text{dealloc}(\mathbb{E}) \\ & \text{let } f(\vec{x}) = \mathbb{C} \text{ in } \mathbb{C} \\ & \text{var } x = \mathbb{E} \text{ in } \mathbb{C} \\ & \text{if}(\mathbb{B})\{\mathbb{C}\}\text{else}\{\mathbb{C}\} \\ & \text{while}(\mathbb{B})\{\mathbb{C}\} \\ & x := f(\vec{\mathbb{E}}) \\ & \mathbb{C}; \mathbb{C} \\ & \mathbb{C} \parallel \mathbb{C} \\ & \langle \mathbb{C} \rangle \end{aligned}$	$\begin{aligned} &(\text{skip}) \\ &(\text{assignment}) \\ &(\text{read}) \\ &(\text{mutate}) \\ &(\text{CAS}) \\ &(\text{FAS}) \\ &(\text{allocate}) \\ &(\text{deallocate}) \\ &(\text{function definition}) \\ &(\text{local variable binding}) \\ &(\text{if}) \\ &(\text{while}) \\ &(\text{function call}) \\ &(\text{sequential composition}) \\ &(\text{parallel composition}) \\ &(\text{primitive atomic block}) \end{aligned}$
---	---

where $\mathbb{E} \in \text{Exp}(\text{PVar}, \text{Val})$, $\mathbb{B} \in \text{BExp}(\text{PVar}, \text{Val})$, $x \in \text{PVar}$, $\vec{x} \in \text{PVar}^*$ is a list of pairwise distinct variables, and $f \in \text{FName}$.

We use $[\mathbb{E}]$ to denote the value of the heap cell with address given by \mathbb{E} . In Fig. 20, we define operators fv and mods , which identify the variables that a command can access and the variables that are potentially modified by a command, respectively. In a command $\mathbb{C}_1 \parallel \mathbb{C}_2$, we apply a strong syntactic restriction that $\text{mods}(\mathbb{C}_1) = \text{mods}(\mathbb{C}_2) = \emptyset$. Each individual thread is still able to modify variables that are created locally and to modify shared heap cells, but are not allowed to modify the free variables.⁹ In a function definition $\text{let } f(x_1, \dots, x_n) = \mathbb{C}_1 \text{ in } \mathbb{C}_2$, we use the natural restriction $\text{fv}(\mathbb{C}_1) \subseteq \{x_1, \dots, x_n, \text{ret}\}$. Also for simplicity, we assume each function name is given a definition at most once. The function $\text{fn}: \text{Cmd} \rightarrow \wp(\text{FName})$ returns the function names occurring in Cmd that are not bound by a **let**.

Definition D.4 (Variable Store). A program variable store, $\sigma \in \text{Store} \triangleq \text{PVar} \rightarrow \text{Val}$, is a finite partial function from program variables to values. The *right-biased union* of variable stores, $\sigma_1 \triangleleft \sigma_2$, is defined by:

$$(\sigma_1 \triangleleft \sigma_2)(x) = \begin{cases} \sigma_2(x) & \text{if } x \in \text{dom}(\sigma_2) \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

⁹To lift this restriction, one could use standard techniques, such as “variables as resources”. Our restriction minimises the noise generated by handling local state in the formalisation of the model and the assertions. Note that expressivity is not really limited by our restriction: any local variable in the scope common to both thread that needs to be modified can be instead implemented by using a shared memory cell.

$\begin{aligned} \text{fv}(\text{skip}) &= \emptyset \\ \text{fv}(x := \mathbb{E}) &= \{x\} \cup \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}(x := [\mathbb{E}]) &= \{x\} \cup \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}([\mathbb{E}_1] := \mathbb{E}_2) &= \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \\ \text{fv}(x := \text{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)) &= \{x\} \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_1) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_2) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}_3) \\ \text{fv}(x := \text{alloc}(\mathbb{E})) &= \{x\} \cup \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}(\text{dealloc}(\mathbb{E})) &= \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}(\text{let } f(\vec{x}) = C_f \text{ in } C) &= \text{fv}(C) \\ \text{fv}(\text{var } x = \mathbb{E} \text{ in } C) &= (\text{fv}(C) \setminus \{x\}) \cup \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}(\text{if}(\mathbb{B})\{C_1\}\text{else}\{C_2\}) &= \text{fv}_{\mathcal{B}}(\mathbb{B}) \cup \text{fv}(C_1) \cup \text{fv}(C_2) \\ \text{fv}(\text{while}(\mathbb{B})\{C\}) &= \text{fv}_{\mathcal{B}}(\mathbb{B}) \cup \text{fv}(C) \\ \text{fv}(x := f(\vec{\mathbb{E}})) &= \{x\} \cup \text{fv}_{\mathcal{E}}(\mathbb{E}) \\ \text{fv}(C_1; C_2) &= \text{fv}(C_1) \cup \text{fv}(C_2) \\ \text{fv}(C_1 \parallel C_2) &= \text{fv}(C_1) \cup \text{fv}(C_2) \end{aligned}$	$\begin{aligned} \text{mods}(\text{skip}) &= \emptyset \\ \text{mods}(x := \mathbb{E}) &= \{x\} \\ \text{mods}(x := [\mathbb{E}]) &= \{x\} \\ \text{mods}([\mathbb{E}_1] := \mathbb{E}_2) &= \emptyset \\ \text{mods}(x := \text{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)) &= \{x\} \\ \text{mods}(x := \text{alloc}(\mathbb{E})) &= \{x\} \\ \text{mods}(\text{dealloc}(\mathbb{E})) &= \emptyset \\ \text{mods}(\text{let } f(\vec{x}) = C_f \text{ in } C) &= \text{mods}(C) \\ \text{mods}(\text{var } x = \mathbb{E} \text{ in } C) &= \text{mods}(C) \setminus \{x\} \\ \text{mods}(\text{if}(\mathbb{B})\{C_1\}\text{else}\{C_2\}) &= \text{mods}(C_1) \cup \text{mods}(C_2) \\ \text{mods}(\text{while}(\mathbb{B})\{C\}) &= \text{mods}(C) \\ \text{mods}(x := f(\vec{\mathbb{E}})) &= \{x\} \\ \text{mods}(C_1; C_2) &= \text{mods}(C_1) \cup \text{mods}(C_2) \\ \text{mods}(C_1 \parallel C_2) &= \text{mods}(C_1) \cup \text{mods}(C_2) \end{aligned}$
--	---

Fig. 20. The sets of free and modified program variables

Definition D.5 (Expression evaluation). Let $\zeta : \text{Vars} \rightarrow_{\text{fin}} \text{Values}$ be an arbitrary function from an arbitrary set of variables to values. The *numeric expression evaluation function*, $\mathcal{E}[\cdot]_{\zeta} : \text{Exp}(\text{Vars}, \text{Values}) \rightarrow \text{Values}$, and the *Boolean expression evaluation function*, $\mathcal{B}[\cdot]_{\zeta} : \text{BExp}(\text{Vars}, \text{Values}) \rightarrow \text{Bool}$, are defined by:

$$\begin{array}{ll} \mathcal{E}[v]_{\zeta} = v & \mathcal{B}[b]_{\zeta} = b \\ \mathcal{E}[x]_{\zeta} = \zeta(x) & \mathcal{B}[\neg \mathbb{B}]_{\zeta} = \neg \mathcal{B}[\mathbb{B}]_{\zeta} \\ \mathcal{E}[\mathbb{E}_1 + \mathbb{E}_2]_{\zeta} = \mathcal{E}[\mathbb{E}_1]_{\zeta} + \mathcal{E}[\mathbb{E}_2]_{\zeta} & \mathcal{B}[\mathbb{B}_1 \wedge \mathbb{B}_2]_{\zeta} = \mathcal{B}[\mathbb{B}_1]_{\zeta} \wedge \mathcal{B}[\mathbb{B}_2]_{\zeta} \\ \mathcal{E}[\mathbb{E}_1 - \mathbb{E}_2]_{\zeta} = \mathcal{E}[\mathbb{E}_1]_{\zeta} - \mathcal{E}[\mathbb{E}_2]_{\zeta} & \mathcal{B}[\mathbb{E}_1 = \mathbb{E}_2]_{\zeta} = (\mathcal{E}[\mathbb{E}_1]_{\zeta} = \mathcal{E}[\mathbb{E}_2]_{\zeta}) \\ \mathcal{E}[\mathbb{E}_1 \cdot \mathbb{E}_2]_{\zeta} = \mathcal{E}[\mathbb{E}_1]_{\zeta} \cdot \mathcal{E}[\mathbb{E}_2]_{\zeta} & \mathcal{B}[\mathbb{E}_1 < \mathbb{E}_2]_{\zeta} = (\mathcal{E}[\mathbb{E}_1]_{\zeta} < \mathcal{E}[\mathbb{E}_2]_{\zeta}) \\ \dots & \dots \end{array}$$

The program expressions are evaluated using program store $\sigma \in \text{Store} \triangleq \text{PVar} \rightarrow_{\text{fin}} \text{Val}$. In Appendix E, we also work with logical expressions which are evaluated over both program and logical variables and values. The right-biased union of stores is used to describe how, when nesting scopes, a variable occurrence is bound by the innermost binder surrounding it. The notation $\text{var } x_1, x_2, \dots, x_n \text{ in } C$ denotes $\text{var } x_1 = \emptyset \text{ in } \text{var } x_2 = \emptyset \text{ in } \dots \text{var } x_n = \emptyset \text{ in } C$.

Definition D.6 (Heap). A *heap*, $h \in \text{Heap} \triangleq \text{Addr} \rightarrow_{\text{fin}} \text{Val}$, is a finite partial function from addresses to values. The *set of heaps*, Heap , forms a PCM $(\text{Heap}, \uplus, \{\emptyset\})$ with $h_1 \uplus h_2$ defined only if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$.

Definition D.7 (Function Implementation Context). A *function implementation context*, $\varphi \in \text{FImpl} \triangleq \text{FName} \rightarrow (\text{PVar}^*, \text{Cmd})$, is a finite partial function from function names to pairs comprising a finite list of distinct variables and a command.

We write $\varphi(f) = (\vec{x}, C)$, where variable list \vec{x} represents the function arguments and C represents the function body. We use the notation φ_{var} and φ_{cmd} to refer to the arguments and function body of f respectively.

In order to describe the behaviour of local variable binding and function calls, we define program states which extends commands with variable stores. For example, the program state (σ, C) indicates that the command C is evaluated in the current store updated with the variables in σ .

Definition D.8 (Program States). The set of program states, $\text{PState} \ni C, C_1, C_2, \dots$ is defined by the grammar:

$$C ::= \checkmark \mid (\sigma, C) \mid C; \mathbb{C} \mid \mathbf{let} \ f(\vec{x}) = \mathbb{C} \ \mathbf{in} \ C \mid C \parallel C \mid \mathbb{C}$$

The \checkmark indicates a terminated program. It is a technical device so that every $\mathbb{C} \in \text{Cmd}$, including **skip**, takes at least one step.

In the operational semantics, we need to keep track of which thread is originating each step to be able to define later concepts of fairness of the scheduling. We do this tracking using *thread identifiers* $t \in \text{TId} \triangleq \{\mathsf{L}, \mathsf{R}\}^*$ which are strings of letters L (for the left thread) and R (for the right thread). ϵ will be used to denote the thread identifier which is an empty sequence. Intuitively, such a string identifies a single thread as the path in the syntax tree of parallel compositions at which the thread is found.

Definition D.9 (Command Semantics). A scheduler annotation t is an element of the set

$$\text{Sched} \triangleq \{\text{loc}_t \mid t \in \text{TId}\} \uplus \{\text{env}\}.$$

A program configuration c is an element of the set $\text{PConf} \triangleq (\text{Store} \times \text{Heap} \times \text{PState}) \uplus \{\frac{1}{2}\}$. Let $\varphi \in \text{Flmpl}$. The operational semantics of the commands is given by the labelled relation, $\longrightarrow_\varphi \subseteq \text{PConf} \times \text{Sched} \times \text{PConf}$, defined in Fig. 21 and Fig. 22. We write $a \xrightarrow{t}_\varphi b$ for $(a, t, b) \in \longrightarrow_\varphi$. We also define $\xrightarrow{\text{loc}^*}_\varphi \triangleq (\cup_{t \in \text{TId}} \xrightarrow{\text{loc}_t}_\varphi)^*$.

To simplify the development, in our programming language the initial state’s store assigns arbitrary values to the free variables of a program. With such assumption, every reference to a local variable will be in the domain of the current store. This ensures that in every application of the rules in Fig. 21 and Fig. 22 to construct a trace, the evaluations of (boolean) expressions are well-defined.

Definition D.10 (Threads). Given a program state $C \in \text{PState}$, the set $\text{threads}(C)$ is the set of threads of C that can take a step. The function $\text{threads}: \text{PState} \rightarrow \mathcal{P}(\text{TId})$ is defined as follows:

$$\begin{aligned} \text{threads}(\checkmark \parallel \checkmark) &\triangleq \{\epsilon\} \\ \text{threads}(C_1 \parallel C_2) &\triangleq \{\mathsf{L}t \mid t \in \text{threads}(C_1)\} \cup \{\mathsf{R}t \mid t \in \text{threads}(C_2)\} \\ \text{threads}(\checkmark; \mathbb{C}) &\triangleq \{\epsilon\} \\ \text{threads}(C; \mathbb{C}) &\triangleq \text{threads}(C) \\ \text{threads}(\mathbf{let} \ f(\vec{x}) = \mathbb{C} \ \mathbf{in} \ C) &\triangleq \text{threads}(C) \\ \text{threads}((\sigma, C)) &\triangleq \text{threads}(C) \\ \text{threads}(\checkmark) &\triangleq \emptyset \\ \text{threads}(_) &\triangleq \{\epsilon\} \end{aligned}$$

Definition D.11 (Program Traces and Fairness). We call *program traces*, the infinite sequences of the form $c_0 \ \pi_0 \ c_1 \ \pi_1 \ \dots$ where, for all $i \in \mathbb{N}$, $c_i \in \text{PConf}$, $\pi_i \in \text{Sched}$. We use τ for ranging over infinite suffixes of program traces and PTrace for the set of all program traces. For a program trace $\tau = c_0 \ \pi_0 \ c_1 \ \pi_1 \ \dots$, we define $\tau(i) \triangleq (c_i, \pi_i)$, and $\tau/i \triangleq c_i \ \pi_i \ c_{i+1} \ \pi_{i+1} \ \dots$. We define the set of φ -program traces

$$\text{PTrace}_\varphi \triangleq \{c_0 \ \pi_0 \ c_1 \ \pi_1 \ \dots \mid \forall i \in \mathbb{N}. c_i \xrightarrow{\pi_i}_\varphi c_{i+1}\}.$$

A program trace $(c_0 \ \pi_0 \ c_1 \ \pi_1 \ \dots) \in \text{PTrace}_\varphi$ is (weakly) fair if and only if:

$$\forall i \in \mathbb{N}. \forall C \in \text{PState}. c_i = (_, _, C) \Rightarrow \forall t \in \text{threads}(C). \exists j \geq i. (\pi_j = \text{loc}_t \vee c_j = \frac{1}{2}) \quad (12)$$

$$\forall i \in \mathbb{N}. \exists j \geq i. \pi_j = \text{env} \quad (13)$$

$$\begin{array}{c}
\frac{}{\sigma, h, \text{skip} \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, \checkmark} \qquad \frac{}{\sigma, h, x := \mathbb{E} \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto \mathcal{E}[\mathbb{E}]_\sigma], h, \checkmark} \\
\frac{\mathcal{E}[\mathbb{E}]_\sigma \in \text{dom}(h)}{\sigma, h, x := [\mathbb{E}] \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto h(\mathcal{E}[\mathbb{E}]_\sigma)], h, \checkmark} \qquad \frac{\mathcal{E}[\mathbb{E}_1]_\sigma \in \text{dom}(h)}{\sigma, h, [\mathbb{E}_1] := \mathbb{E}_2 \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h[\mathcal{E}[\mathbb{E}_1]_\sigma \mapsto \mathcal{E}[\mathbb{E}_2]_\sigma], \checkmark} \\
\frac{\mathcal{E}[\mathbb{E}_1]_\sigma \in \text{dom}(h) \quad h(\mathcal{E}[\mathbb{E}_1]_\sigma) = \mathcal{E}[\mathbb{E}_2]_\sigma}{\sigma, h, x := \text{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto 1], h[\mathcal{E}[\mathbb{E}_1]_\sigma \mapsto \mathcal{E}[\mathbb{E}_3]_\sigma], \checkmark} \\
\frac{\mathcal{E}[\mathbb{E}_1]_\sigma \in \text{dom}(h) \quad h(\mathcal{E}[\mathbb{E}_1]_\sigma) \neq \mathcal{E}[\mathbb{E}_2]_\sigma}{\sigma, h, x := \text{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto 0], h, \checkmark} \qquad \frac{a = \mathcal{E}[\mathbb{E}_1]_\sigma \in \text{dom}(h) \quad v = \mathcal{E}[\mathbb{E}_2]_\sigma}{\sigma, h, x := \text{FAS}(\mathbb{E}_1, \mathbb{E}_2) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto h(a)], h[a \mapsto v], \checkmark} \\
\frac{l = \mathcal{E}[\mathbb{E}]_\sigma \quad l > 0 \quad \{r, r+1, \dots, r+l-1\} \cap \text{dom}(h) = \emptyset \quad v_0, v_1, \dots, v_{l-1} \in \text{Val}}{\sigma, h, x := \text{alloc}(\mathbb{E}) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma[x \mapsto r], h[r \mapsto v_0, r+1 \mapsto v_1, \dots, r+l-1 \mapsto v_{l-1}], \checkmark} \\
\frac{\mathcal{E}[\mathbb{E}]_\sigma \in \text{dom}(h)}{\sigma, h, \text{dealloc}(\mathbb{E}) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h[\mathcal{E}[\mathbb{E}]_\sigma \mapsto \perp], \checkmark} \\
\frac{\sigma, h, C \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C' \quad \varphi' = \varphi[f \mapsto (\vec{x}, C_f)]}{\sigma, h, \text{let } f(\vec{x}) = C_f \text{ in } C \xrightarrow{\text{loc}_t}_\varphi \sigma', h', \text{let } f(\vec{x}) = C_f \text{ in } C'} \qquad \frac{}{\sigma, h, \text{let } f(\vec{x}) = C_f \text{ in } \checkmark \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, \checkmark} \\
\frac{}{\sigma, h, \text{var } x = \mathbb{E} \text{ in } C \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, ([x \mapsto \mathcal{E}[\mathbb{E}]_\sigma], C)} \qquad \frac{}{\sigma, h, (\sigma', \checkmark) \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, \checkmark} \\
\frac{\sigma \triangleleft \sigma_1, h, C \xrightarrow{\text{loc}_t}_\varphi \sigma' \triangleleft \sigma'_1, h', C' \quad \text{dom}(\sigma) = \text{dom}(\sigma') \quad \text{dom}(\sigma_1) = \text{dom}(\sigma'_1)}{\sigma, h, (\sigma_1, C) \xrightarrow{\text{loc}_t}_\varphi \sigma', h', (\sigma'_1, C')} \\
\frac{\mathcal{B}[\mathbb{B}]_\sigma}{\sigma, h, \text{if}(\mathbb{B})\{C_1\}\text{else}\{C_2\} \xrightarrow{\text{loc}_t}_\varphi \sigma, h, C_1} \qquad \frac{\neg \mathcal{B}[\mathbb{B}]_\sigma}{\sigma, h, \text{if}(\mathbb{B})\{C_1\}\text{else}\{C_2\} \xrightarrow{\text{loc}_t}_\varphi \sigma, h, C_2} \\
\frac{\mathcal{B}[\mathbb{B}]_\sigma}{\sigma, h, \text{while}(\mathbb{B})\{C\} \xrightarrow{\text{loc}_t}_\varphi \sigma, h, C; \text{while}(\mathbb{B})\{C\}} \qquad \frac{\neg \mathcal{B}[\mathbb{B}]_\sigma}{\sigma, h, \text{while}(\mathbb{B})\{C\} \xrightarrow{\text{loc}_t}_\varphi \sigma, h, \checkmark} \\
\frac{\varphi(f) = (\vec{x}, C)}{\sigma, h, y := f(\vec{\mathbb{E}}) \xrightarrow{\text{loc}_t}_\varphi \sigma, h, \text{var } \text{ret} = \emptyset \text{ in } (\text{var } \vec{x} = \vec{\mathbb{E}} \text{ in } C); y := \text{ret}} \qquad \frac{\sigma, h, C_1 \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C'_1}{\sigma, h, C_1; C_2 \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C'_1; C'_2} \\
\frac{}{\sigma, h, \checkmark; C \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, C} \qquad \frac{\sigma, h, C_1 \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C'_1 \quad C_2 \neq \checkmark}{\sigma, h, C_1 \parallel C_2 \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C'_1 \parallel C_2} \qquad \frac{\sigma, h, C_2 \xrightarrow{\text{loc}_t}_\varphi \sigma', h', C'_2 \quad C_1 \neq \checkmark}{\sigma, h, C_1 \parallel C_2 \xrightarrow{\text{loc}_{rt}}_\varphi \sigma', h', C_1 \parallel C'_2} \\
\frac{}{\sigma, h, \checkmark \parallel \checkmark \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma, h, \checkmark} \qquad \frac{\sigma, h, C \xrightarrow{\text{loc}^*}_\varphi \sigma', h', \checkmark}{\sigma, h, \langle C \rangle \xrightarrow{\text{loc}_\epsilon}_\varphi \sigma', h', \checkmark} \qquad \frac{h' \in \text{Heap}}{\sigma, h, C \xrightarrow{\text{env}}_\varphi \sigma, h', C}
\end{array}$$

Fig. 21. The small-step operational semantics

$$\begin{array}{c}
\frac{\mathcal{E}[\mathbb{E}]_{\sigma} \notin \text{dom}(h)}{\sigma, h, t, x := [\mathbb{E}] \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \quad \frac{\mathcal{E}[\mathbb{E}_1]_{\sigma} \notin \text{dom}(h)}{\sigma, h, t, [\mathbb{E}_1] := \mathbb{E}_2 \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \quad \frac{\mathcal{E}[\mathbb{E}_1]_{\sigma} \notin \text{dom}(h)}{\sigma, h, t, x := \text{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3) \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \\
\\
\frac{\mathcal{E}[\mathbb{E}_1]_{\sigma} \notin \text{dom}(h)}{\sigma, h, t, x := \text{FAS}(\mathbb{E}_1, \mathbb{E}_2) \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \quad \frac{\mathcal{E}[\mathbb{E}]_{\sigma} \notin \text{dom}(h)}{\sigma, h, t, \text{dealloc}(\mathbb{E}) \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \\
\\
\frac{\sigma, h, t, C \xrightarrow{\text{loc}_t}_{\varphi'} \not\downarrow \quad \varphi' = \varphi[f \mapsto (\vec{x}, \mathbb{C}_f)]}{\sigma, h, t, \text{let } f(\vec{x}) = \mathbb{C}_f \text{ in } C \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow} \quad \frac{\sigma \triangleleft \sigma', h, t, C \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow}{\sigma, h, t, (\sigma', C) \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow} \\
\\
\frac{f \notin \text{dom}(\varphi)}{\sigma, h, y := f(\vec{\mathbb{E}}) \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \quad \frac{\sigma, h, t, C_1 \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow}{\sigma, h, t, C_1; \mathbb{C}_2 \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow} \quad \frac{\sigma, h, t, C_1 \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow}{\sigma, h, t, C_1 \parallel C_2 \xrightarrow{\text{loc}_{\text{LT}}}_{\varphi} \not\downarrow} \\
\\
\frac{\sigma, h, t, \mathbb{C} \xrightarrow{\text{loc}^*}_{\varphi} \not\downarrow}{\sigma, h, t, \langle \mathbb{C} \rangle \xrightarrow{\text{loc}_{\epsilon}}_{\varphi} \not\downarrow} \quad \frac{\sigma, h, t, C_2 \xrightarrow{\text{loc}_t}_{\varphi} \not\downarrow}{\sigma, h, t, C_1 \parallel C_2 \xrightarrow{\text{loc}_{\text{RT}}}_{\varphi} \not\downarrow} \quad \frac{c \in \text{PConf}}{c \xrightarrow{\text{env}}_{\varphi} \not\downarrow}
\end{array}$$

Fig. 22. The small-step operational semantics, failure cases

That is: a trace is fair if, at any point in time, every thread that can take a step (and the environment) will eventually be scheduled.

The open-world program semantics defines the behaviour of a command when run concurrently with an arbitrary environment. This semantics interleaves steps from two “players”: the local thread given by the loc relation; and its environment given by the env relation, respectively. It describes a finite trace obtained by the interleaving of local and environment steps starting from command \mathbb{C} and running the program to completion, and arbitrary environmental steps.

Definition D.12 (Open World Semantics). We call *traces* the infinite sequences of the form $c_0 \pi_0 c_1 \pi_1 \dots$ where, for all $i \in \mathbb{N}$, $c_i \in \text{Conf} \triangleq (\text{Store} \times \text{Heap}) \cup \{\not\downarrow\}$, $\pi_i \in \{\text{loc}, \text{env}\}$. We use τ for ranging over infinite suffixes of traces and Trace for the set of all traces. For a trace $\tau = c_0 \pi_0 c_1 \pi_1 \dots$, we define $\tau(i) \triangleq (c_i, \pi_i)$, and $\tau_{/i} \triangleq c_i \pi_i c_{i+1} \pi_{i+1} \dots$. The function $[\cdot]: \text{PTrace} \rightarrow \text{Trace}$ is defined by $[c_0 \pi_0 c_1 \pi_1 \dots] \triangleq c_0 \pi_0 c_1 \pi_1 \dots$ where

$$c_i \triangleq \begin{cases} (\sigma, h) & \text{if } c_i = (\sigma, h, _, _) \\ \not\downarrow & \text{if } c_i = \not\downarrow \end{cases} \quad \pi_i \triangleq \begin{cases} \text{loc} & \text{if } \pi_i \in \text{Sched} \setminus \{\text{env}\} \\ \text{env} & \text{if } \pi_i = \text{env} \end{cases}$$

The *open-world program semantics function*, $[\cdot]_{\varphi}: \text{Cmd} \rightarrow \wp(\text{Trace})$ is the function such that

$$[\mathbb{C}]_{\varphi} \triangleq \{ [c_0 \tau] \mid (c_0 \tau) \in \text{PTrace}_{\varphi}, \text{fv}(\mathbb{C}) \subseteq \text{dom}(\sigma_0), c_0 = (\sigma_0, _, _, \mathbb{C}), c_0 \tau \text{ is fair} \}$$

The notation $[\mathbb{C}]$ is syntactic sugar for $[\mathbb{C}]_{\emptyset}$.

Definition D.13. A trace $\tau \in \text{Trace}$ is *locally terminating*, written $\text{lterm}(\tau)$, if it contains finitely many occurrences of loc .

Remark 1 (Design of semantics). We made some design choices in crafting this semantics, with the motivation of making manipulation easier in the proofs. The first choice is to model environmental steps explicitly. These steps drive the argument about progress in the presence of blocking, where the local thread is not able to make progress in isolation but is relying on the environment actively performing some state changes that would lead to local progress.

The second choice we highlight is that the semantics of a program only contains infinite traces. This might seem odd when the goal is proving termination. Traces that locally terminate simply have an infinite tail of environment steps. To simulate a closed system one can select for the traces where the environment steps preserve the heaps. More importantly, we strip the information about threads and program state, which means that information about when the local thread terminated (in the form of \checkmark or end_t) has been erased. However, by construction, traces obtained from fair program traces can only contain finitely many local steps if the program terminated, justifying our definition of local termination.

Example D.14. The traces in $\llbracket [x] := y \rrbracket$ can be characterised as follows. They all start from some configuration (σ, h_0) with $x, y \in \text{dom}(\sigma)$. A (possibly zero) finite number of environment steps follow; these steps preserve the store, but arbitrarily alter the heap, or they lead to a fault, terminating the trace with an infinite tail of $\frac{1}{2} \text{env} \frac{1}{2} \text{env} \cdots$ steps. If no fault happened, a local step is taken from some configuration (σ, h) for an arbitrary $h \in \text{Heap}$. If $\sigma(x) \notin \text{dom}(h)$ then the local step leads to a fault, leading again to a $\frac{1}{2} \text{env} \frac{1}{2} \text{env} \cdots$ tail. Otherwise, it leads to the configuration $(\sigma, h[\sigma(x) \mapsto \sigma(y)])$. After that there is an infinite number of environment steps, which again preserve the store but arbitrarily mutate the heap, or lead to an infinite fault tail.

Remark 2 (Blocking primitives and strong fairness). The definition of traces, terminating traces and fairness are made somewhat simpler by the fact that every primitive of our language is not blocking, i.e. has at least one local successor in \longrightarrow_φ . For languages which have blocking primitives (e.g. built-in locks/channels) traces may be locally terminating because a configuration (different from \checkmark) may not have a local successor (i.e. it is not enabled) in any point in the future (e.g. if a built-in lock remains locked forever, an acquire operation would not have local successors). With blocking primitives, fairness also comes in two variants: strong and weak. Strong fairness requires that if an operation is infinitely often enabled it is infinitely often executed. Strong and weak fairness coincide for languages like ours where every primitive is enabled at all times.

Not considering blocking primitives does not make our setting less general: blocking primitives can be implemented on top of non-blocking ones, both with weak and strong fairness assumptions for termination, as illustrated by our spin and ticket lock examples. In other words, blocking primitives can be given TaDA Live specifications and be treated uniformly by the logics.

E ASSERTION LANGUAGE

We use a *set of logical variables*, denoted LVar , which is always assumed to be disjoint from the set of program variables, PVar .

Our logics is parametrised over a number of basic domains:

- An enumerable set of *region types* $\text{RType} \ni \mathbf{t}$. They are names that we will associate with ghost state information.
- An enumerable set of *region identifiers* $\text{RId} \ni r$.
- A set of *abstract states* $\text{AState} \ni a, a_1, \dots$. It may for example include sets and lists of values. Elements of this set are used to abstractly represent the state of some object using a mathematical structure.
- A set of *guards* $\text{Guard} \ni G$, which will offer the support for *guard algebras*, defined later.
- A well-founded partial order $(\mathcal{L}, \leq, \top, \perp)$ of *layers*, which will be associated to special guards called obligations. We will use the anti-reflexive restriction of the partial order: $k_1 < k_2 \triangleq k_1 \leq k_2 \wedge k_2 \not\leq k_1$.
- A set of ordinals \mathbb{O} .

The *set of abstract values* is $\text{AVal} \triangleq \text{Val} \cup \text{AState} \cup \text{Guard} \cup \text{RId} \cup \mathcal{L}$. The *set of logical expressions* is $\text{Exp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$ and the *set of logical Boolean expressions* is $\text{BExp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$.

We overload notation, writing $\mathbb{E} \in \text{Exp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$ and $\mathbb{B} \in \text{BExp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$. The expression evaluation function maps expressions d to the corresponding elements of ATrack : $\mathcal{E}[\![\diamond]\!]_{\zeta} = \diamond$, $\mathcal{E}[\![\diamond]\!]_{\zeta} = \diamond$, $\mathcal{E}[\![\mathbb{E}_1, \mathbb{E}_2]\!]_{\zeta} = (\mathcal{E}[\![\mathbb{E}_1]\!]_{\zeta}, \mathcal{E}[\![\mathbb{E}_2]\!]_{\zeta})$.

A *logical variable store*, $l \in \text{LStore} \triangleq \text{LVar} \rightarrow \text{AVal}$, is a function from logical variables to abstract values. The evaluation of logical expressions follows Definition D.5 with $\text{Vars} = \text{PVar} \uplus \text{LVar}$ and $\text{Values} = \text{AVal}$. We will also use the *set of levels* $\text{Lvl} \triangleq \mathbb{N}$ ranged over by λ .

TaDA Live assertions include ghost state built using guards which control the atomic update of a shared region, obligations which control what one can expect to eventually happen to a shared region, and atomicity tracking components which track the state of an open region that is in the process of being atomically updated.

Definition E.1 (Guard Algebras). A *guard algebra* is a PCM $(\text{Grd}, \bullet, \{0\})$ with $\text{Grd} \subseteq \text{Guard}$. TaDA Live is parametrised by a function $\mathcal{G}(\cdot)$ mapping a region type \mathbf{t} to a guard algebra $\mathcal{G}(\mathbf{t}) = (\mathcal{G}_{\mathbf{t}}, \bullet_{\mathbf{t}}, \{0_{\mathbf{t}}\})$. The \mathbf{t} subscript is omitted from $\bullet_{\mathbf{t}}$ and $0_{\mathbf{t}}$ when its is clear from the context.

Definition E.2 (Obligation Algebras). TaDA Live is parametrised by a *layered obligation structure*: that is, a pair $(\text{Oblig}, \text{lay})$ where $\text{Oblig} \subseteq \text{Guard}$ and $\text{lay}: \text{Oblig} \rightarrow \mathcal{L}$. such that $\forall O \in \text{Oblig}. \perp < \text{lay}(O) \leq \top$. A *obligation algebra* is a cancellative¹⁰ guard algebra $(\text{Obl}, \bullet, \{0\})$ where $\text{Obl} \subseteq \text{Oblig}$ and $\forall O_1, O_2 \in \text{Obl}. O_1 \sqsubseteq O_2 \Rightarrow \text{lay}(O_1) \geq \text{lay}(O_2)$. The set $\text{AOB} \subseteq \text{Oblig}$ is a subset of obligations that we call *atoms*. We require, for each obligation algebra Obl that $\text{AOB} \cap \text{Obl} = \{O \in \text{Obl} \mid \forall O_1, O_2 \in \text{Obl}. O \sqsubseteq O_1 \bullet O_2 \Rightarrow O \sqsubseteq O_1 \vee O \sqsubseteq O_2\}$.

TaDA Live is parametrised by a function $\mathcal{O}(\cdot)$ mapping a region type \mathbf{t} to an obligation algebra $\mathcal{O}(\mathbf{t}) = (\mathcal{O}_{\mathbf{t}}, \bullet_{\mathbf{t}}, \{0_{\mathbf{t}}\})$. The \mathbf{t} subscript is omitted from $\bullet_{\mathbf{t}}$ and $0_{\mathbf{t}}$ when its clear from the context.

In practice, obligation algebras are often constructed from some basic set of atoms (e.g. κ and \mathbf{d} of Fig. 4), to which we assign some layers, and then extend the layers to the compositions of atoms by taking the minimum layer of the composed atoms (e.g. since $\text{lay}(\kappa) < \text{lay}(\mathbf{d})$, we can set $\text{lay}(\kappa \bullet \mathbf{d}) = \text{lay}(\kappa)$).

Definition E.3 (TaDA Live Assertions). The set of TaDA Live assertions, $\text{Assrt} \ni P, Q, \dots$, is defined by the grammar:

P	$:=$	\mathbb{B}	
		$\exists x. P$	$x \in \text{LVar}$
		$\mathbb{E} \in X$	$X \subseteq \text{AVal}$
		$\neg P$	
		$P \wedge Q$	
		emp	
		$\mathbb{E} \mapsto \mathbb{E}$	
		$P * Q$	
		$P \multimap Q$	
		$\mathbf{t}_r^\lambda(\mathbb{E})$	$\mathbf{t} \in \text{RType}, \lambda \in \text{Lvl}$
		$r \Rightarrow d$	$d := \diamond \mid \diamond \mid (\mathbb{E}, \mathbb{E})$
		$[\mathbb{E}]_r$	
		$[\mathbb{E}]_r^L$	
		$[\mathbb{E}]_r^E$	
		emp_{Ob}^R	$R \subseteq \text{RId}$
		$r \triangleright m$	$m \in \mathcal{L}$

¹⁰cancellativity is a simplifying assumption; although it could be lifted, we found no evidence that non-cancellative obligation algebras could be needed in proofs.

where $\mathbb{B} \in \text{BExp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$, $\mathbb{E} \in \text{Exp}(\text{PVar} \uplus \text{LVar}, \text{AVal})$, and $r \in \text{RId} \cup \text{LVar}$.

The only binder is \exists . The function $\text{fv}: \text{Assrt} \rightarrow (\text{PVar} \uplus \text{LVar})$ returns the free variables of an assertion and its definition is standard. We also define $\text{pv}(P) \triangleq \text{fv}(P) \cap \text{PVar}$ and $\text{lv}(P) \triangleq \text{fv}(P) \cap \text{LVar}$. We write $P(x_1, \dots, x_n)$ to indicate that $\text{lv}(P) \subseteq \{x_1, \dots, x_n\}$, and for $v_1, \dots, v_n \in \text{AVal}$, $P(v_1, \dots, v_n)$ for $P[v_1/x_1, \dots, v_n/x_n]$.

The interpretation of the separation logics construct over (local) heaps is standard. We adopt here a classical interpretation of separation, where the elimination rule does not hold. The last three cases in the definition of assertions are TaDA-specific. They are used to represent shared abstract resources ($\text{t}_r^{\lambda}(\mathbb{E})$), represent ghost local resources ($\llbracket \mathbb{E} \rrbracket_r$) and keep track of atomicity of manipulation of abstract resources ($r \Rightarrow d$).

The TaDA Live *worlds* provide the models of TaDA Live's assertions. Worlds provide a *local* model in that they reflect the state as seen from the perspective of a single thread. A world is built from a local heap and a set of shared regions with guards, obligations and atomic tracking components describing how the shared regions are atomically updated.

Definition E.4 (Atomicity tracking Algebras). The *atomicity tracking algebra* is a PCM defined by $\text{ATrack} \triangleq ((\text{AState} \times \text{AState}) \uplus \{\diamond, \diamond\}, \cdot, \text{Emp}_{\diamond})$, where the composition is $\diamond \cdot \diamond = \diamond = \diamond \cdot \diamond$, $\diamond \cdot \diamond = \diamond$ and $\forall a, b \in \text{AState}$. $(a, b) \cdot (a, b) = (a, b)$ (undefined otherwise), and the set of unit elements is $\text{Emp}_{\diamond} \triangleq (\text{AState} \times \text{AState}) \uplus \{\diamond\}$.

Worlds are parametrised by a set of region identifiers \mathcal{R} which, intuitively, are the regions which the current operation is supposed to abstractly update exactly once.

Definition E.5 (Worlds). Let $\mathcal{R} \subseteq \text{RId}$. A *world*, $w \in \text{World}_{\mathcal{R}}$, is a tuple $w = (h, \rho, \gamma, \chi, \theta, \xi)$ where

- $h \in \text{Heap}$ is the local heap;
- $\rho \in \text{RMap} \triangleq \text{RId} \rightarrow_{\text{fin}} (\text{RType} \times \text{Lvl} \times \text{AState})$ describes the shared regions;
- $\gamma \in \text{GMap} \triangleq \text{RId} \rightarrow_{\text{fin}} \text{Guard}$ describes the local guards;
- $\chi \in \text{AMap}_{\mathcal{R}} \triangleq \mathcal{R} \rightarrow \text{ATrack}$ describes the local atomicity tracking components;
- $\theta \in \text{OMap} \triangleq \text{RId} \rightarrow_{\text{fin}} \text{Oblig}$ describes the local obligations;
- $\xi \in \text{OMap} \triangleq \text{RId} \rightarrow_{\text{fin}} \text{Oblig}$ describes the environment obligations, known to be held locally by the environment;

satisfying the following well-formedness constraints:

- $\text{dom}(\rho) = \text{dom}(\gamma) = \text{dom}(\theta) = \text{dom}(\xi) \supseteq \mathcal{R}$,
- $\forall r \in \text{RId}$. if $\rho(r) = (\mathbf{t}, _, _)$ then $\gamma(r) \in \mathcal{G}_{\mathbf{t}}$, $\theta(r) \in \mathcal{O}_{\mathbf{t}}$, $\xi(r) \in \mathcal{O}_{\mathbf{t}}$,
- $\forall r \in \text{dom}(\theta)$. $\theta(r) \# \xi(r)$.

A shared region with identity r , given by $\rho(r) = (\mathbf{t}, \lambda, a)$, has type \mathbf{t} , region identifier r and abstract state a . For a world w , we write h_w and ρ_w and so on for the corresponding components of w . We also define $\text{rty}_w(r) \triangleq \mathbf{t}$, $\text{lvl}_w(r) \triangleq \lambda$, and $\text{ast}_w(r) \triangleq a$, if $\rho_w(r) = (\mathbf{t}, \lambda, a)$.

We define the *world algebras* by first giving the composition for each of the world components. Heap composition is disjoint union. Shared regions only compose if they are equal. For $\rho \in \text{RMap}$, the compositions $\bullet_{\rho}: \text{GMap} \times \text{GMap} \rightarrow \text{GMap}$ and $\circ_{\mathcal{R}}: \text{AMap}_{\mathcal{R}} \times \text{AMap}_{\mathcal{R}} \rightarrow \text{AMap}_{\mathcal{R}}$ are:

$$\begin{aligned} \gamma_1 \bullet_{\rho} \gamma_2 &\triangleq \lambda r \in \text{dom}(\rho). \gamma_1(r) \bullet_{\mathbf{t}} \gamma_2(r) && \text{if } \forall r \in \text{dom}(\rho). \rho(r) = (\mathbf{t}, _, _) \wedge \gamma_1(r) \bullet_{\mathbf{t}} \gamma_2(r) \neq \perp \\ \chi_1 \circ_{\mathcal{R}} \chi_2 &\triangleq \lambda r \in \text{dom}(\rho). \chi_1(r) \cdot \chi_2(r) && \text{if } \forall r \in \mathcal{R}. \chi_1(r) \cdot \chi_2(r) \neq \perp \end{aligned}$$

and undefined otherwise. The composition \bullet_{ρ} on OMap is inherited from \bullet_{ρ} on GMap .

The composition of local and environment obligation maps compose in a subtle way inspired by the subjective separation of [19]. To express this interaction, we define a composition on *pairs* of

$$\begin{aligned}
& \zeta, w \vDash_{\mathcal{R}} \text{emp} \Leftrightarrow w \in \text{Emp}_{\mathcal{R}} \\
& \zeta, w \vDash_{\mathcal{R}} \mathbb{B} \Leftrightarrow \mathcal{B}[\mathbb{B}]_{\zeta} \\
& \zeta, (h, \rho, \gamma, \chi, \theta, \xi) \vDash_{\mathcal{R}} \mathbb{E}_1 \mapsto \mathbb{E}_2 \Leftrightarrow h = [\mathcal{E}[\mathbb{E}_1]_{\zeta} \mapsto \mathcal{E}[\mathbb{E}_2]_{\zeta}] \wedge (\theta, \rho, \gamma, \chi, \theta, \xi) \in \text{Emp}_{\mathcal{R}} \\
& \zeta, w \vDash_{\mathcal{R}} \neg P \Leftrightarrow \zeta, w \vDash_{\mathcal{R}} P \text{ does not hold} \\
& \zeta, w \vDash_{\mathcal{R}} \exists x. P \Leftrightarrow \exists v \in \text{AVal}. \zeta[x \mapsto v], w \vDash_{\mathcal{R}} P \\
& \zeta, w \vDash_{\mathcal{R}} \mathbb{E} \in X \Leftrightarrow \mathcal{E}[\mathbb{E}]_{\zeta} \in X \\
& \zeta, w \vDash_{\mathcal{R}} P_1 \wedge P_2 \Leftrightarrow (\zeta, w \vDash_{\mathcal{R}} P_1) \wedge (\zeta, w \vDash_{\mathcal{R}} P_2) \\
& \zeta, w_1 \odot w_2 \vDash_{\mathcal{R}} P_1 * P_2 \Leftrightarrow (\zeta, w_1 \vDash_{\mathcal{R}} P_1) \wedge (\zeta, w_2 \vDash_{\mathcal{R}} P_2) \\
& \zeta, w \vDash_{\mathcal{R}} P_1 * P_2 \Leftrightarrow \forall w' \in \text{World}_{\mathcal{R}}. (\zeta, w' \vDash_{\mathcal{R}} P_1) \wedge w \# w' \Rightarrow \zeta, w \cdot w' \vDash_{\mathcal{R}} P_2 \\
& \zeta, w \vDash_{\mathcal{R}} \mathbf{t}_r^{\lambda}(\mathbb{E}) \Leftrightarrow \rho_w(\mathcal{E}[\mathbb{E}]_{\zeta}) = (\mathbf{t}, \lambda, \mathcal{E}[\mathbb{E}]_{\zeta}) \wedge w \in \text{Emp}_{\mathcal{R}} \\
& \zeta, (h, \rho, \gamma[\mathcal{E}[r]_{\zeta} \mapsto \mathcal{E}[\mathbb{E}]_{\zeta}], \chi, \theta, \xi) \vDash_{\mathcal{R}} [\mathbb{E}]_r \Leftrightarrow (h, \rho, \gamma, \chi, \theta, \xi) \in \text{Emp}_{\mathcal{R}} \\
& \zeta, (h, \rho, \gamma, \chi[\mathcal{E}[r]_{\zeta} \mapsto v], \theta, \xi) \vDash_{\mathcal{R}} r \mapsto d \Leftrightarrow v = \mathcal{E}[d]_{\zeta} \wedge (h, \rho, \gamma, \chi, \theta, \xi) \in \text{Emp}_{\mathcal{R}} \\
& \zeta, (_, \rho, _, _, _, _) \vDash_{\mathcal{R}} \text{emp}_{\text{Ob}}^R \Leftrightarrow \forall r \in R. \rho(r) = (\mathbf{t}, _, _) \Rightarrow \theta(r) = \mathbf{0} \\
& \zeta, w \vDash_{\mathcal{R}} r \triangleright m \Leftrightarrow \text{lay}(\theta_w(\mathcal{E}[r]_{\zeta})) \geq m \\
& \zeta, (h, \rho, \gamma, \chi, \theta[\mathcal{E}[r]_{\zeta} \mapsto \mathcal{E}[\mathbb{E}]_{\zeta}], \xi) \vDash_{\mathcal{R}} [\mathbb{E}]_r^{\perp} \Leftrightarrow (h, \rho, \gamma, \chi, \theta, \xi) \in \text{Emp}_{\mathcal{R}} \\
& \zeta, w \vDash_{\mathcal{R}} [\mathbb{E}]_r^{\mathbb{E}} \Leftrightarrow \mathcal{E}[\mathbb{E}]_{\zeta} \sqsubseteq \xi_w(\mathcal{E}[r]_{\zeta}) \wedge w \in \text{Emp}_{\mathcal{R}}
\end{aligned}$$

Fig. 23. Definition of assertion satisfaction.

local/environment obligation maps. Given $\theta_1, \theta_2, \xi_1, \xi_2 \in \text{OMap}$, we define

$$(\theta_1, \xi_1) \odot_{\rho} (\theta_2, \xi_2) \triangleq \begin{cases} (\theta_1 \bullet_{\rho} \theta_2, \xi) & \text{if } \xi_1 = (\theta_2 \bullet_{\rho} \xi) \wedge \xi_2 = (\theta_1 \bullet_{\rho} \xi) \wedge (\theta_1 \bullet_{\rho} \theta_2) \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

This definition allows the splitting of local obligations into two resources, each knowing which obligations are held by the other.

Definition E.6 (World Algebras). The PCM of *world algebras*, $(\text{World}_{\mathcal{R}}, \odot, \text{Emp}_{\mathcal{R}})$, is defined by the set of worlds $\text{World}_{\mathcal{R}}$,

- the *subjective world composition*, \odot , given by:

$$(h_1, \rho_1, \gamma_1, \chi_1, \theta_1, \xi_1) \odot (h_2, \rho_2, \gamma_2, \chi_2, \theta_2, \xi_2) = (h_1 \uplus h_2, \rho, \gamma_1 \bullet_{\rho} \gamma_2, \chi_1 \circ_{\mathcal{R}} \chi_2, \theta, \xi)$$

if $h_1 \# h_2, \rho = \rho_1 = \rho_2, \gamma_1 \bullet_{\rho} \gamma_2 \neq \perp, \chi_1 \circ_{\mathcal{R}} \chi_2 \neq \perp$, and $(\theta_1, \xi_1) \odot_{\rho} (\theta_2, \xi_2) = (\theta, \xi)$, undefined otherwise; and

- the set of unit elements given by:

$$\text{Emp}_{\mathcal{R}} \triangleq \left\{ (\emptyset, \rho, \gamma, \chi, \theta, \xi) \in \text{World}_{\mathcal{R}} \mid \begin{array}{l} \forall r. \rho(r) = (\mathbf{t}, _, _) \Rightarrow \gamma(r) = \mathbf{0}_t \wedge \theta(r) = \mathbf{0}_t, \\ \forall r \in \mathcal{R}. \chi(r) \in \text{Emp}_{\spadesuit} \end{array} \right\}$$

Notice that the units are worlds with arbitrary shared regions, atomicity components from Emp_{\spadesuit} , and arbitrary environment obligations.

Definition E.7 (Satisfaction Relation). Let $\zeta : (\text{PVar} \uplus \text{LVar}) \rightarrow \text{AVal}$. For a world $w \in \text{World}_{\mathcal{R}}$ and an assertion P , the *assertion satisfaction* relation, $\zeta, w \vDash_{\mathcal{R}} P$, is defined in Fig. 23.

We write $\vdash_{\mathcal{R}} P$ if, for $\forall \zeta : (\text{PVar} \uplus \text{LVar}) \rightarrow \text{AVal}, w \in \text{World}_{\mathcal{R}}, \zeta, w \vDash_{\mathcal{R}} P$, and write $\mathcal{W}[[P]]_{\mathcal{R}}^{\zeta} \triangleq \{ w \mid \zeta, w \vDash_{\mathcal{R}} P \}$ for any assertion P .

E.1 Trace Semantics for Specifications

The type of each region is associated with a *region interference function* which establishes which updates to a shared region are allowed to the owner of which guards.

Definition E.8 (Region Interference). TaDA Live is parametrised by the *region interference function*, \mathcal{T} , which takes a region type $\mathbf{t} \in \text{RType}$ and returns a function $\mathcal{T}_{\mathbf{t}}: \mathcal{G}_{\mathbf{t}} \rightarrow \wp((\text{AState} \times \mathcal{O}_{\mathbf{t}}) \times (\text{AState} \times \mathcal{O}_{\mathbf{t}}))$. Every function $\mathcal{T}_{\mathbf{t}}$ is required to satisfy three properties:

- monotonicity in the guards: $\forall G_1, G_2 \in \mathcal{G}_{\mathbf{t}}. G_1 \sqsubseteq G_2 \Rightarrow \mathcal{T}_{\mathbf{t}}(G_1) \subseteq \mathcal{T}_{\mathbf{t}}(G_2)$;
- reflexivity: $((a, \mathbf{0}_{\mathbf{t}}), (a, \mathbf{0}_{\mathbf{t}})) \in \mathcal{T}_{\mathbf{t}}(\mathbf{0}_{\mathbf{t}})$, for all $a \in \text{AState}$;
- closure under obligation frames: for all $O_1, O_2, O \in \mathcal{O}_{\mathbf{t}}$, if $((a_1, O_1), (a_2, O_2)) \in \mathcal{T}_{\mathbf{t}}(G)$ and $O_1 \# O$ and $O_2 \# O$, then $((a_1, O_1 \bullet_{\mathbf{t}} O), (a_2, O_2 \bullet_{\mathbf{t}} O)) \in \mathcal{T}_{\mathbf{t}}(G)$.

We write $\mathcal{T}_{\mathbf{t}}(_)$ for $\bigcup_{G \in \mathcal{G}_{\mathbf{t}}} \mathcal{T}_{\mathbf{t}}(G)$. For any $R \subseteq (\text{AState} \times \text{Oblig}) \times (\text{AState} \times \text{Oblig})$ we write $\text{io}(R) \triangleq \{(a, b) \mid ((a, _), (b, _)) \in R\}$.

In TaDA, we prove abstract atomicity using the **MkATOM** rule, which converts a Hoare triple to an atomic triple, provided the Hoare triple bears evidence that, although many steps might have been taken, the abstract state was changed by the command exactly once. The atomic triple may be constraining the environment interference with a non-trivial pseudoquantifier, and we need make this assumption on the environment available to the proof of the Hoare triple. For this reason, TaDA Live judgments have a so-called *atomicity context* component \mathcal{A} that records exactly the atomicity and interference information of outer proof goals.

Definition E.9 (Atomicity Context). An *atomicity context* \mathcal{A} is a finite partial function from RId to tuples of the form (X, k, X', R) where $X, X' \subseteq \text{AState}$, $k \in \mathcal{L}$, and $R \subseteq (\text{AState} \times \text{Oblig}) \times (\text{AState} \times \text{Oblig})$ is closed under obligation frames (as in Definition E.8).

Assuming $\mathcal{A}(r) = (X, k, X', R)$, we write $\text{safe}(\mathcal{A}, r) \triangleq X$, $\text{good}(\mathcal{A}, r) \triangleq X'$, $\text{live}(\mathcal{A}, r) \triangleq (X, k, X')$ which we write $X \rightarrow_k X'$, and $\text{tr}(\mathcal{A}, r) \triangleq R$. For every $r \in \text{dom}(\mathcal{A})$, we require $\{x \mid (x, _) \in \text{io}(R)\} \subseteq \text{safe}(\mathcal{A}, r)$. The set $\text{dom}(\mathcal{A})$ declares the regions for which we are tracking atomicity: for $r \in \text{dom}(\mathcal{A})$, the environment will only change the abstract state within $\text{safe}(\mathcal{A}, r)$ and will obey the liveness condition given by $\text{live}(\mathcal{A}, r)$ that the environment will always eventually return a good state in $\text{good}(\mathcal{A}, r) \triangleq X'$; and the local thread will only change the abstract state at most once according to the relation $\text{io}(\text{tr}(\mathcal{A}, r))$.

A world describes the state of the current thread, both the local state owned by the thread (the heap, guards, local obligations and atomicity tracking components), the shared state (the regions) and the environment obligations describing obligations owned locally by the environment. We define the *world rely relation* which describes how the world may change as a result of the “well-behaved” interference of the environment characterised by the region interference relations, the atomicity tracking components and the environment obligations.

Definition E.10 (World Rely). The *world rely relation*, $\mathbf{R}_{\mathcal{A}} \subseteq \text{World}_{\mathcal{A}} \times \text{World}_{\mathcal{A}}$, is the smallest reflexive and transitive relation satisfying the rules in Fig. 24.

Rule **wR₁** describes the case where the environment can update the abstract state of a region according to the interference relation $\mathcal{T}_{\mathbf{t}}$. Notice that, for this rule, when $\chi(r) \in \{\diamond, \diamond\}$, the environment can only change the abstract state to something in $\text{safe}(\mathcal{A}, r)$. When $\chi(r)$ is undefined or a pair of abstract states, then the environment does not have this restriction and can do any update consistent with $\mathcal{T}_{\mathbf{t}}$. Also, notice how the environment obligations map ξ is affected by the transition. Rule **wR₂** describes the case where the atomic update given by \mathcal{A} has been delegated to the environment ($\chi[r \mapsto \diamond]$) in which case the current thread observes the abstract state change

$$\begin{array}{c}
\frac{\gamma(r) \# G \quad ((a_1, O_1), (a_2, O_2)) \in \mathcal{T}_t(G) \quad \chi(r) \in \{\blacklozenge, \diamond\} \Rightarrow a_2 \in \text{safe}(\mathcal{A}, r) \quad O_2 \# \theta(r)}{(h, \rho[r \mapsto (t, \lambda, a_1)], \gamma, \chi, \theta, \xi[r \mapsto O_1]) \mathbf{R}_{\mathcal{A}} (h, \rho[r \mapsto (t, \lambda, a_2)], \gamma, \chi, \theta, \xi[r \mapsto O_2])} \text{WR}_1 \\
\\
\frac{\gamma(r) \# G \quad ((a_1, O_1), (a_2, O_2)) \in \text{tr}(\mathcal{A}, r) \quad O_2 \# \theta(r)}{(h, \rho[r \mapsto (t, \lambda, a_1)], \gamma, \chi[r \mapsto \diamond], \theta, \xi[r \mapsto O_1]) \mathbf{R}_{\mathcal{A}} (h, \rho[r \mapsto (t, \lambda, a_2)], \gamma, \chi[r \mapsto (a_1, a_2)], \theta, \xi[r \mapsto O_2])} \text{WR}_2 \\
\\
\frac{r \notin \text{dom}(\rho) \quad t \in \text{RType} \quad \lambda \in \mathbb{N} \quad a \in \text{AState} \quad O \in \mathcal{O}_t}{(h, \rho, \gamma, \chi, \theta, \xi) \mathbf{R}_{\mathcal{A}} (h, \rho[r \mapsto (t, \lambda, a)], \gamma[r \mapsto \mathbf{0}_t], \chi, \theta[r \mapsto \mathbf{0}_t], \xi[r \mapsto O])} \text{WR}_3 \\
\\
\frac{\forall r \in \text{dom}(\xi). \xi(r) \sqsubseteq \xi'(r) \wedge \theta(r) \# \xi'(r)}{(h, \rho, \gamma, \chi, \theta, \xi) \mathbf{R}_{\mathcal{A}} (h, \rho, \gamma, \chi, \theta, \xi')} \text{WR}_4
\end{array}$$

Fig. 24. World Rely rules

corresponding to the update. Rule WR_3 allows the environment to create arbitrary new shared regions. Note how this cannot assign non-trivial local obligations to the current thread. Rule WR_4 formalises that $\xi(r) = O$ does not imply that the environment has *exactly* the obligation O , but *at least* O .

We write $\vDash_{\mathcal{A}}$ for $\vDash_{\text{dom}(\mathcal{A})}$, and similarly for $\vdash_{\mathcal{A}}$, $\mathcal{W}[[P]]_{\mathcal{A}}^{\zeta}$, $\text{World}_{\mathcal{A}}$ and $\text{Emp}_{\mathcal{A}}$.

So far, we have introduced assertions, and worlds as their models. These structures express information mostly over *ghost state*, that is, state that is purely logical and has no representation in concrete executions. For example, the notion that there is some shared region is purely fictional, as in the concrete machine there is no special way to mark a portion of the heap as shared. We introduced interference protocols and the world rely, as a way to specify the expected well-behaved transformations shared resources may be subjected to. Since well-behaved interference from the environment can change the state of shared regions, a single world (describing a single state for each region) cannot capture the logical state we may be in, when interleaved with environment actions. Views are the sets of worlds that can explain the logical state we may be in after being suspended for an arbitrary number of environment steps. Views represent information about the logical state that cannot be invalidated by a well-behaved environment.

Definition E.11 (Views, Stability). A set of worlds $p \subseteq \text{World}_{\mathcal{A}}$ is an \mathcal{A} -view if it is closed under $\mathbf{R}_{\mathcal{A}}$: that is, $\forall w \in p, w' \in \text{World}_{\mathcal{A}}. w \mathbf{R}_{\mathcal{A}} w' \Rightarrow w' \in p$. An assertion P is \mathcal{A} -stable, written $\mathcal{A} \vDash P$ stable, if and only if, for all $\zeta : (\text{PVar} \uplus \text{LVar}) \rightarrow \text{AVal}$, $\mathcal{W}[[P]]_{\mathcal{A}}^{\zeta}$ is a \mathcal{A} -view.

We write $\text{View}_{\mathcal{A}}$ for the set of all \mathcal{A} -views and $\text{Stable}_{\mathcal{A}}$ for the set of all \mathcal{A} -stable assertions.

Definition E.12 (View Algebra). The PCM of *view algebras*, $(\text{View}_{\mathcal{A}}, *, \text{emp}_{\mathcal{A}})$, is generated by the set $\text{View}_{\mathcal{A}}$, the composition $p_1 * p_2 \triangleq \{w_1 \odot w_2 \mid w_1 \in p_1, w_2 \in p_2, w_1 \# w_2\}$ and the unit $\text{emp}_{\mathcal{A}} \triangleq \{\text{Emp}_{\mathcal{A}}\}$.

Notice that the composition of views always gives rise to a view: in the case where there are no compatible pairs of worlds exists in the views, the result is the empty view.

As we mentioned, worlds and views represent ghost information about state. Ultimately, however, we want to use this information to express properties of concrete execution traces. To do so, we need to formalise the link between the logical information in worlds and the concrete state of the store/heap. The first component that contributes to this link is *region interpretations*, which specify what is the (implementation dependent) shared resource’s content. For example, for a shared spin lock, we want to specify that the abstract shared region $\text{spin}_r^\lambda(x, l)$ is implemented as a single cell storing l at x , $x \mapsto l$.

Definition E.13 (Region Interpretation). TaDA Live is parametrised by a *region interpretation function* $\mathcal{I}_t[\cdot]: \text{RId} \times \text{Lvl} \times \text{AState} \rightarrow \text{View}_0$ for each $t \in \text{RType}$, such that, for every $r \in \text{RId}$, $\lambda \in \text{Lvl}$, $a \in \text{AState}$, $\forall w \in \mathcal{I}_t[r, \lambda, a]$, $\forall r' \in \text{dom}(\theta_w) \setminus \{r\}$, $\theta_w(r') = \mathbf{0}$. Its companion is the *syntactic region interpretation* $\mathcal{I}_t = (r, l, a, P)$ where $r, l, a \in \text{LVar}$, $\text{fv}(P) \subseteq \{r, l, a\}$, $\mathbf{0} \models P$ stable, and $\vdash_0 P[\lambda/l] \Rightarrow \text{emp}_{\text{Ob}}^{\text{RId} \setminus \{r\}}$. We write $\mathcal{I}(t_{\mathbb{E}_1}^\lambda(\mathbb{E}_2))$ for $P[\mathbb{E}_1/r, \lambda/l, \mathbb{E}_2/a]$. We require that $\mathcal{I}_t[r, \lambda, a] = \mathcal{W}_0[\mathcal{I}(t_r^\lambda(a))]$; in practice, we will define region interpretations by writing syntactic interpretations and using the previous equation as a definition for the corresponding region interpretation functions.

It is important to understand that interpretations are not merely an indirect way of writing assertions. In our spin lock example, the crucial difference between the two assertions $\text{spin}_r^\lambda(x, l)$ and $x \mapsto l$, is that the first is subjected to interference, while the latter expresses ownership of the cell at x .

As in TaDA, the region interpretation is used to ‘open’ a region: that is, import the region interpretation as local state in order to do a single atomic update. The idea is to obtain intantaneously the ownership of the content of the region for the atomic update, and to re-establish the region interpretation for the updated abstract state, before immediately relinquishing ownership by ‘closing’ up the region. As in TaDA this opening and closing mechanism depends on the level of the region, which is a device to avoid inconsistencies since interpretations can mention other regions in a nested fashion. With a specification at level λ , the rules enable a region to be opened at level $\lambda' < \lambda$ to obtain a resulting specification at level λ' . This means that, although region can be shared, $(\vdash t_r^{\lambda'}(a) \Leftrightarrow t_r^{\lambda'}(a) * t_r^{\lambda'}(a))$ they cannot be *opened* twice, thus avoiding the potential clashing duplication of local state in the region interpretation.

The final component that expresses the link between the logical information and concrete state, is the *reification* function. Intuitively, the reification $[w]_\lambda$ of a world w , opens all closed regions (according to λ) and computes the heap consisting of the local heaplet, composed with the heaplet of all the concrete contents of shared regions. Overall, a world w is conceptually linked to a global heap h in a step of a trace, if $h = h_1 \uplus h_2$ and h_1 is in the reification of w ; here h_2 represents the heaplet that is local to other threads.

Definition E.14 (World and View Reification). Let $\lambda \in \text{Lvl}$ and let $\text{closed}(\lambda, w) \triangleq \{r \in \text{RId} \mid \text{lvl}_w(r) < \lambda\}$. The *region collapse function*, $(\cdot) \downarrow_\lambda: \text{World}_{\mathcal{A}} \rightarrow \text{World}_{\mathcal{A}}$, is defined by:

$$w \downarrow_\lambda \triangleq \left\{ w_0 \odot w_1 \odot \dots \odot w_n \left| \begin{array}{l} \text{closed}(\lambda, w_0) = \{r_1, \dots, r_n\}, \rho_{w_0}(r_i) = (t_i, \lambda_i, a_i), w_i \in \mathcal{I}_{t_i}[r_i, \lambda_i, a_i], \\ \forall i \leq n. \forall r \in \text{dom}(\xi_{w_0 \odot \dots \odot w_i}). \xi_{w_0 \odot \dots \odot w_i}(r) = \mathbf{0} \end{array} \right. \right\}$$

The function $[w]_\lambda \triangleq \{h \in \text{Heap} \mid (h, _ , _ , _ , _) \in w \downarrow_\lambda\}$ is called the *world reification* of w at level λ . For any view $p \subseteq \text{World}_{\mathcal{A}}$, the function $\llbracket p \rrbracket_\lambda \triangleq \bigcup_{w \in p} [w]_\lambda$ is called *view reification* of p at level λ .

The reification only considers the collapsed worlds where no environment obligation can be assumed in the environment. This is because we are collapsing to obtain the global (local plus shared) concrete state, not one that we can hypothesise can be completed with a world from the environment with the appropriate obligations.

Having established the link between worlds/views and concrete state, we can move to establishing a link between concrete *steps* in a trace, and their logical justification in terms of logical state. The fundamental driver of this link is the notion of *frame-preserving update*, inspired by the frame-preserving update from [15], which represents the essence of the Rely/Guarantee reasoning in TaDA Live. The frame-preserving update looks at a specific concrete update from some h to h' , and checks whether the update can be logically described as an update from logical state p to logical state q (both sets of worlds): this is the case when all the views that are compatible with p are compatible with q . A local step in a concrete trace can be justified as going from p to q if

all the frames of p that are valid views at the source of the update, are going to be valid at the target of the update. Since the environment is expected to be representing the logical state with a view, the local step will not invalidate the environment’s beliefs about the logical state of the system (the guarantee implies the rely). Conversely, if the frame-preserving update corresponds to a concrete step of the environment, no view we may be holding locally can be invalidated by it (the environment satisfies the rely).

Definition E.15 (Frame-Preserving Update). Given $h_1, h_2 \in \text{Heap}$, $p_1, p_2 \subseteq \text{World}_{\mathcal{A}}$ and $\lambda_1, \lambda_2 \in \text{Lvl}$, define

$$(h_1, h_2) \vDash_{\mathcal{A}} p_1 : \lambda_1 \rightarrow^* q_2 : \lambda_2 \quad \text{if and only if} \quad \forall f \in \text{View}_{\mathcal{A}}. h_1 \in \llbracket p_1 * f \rrbracket_{\lambda_1} \Rightarrow h_2 \in \llbracket q_2 * f \rrbracket_{\lambda_2}.$$

We write $(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_1 \rightarrow^* q_2$ for $(h_1, h_2) \vDash_{\mathcal{A}} p_1 : \lambda \rightarrow^* q_2 : \lambda$.

Concretely, the frame-preserving update definition lifts the world rely to a rely on steps as follows. To see how this works, let us consider an example and define $\vDash_{\lambda} p \rightarrow^* q$ to mean $\forall h \in \llbracket p * \text{True} \rrbracket. \exists h'. (h, h') \vDash_{\lambda} p \rightarrow^* q$, that is, $\vDash_{\lambda} p \rightarrow^* q$ holds when p to q can be used to justify some concrete update.

Example E.16. Assume we have a region type \mathbf{t} with abstract states a, b, c, d , a single guard \mathbf{E} (with $\mathbf{E} \bullet \mathbf{E} = \perp$) and interference protocol consisting of transitions $\mathbf{E} : (a, \mathbf{0}) \rightsquigarrow (b, \mathbf{0})$ and $\mathbf{E} : (b, \mathbf{0}) \rightsquigarrow (c, \mathbf{0})$. We want to show that (for $\lambda < \lambda'$) $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) * \llbracket \mathbf{E} \rrbracket_r \rightarrow^* \mathbf{t}_r^{\lambda}(c) * \llbracket \mathbf{E} \rrbracket_r$ holds, but $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) * \llbracket \mathbf{E} \rrbracket_r \rightarrow^* \mathbf{t}_r^{\lambda}(d) * \llbracket \mathbf{E} \rrbracket_r$ and $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) \rightarrow^* \mathbf{t}_r^{\lambda}(b)$ do not. Consider any view f that is a frame of $\mathbf{t}_r^{\lambda}(a) * \llbracket \mathbf{E} \rrbracket_r$: f cannot hold $\llbracket \mathbf{E} \rrbracket_r$ because \mathbf{E} is not compatible with itself. As a consequence, since f is view, it needs to be closed under world rely, which means that it is closed under the interference, which can transform a into b and b into c . For f to be compatible with $\mathbf{t}_r^{\lambda}(a)$, it needs to contain some world associating a to r ; to be a view, f needs to contain some other world associating c to r , which makes it compatible with $\mathbf{t}_r^{\lambda}(c) * \llbracket \mathbf{E} \rrbracket_r$. Therefore $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) * \llbracket \mathbf{E} \rrbracket_r \rightarrow^* \mathbf{t}_r^{\lambda}(c) * \llbracket \mathbf{E} \rrbracket_r$ holds.

Now, the view f above is not required to contain any world associating d to r . Such an f is a counterexample to $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) * \llbracket \mathbf{E} \rrbracket_r \rightarrow^* \mathbf{t}_r^{\lambda}(d) * \llbracket \mathbf{E} \rrbracket_r$ holding.

Finally, consider $\mathbf{t}_r^{\lambda}(a)$; we can construct a frame f_a in which all worlds associate a to r and own the guard \mathbf{E} . Such set of worlds can be a view because owning \mathbf{E} disables the transition from a to b . However, f_a would be compatible with $\mathbf{t}_r^{\lambda}(a)$ but not with $\mathbf{t}_r^{\lambda}(b)$, which means $\vDash_{\lambda'} \mathbf{t}_r^{\lambda}(a) \rightarrow^* \mathbf{t}_r^{\lambda}(b)$ does not hold.

Using this definition of frame-preserving update, we have been able to simplify drastically the semantics of TaDA specifications. For TaDA Live, however, we need to introduce the stronger notion of *atomic* frame-preserving update. To see the motivation behind the stronger condition, consider the region interference relation $\mathbf{E} : (a, \mathbf{k}) \rightsquigarrow (b, \mathbf{0})$ and $\mathbf{E} : (b, \mathbf{0}) \rightsquigarrow (c, \mathbf{k})$. The update from a to c via b is very different from a direct update from a to c . The intermediate step to b fulfils the obligation \mathbf{k} , which may be a crucial information for the progress argument. We therefore want to enforce that if we are justifying a step as going from p to q , all the allowed transitions between region states need to match a *single transition* in the interference protocol.

Definition E.17 (Atomic Frame-Preserving Update). Given $h_1, h_2 \in \text{Heap}$, $p_1, p_2 \subseteq \text{World}_{\mathcal{A}}$, and $\lambda_1, \lambda_2 \in \text{Lvl}$, define $(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_1 : \lambda_1 \rightarrow p_2 : \lambda_2$ if and only if $(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_1 : \lambda_1 \rightarrow^* p_2 : \lambda_2$ holds and, for all $\mathbf{t} \in \text{RType}$, $r \in \text{RId}$, $\lambda' \in \text{Lvl}$, $a_1 \in \text{AState}$, $O_1 \in O_{\mathbf{t}}$:

$$\forall f \in \text{View}_{\mathcal{A}}. h_1 \in \llbracket (p_1 * f) \cap W_1 \rrbracket_{\lambda_1} \Rightarrow \exists a_2, O_2. ((a_1, O_1), (a_2, O_2)) \in \mathcal{T}_{\mathbf{t}} \wedge h_2 \in \llbracket (p_2 * f) \cap W_2 \rrbracket_{\lambda_2} \quad (14)$$

where $W_i \triangleq \{w \in \text{World}_{\mathcal{A}} \mid \rho_w(r) = (\mathbf{t}, \lambda', a_i), \theta_w(r) = O_i\}$. Since most of the times we will consider updates at constant level (i.e. $\lambda_1 = \lambda_2$) we write $(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_1 \rightarrow q_2$ for $(h_1, h_2) \vDash_{\mathcal{A}} p_1 : \lambda \rightarrow q_2 : \lambda$.

Definition E.17 is in a form that is easier to manipulate for the proofs, but may seem ad-hoc at first sight. There is an alternative definition which clarifies how this is a generalisation of the concept of frame-preserving update. Define the *one-step world rely relation*, $\mathbf{R}_{\mathcal{A}}^1$, to be the smallest reflexive relation closed under the rules of Fig. 24, with the restriction that rules \mathbf{WR}_1 and \mathbf{WR}_2 can be applied at most once per region identifier. Then, $(h_1, h_2) \vDash_{\lambda, \mathcal{A}} p_1 : \lambda_1 \rightarrow p_2 : \lambda_2$ if and only if

$$\forall f \subseteq \text{World}_{\mathcal{A}}. h_1 \in \llbracket p_1 * f \rrbracket_{\lambda_1} \Rightarrow h_2 \in \llbracket p_2 * \mathbf{R}_{\mathcal{A}}^1(f) \rrbracket_{\lambda_2}$$

Intuitively, this says that if the environment has some resource f compatible with p , it should expect that after a step, the resource f might be transformed into $\mathbf{R}_{\mathcal{A}}^1(f)$. When f is a view, one gets back Definition E.15, as views are precisely the resources that cannot be invalidated by any number of updates of the environment.

We will use atomic frame-preserving updates to check safety of traces (according to some specification) in Definition E.23.

Before moving to specifications, we briefly define *view shift*, a semantic generalisation of implication, which is a prime example of application of frame-preserving update, used in our **CONS** rule.

Definition E.18 (View Shift). Given $p_1, p_2 \subseteq \text{World}_{\mathcal{A}}$, the judgment $\mathcal{A} \vDash p_1 \stackrel{\lambda_1 \Rightarrow \lambda_2}{\rightsquigarrow} p_2$, holds if $\forall h \in \text{Heap}. (h, h) \vDash_{\mathcal{A}} p_1 : \lambda_1 \rightarrow p_2 : \lambda_2$. For two assertions P, Q , the assertion P *viewshifts* to Q , written $\mathcal{A} \vDash P \stackrel{\lambda_1 \Rightarrow \lambda_2}{\rightsquigarrow} Q$, if and only if, $\forall \zeta : (\text{PVar} \uplus \text{LVar}) \rightarrow \text{AVal}, \mathcal{A} \vDash \mathcal{W} \llbracket P \rrbracket_{\mathcal{A}}^{\zeta} \stackrel{\lambda_1 \Rightarrow \lambda_2}{\rightsquigarrow} \mathcal{W} \llbracket Q \rrbracket_{\mathcal{A}}^{\zeta}$. Since it is very common to consider viewshifts at constant level (i.e. $\lambda_1 = \lambda_2$) we write $\lambda; \mathcal{A} \vDash p \Rightarrow q$ for $\mathcal{A} \vDash p \stackrel{\lambda \Rightarrow \lambda}{\rightsquigarrow} q$ (and similarly for the lift to assertions).

View shifts are typically employed to “allocate” a new region by sharing some local resource (a form of weakening). For example, assume $I(\mathbf{t}_r(x, v)) \triangleq x \mapsto v$. We have that $x \mapsto v$ viewshifts to $\exists r. \mathbf{t}_r(x, v)$: the underlying reification does not change, and any frame of $x \mapsto v$ with non-empty reification, must only have regions reifying to cells disjoint from x ; moreover, such frame will only have finitely many regions allocated, so it is always possible to draw a fresh one from the infinite set Rld to satisfy the existential quantification over r .

With all these definitions in place, we can now proceed in formally defining TaDA Live specifications in their general form, and give them trace semantics.

In our examples and in the simplified rules of Fig. 6, we always only used triples of two forms:

$$m, \lambda, \mathcal{A} \vdash \forall x \in X \rightarrow_k X'. \langle P(x) \rangle \mathbb{C} \langle Q(x) \rangle \quad (15a)$$

$$m, \lambda, \mathcal{A} \vdash \{P\} \mathbb{C} \{Q\} \quad (15b)$$

which we call *pure atomic* and *Hoare*, respectively. In the general case, a command may manipulate some resources P_h non-atomically, and some other resources $P_a(x)$ atomically, at the same time. The specifications in their general form are therefore an hybrid of pure atomic and Hoare triples:

$$m; \lambda; \mathcal{A} \vDash \forall x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \mathbb{C} \langle Q_h(x) \mid Q_a(x) \rangle$$

Intuitively, the Hoare precondition P_h is a resource that is owned by the command and, as such, cannot be invalidated by actions of the environment. The command is allowed to manipulate this owned resource non-atomically, provided it satisfies the Hoare postcondition Q_h upon termination. The atomic precondition $P_a(x)$ represents the resource that can be shared between the command and the environment. The environment can update it, but only with the effect of going from $P_a(x)$ for some $x \in X$ to $P_a(x')$ for some $x' \in X$. The command is allowed to update it exactly once from $P_a(x)$ to perform its linearisation point, transforming it to a resource satisfying the atomic postcondition $Q_a(x)$. The atomic postcondition only needs to be true *just after* the linearisation point as the environment is allowed to update it immediately afterwards. The pseudo-quantified variable x

has two important uses: it represents the “surface” of allowed interference by the environment; it is bound in the postcondition to the value of the parameter of the atomic precondition *just before* the linearisation point.

The pure and Hoare triples in (15b) and (15a) are then special cases of the hybrid triple:

$$m; \lambda; \mathcal{A} \vdash \langle P \mid \text{emp} \rangle \mathbb{C} \langle Q \mid \text{emp} \rangle \quad (16a)$$

$$\forall \vec{v}_0. m; \lambda; \mathcal{A} \vdash \mathbb{W}x \in X \rightarrow_k X'. \langle \vec{v}_0 \doteq \vec{v}_0 \mid P'(x) \rangle \mathbb{C} \exists \vec{v}_1. \langle \vec{v}_0 \doteq \vec{v}_0 \wedge \vec{v}_1 \doteq \vec{v}_1 \mid Q'(x) \rangle \quad (16b)$$

respectively, where $\vec{v}_0 = \text{pv}(P(x))$, $\vec{v}_1 = \text{pv}(Q(x)) \setminus \vec{v}_0$, $P'(x) = P(x)[\vec{v}_0/\vec{v}_0]$ and $Q'(x) = Q(x)[\vec{v}_0/\vec{v}_0, \vec{v}_1/\vec{v}_1]$ (for technical reasons the atomic pre/post-conditions in the general triples cannot contain program variables). We omit the pseudo-quantifier form an atomic triple (as above) when the pseudo-quantified variable does not occur in the triple, and thus could be trivially quantified as $\mathbb{W}x \in \text{AVal} \rightarrow_{\perp} \text{AVal}$. In Section 3 we also used the abbreviated form $\mathbb{W}x \in X$ when the liveness assumption is trivial, i.e. $\mathbb{W}x \in X \rightarrow_k X$.

Definition E.19 (Specification). The set of specifications, $\text{Spec} \ni \mathbb{S}, \mathbb{S}_1, \mathbb{S}_2, \dots$, have the form:¹¹

$$\mathbb{W}x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{m; \lambda; \mathcal{A}} \quad (17)$$

where

- $m \in \mathcal{L}$, $\lambda \in \text{Lvl}$ and $\mathcal{A} \in \text{ACTxt}$;
- $x, y \in \text{LVar}$;
- $X' \subseteq X \subseteq \text{AVal}$ and $k \in \mathcal{L}$;
- $P_h, Q_h(v, v') \in \text{Stable}_{\mathcal{A}}$ for all $v \in X$ and $v' \in \text{AVal}$;
- $P_a(v), Q_a(v, v') \in \text{Assrt}$ for all $v \in X$ and $v' \in \text{AVal}$, and $\text{pv}(P_a) = \text{pv}(Q_a) = \emptyset$.

A function specification context provide the arguments of the function and the specification of the function satisfied by the function body.

Definition E.20 (Function Specification Context). A function specification context, Φ , is a partial function $\Phi \in \text{FSpec} \triangleq \text{FName} \rightarrow (\text{PVar}^*, \text{Spec})$.

Finally, we can define the semantics of a specification. Intuitively, a specification like (17) requires the starting state to satisfy $P_h * P_a(x)$, for some $x \in X$, and the environment to interfere by changing x between values in X ; the liveness assumption requires that the environment should always eventually set x to some value in X' during the interference phase. In return, the specification guarantees that:

- (1) The resources in P_h will be (non-atomically) modified and upon termination of the command, will satisfy Q_h .
- (2) There is a single linearisation point going from $P_a(x)$ to $Q_a(x, y)$, for any $x \in X$ and some y .
- (3) The state at the end of the execution will satisfy $Q_h(x, y)$.

In this sense, the resources in P_a should be understood as shared: the environment can use them to change the value of x , and the local command will be able to use them atomically to perform its linearisation point. Note that $Q_a(x, y)$ is only guaranteed to hold immediately after the linearisation point. The specifications mention two layers: m and k . Layer m represents a (strict) upper bound on the layers that we may consider live when proving some command satisfies the specification. Layer k decorates the liveness assumption; it will restrict the layers one can be depending on, when using the liveness assumption in a proof. For example, it will not be possible to invoke the liveness assumption while continuously holding an obligation of layer $k' \leq k$.

¹¹The $\exists y$ is needed in the (uncommon) case when the linearisation point is non-deterministic *and* the Hoare postcondition depends on this non-deterministic choice.

We now define the formal semantics of specifications, as set of concrete traces that satisfy the specification. To check if a concrete trace τ satisfies a specification, the semantics first collects all the possible “logical” justifications of the trace in a set \mathbb{T} . To justify a trace means to instrument each step with views that show how the trace respects the (safety) logical constraints of the specification. The set \mathbb{T} is then further analysed to check that every instrumented trace where the environment satisfies the liveness assumptions is locally terminating.

Specification traces are traces instrumented with sets of worlds representing the logical justification for each step. For each state in a trace, the instrumentation consists of a view p_h representing the local Hoare view, a set of worlds $p_a(x)$ parametric on a value, representing the atomic resource, and a value v or $\langle v, v' \rangle$ that either represents the current choice for x in $p_a(x)$, if before the linearisation point, or records the linearisation point from v to v' that took place in some previous step.

Definition E.21 (Specification Traces). Define $\text{AVal}' \triangleq \text{AVal} \uplus \{\langle v_1, v_2 \rangle \mid v_1, v_2 \in \text{AVal}\}$, the set of *specification states* to be $\text{SState}_{\mathcal{A}} \triangleq \text{View}_{\mathcal{A}} \times (\text{AVal}' \rightarrow \wp(\text{World}_{\mathcal{A}})) \times \text{AVal}'$ and the set of *specification configurations* to be $\text{SConf}_{\mathcal{A}} \triangleq \text{Store} \times \text{Heap} \times \text{SState}_{\mathcal{A}}$. The set of *specification traces*, $\text{STrace}_{\mathcal{A}}$, is the set of infinite sequences of the form $\hat{c}_1 \pi_1 \hat{c}_2 \pi_2 \cdots$ where $\hat{c}_i \in \text{SConf}_{\mathcal{A}}$ and $\pi_i \in \{\text{env}, \text{loc}\}$.

Given a set of specification traces $\mathbb{T} \subseteq \text{STrace}$, we write $\hat{c} \pi \mathbb{T}$ for the set $\{\hat{c} \pi \hat{\tau} \mid \hat{\tau} \in \mathbb{T}\}$. We say a trace $\tau \in \text{Trace}$ is *terminal*, written $\text{term}(\tau)$, if it contains no local steps. We say a trace is *locally terminating*, written $\text{lterm}(\tau)$, if it contains finitely many local steps.

Definition E.22 (Trace Safety). Fix some $\mathbb{S} \in \text{Spec}$ of the form

$$\mathbb{S} = \forall x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{m; \lambda; \mathcal{A}} \quad (18)$$

For $\tau \in \text{Trace}$, $(p_h, p_a, v) \in \text{SState}_{\mathcal{A}}$, and $\mathbb{T} \subseteq \text{STrace}$, the *trace safety judgement* is the relation $\tau \vDash_{\mathbb{S}} p_h, p_a, v : \mathbb{T}$ defined coinductively in Fig. 25.¹²

Intuitively, the trace safety judgement walks down a concrete trace, checking its safety, and at the same time accumulating the possible instrumentations of the trace in \mathbb{T} which can later be checked against liveness properties. The judgement $\tau \vDash_{\mathbb{S}} p_h, p_a, v : \mathbb{T}$ assumes the initial configuration (σ_0, h_0) of the trace τ satisfies $h_0 \in \llbracket p_h * p_a(v) * \text{True} \rrbracket_{\lambda}$. Rule **STUTTER** checks that any local step other than the linearization point updates the local Hoare view (to some p'_h) in a frame-preserving manner; this implies that, before the linearisation point, the abstract state v needs to be preserved by such step. Rule **LINPT** checks that the linearisation point is frame-preserving and consistent with the atomic postcondition Q_a . Both rules **STUTTER** and **LINPT** check that the Hoare postcondition is satisfied if we are considering the last local step of the trace. Rule **ENV** checks whether the current environment step, assumed to happen before the linearization point, can be seen as a transition changing the abstract state from v to v' in a way that does not disrupt any frame (including p_h). If that is the case, the rest of the trace is checked for safety. Rule **ENV'** performs the same check but after the linearisation point. In both cases, if the environment step cannot be seen as frame-preserving, then the trace is accepted since the environment did not respect the assumptions. Similarly, Rule **ENV_f** accepts the trace after a fault caused by the environment.

Building on trace safety, we can now define the semantics of a specification $\llbracket \mathbb{S} \rrbracket$ as the set of traces that are safe and that additionally satisfy the liveness constraints implied by the obligations and the liveness assumptions of \mathbb{S} . Conceptually, we want to require local termination if the environment satisfies the layered liveness invariants represented by pseudo-quantifiers and obligations. To

¹²By slight abuse of notation we write emp even though by the type $\text{AVal}' \rightarrow \wp(\text{World}_{\mathcal{A}})$ we should write $\lambda_. \text{Emp}_{\mathcal{A}}$; here τ ranges over subsequences of traces.

$$\begin{array}{c}
\frac{(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_h * p_a(v) \rightarrow p'_h * p_a(v) \quad (\sigma_2, h_2) \tau \vDash_{\mathbb{S}} p'_h, p_a, v : \mathbb{T} \quad \text{term}(\tau) \Rightarrow v = \langle v_1, v_2 \rangle \wedge p'_h = \mathcal{W} \llbracket Q_h(v_1, v_2) \rrbracket_{\mathcal{A}}^{\sigma_2}}{(\sigma_1, h_1) \text{loc } (\sigma_2, h_2) \tau \vDash_{\mathbb{S}} p_h, p_a, v : ((\sigma_1, h_1, p_h, p_a, v) \text{loc } \mathbb{T})} \text{STUTTER} \\
\\
\frac{(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_h * p_a(v) \rightarrow q'_h * \mathcal{W} \llbracket Q_a(v, v') \rrbracket_{\mathcal{A}} \quad \text{term}(\tau) \Rightarrow q'_h = \mathcal{W} \llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_2} \quad (\sigma_2, h_2) \tau \vDash_{\mathbb{S}} q'_h, \text{emp}, \langle v, v' \rangle : \mathbb{T}}{(\sigma_1, h_1) \text{loc } (\sigma_2, h_2) \tau \vDash_{\mathbb{S}} p_h, p_a, v : ((\sigma_1, h_1, p_h, p_a, v) \text{loc } \mathbb{T})} \text{LINPT} \\
\\
\frac{\mathbb{T} = \bigcup \{ (\sigma, h_1, p_h, p_a, v) \text{env } \mathbb{T}_{v'} \mid v' \in X, E(v') \} \quad \forall v' \in X. E(v') \Rightarrow (\sigma, h_2) \tau \vDash_{\mathbb{S}} p_h, p_a, v' : \mathbb{T}_{v'} \quad v \in \text{AVal} \quad E(v') \triangleq (\exists p_e, p'_e. h_1 \in \llbracket p_h * p_a(v) * p_e \rrbracket_{\lambda} \wedge (h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_a(v) * p_e \rightarrow p_a(v') * p'_e)}{(\sigma, h_1) \text{env } (\sigma, h_2) \tau \vDash_{\mathbb{S}} p_h, p_a, v : \mathbb{T}} \text{ENV} \\
\\
\frac{\text{if } \exists p_e, p'_e. h_1 \in \llbracket p_h * p_e \rrbracket_{\lambda} \wedge (h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_e \rightarrow p'_e \text{ then } (\sigma, h_2) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : \mathbb{T} \text{ else } \mathbb{T} = \emptyset}{(\sigma, h_1) \text{env } (\sigma, h_2) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : ((\sigma_1, h_1, p_h, \text{emp}, \langle v, v' \rangle) \text{env } \mathbb{T})} \text{ENV}' \\
\\
\frac{}{(\sigma, h) \text{env } \not\vdash \tau \vDash_{\mathbb{S}} p_h, p_a, v : \emptyset} \text{ENV} \not\vdash
\end{array}$$

Fig. 25. Safety Specification Semantics

understand the idea, consider the case of liveness invariants encoded by obligations. The idea is to examine the traces instrumented with the logical state, and consider for each position which obligations are held by the environment and which are held locally. Now suppose the environment always eventually fulfills *every* obligation (i.e. for each obligation O there are infinitely many positions where O is not held by the environment). This environment is certainly *good* with respect to the liveness assumptions. When is the environment allowed to keep an obligation O forever? Only when we locally hold forever some obligation of layer strictly smaller than $\text{lay}(O)$. This intuition about obligations extends to liveness assumptions attached to pseudo-quantifications in the triple and in the atomicity context. To harmonise pseudo-quantification and obligation-related liveness assumptions, we collect all of them in a set of so-called *pseudo-obligations*, and uniformly impose the above conditions on liveness, as formalised by the *goodenv* predicate.

Definition E.23 (Specification Semantics). Fix \mathbb{S} and its components as in (18). Such a specification constrains the environment to maintain the liveness invariants implied by obligations, the liveness assumptions in \mathcal{A} and the one of the pseudo-quantifier. We collect them in the set of *pseudo-obligations*

$$\text{POb} \triangleq \text{AOB} \uplus \{ (r, \text{live}(\mathcal{A}, r)) \mid r \in \text{dom}(\mathcal{A}) \} \uplus \{ X \twoheadrightarrow_k X' \}.$$

We extend the layer function to $\text{lay} : \text{POb} \rightarrow \mathcal{L}$ by setting $\text{lay}(a) = k$ if $a = (r, X \twoheadrightarrow_k X')$ and $\text{lay}(X \twoheadrightarrow_k X') = k$. Furthermore, define

$$\text{POb}_{<k} \triangleq \{ \widehat{O} \in \text{POb} \mid \text{lay}(\widehat{O}) < k \} \quad \text{AOB}_{<k} \triangleq \{ O \in \text{AOB} \mid \text{lay}(O) < k \}$$

Given a $\hat{c} \in \text{SConf}_{\mathcal{A}}$, we want to define which are the possible worlds that represent the current local and shared information about the configuration. We thus define the predicate $\hat{c} \prec (w_h, w_a)$ which holds, for $\hat{c} = (\sigma, h, p_h, p_a, v)$, if $w_h \in p_h \wedge w_a \in p_a(v) \wedge \exists w_e. h \in \llbracket w_h \odot w_a \odot w_e \rrbracket_{\lambda}$.

We define two predicates $\text{envheld}(\widehat{O}, \hat{c})$, indicating when a pseudo-obligation $\widehat{O} \in \text{POb}$ is considered to be held by the environment in configuration $\hat{c} \in \text{SConf}_{\mathcal{A}}$, and $\text{loheld}(O, \hat{c})$, indicating

when an obligation $O \in \text{Oblig}$ is considered to be held locally in configuration $\hat{c} \in \text{SConf}_{\mathcal{A}}$:

$$\text{envheld}(\widehat{O}, \hat{c}) \triangleq \begin{cases} \forall w_h, w_a. \hat{c} \prec (w_h, w_a) \Rightarrow \exists r. \xi_{w_h \odot w_a}(r) \sqsupseteq \widehat{O} & \text{if } \widehat{O} \in \text{AOB} \\ \forall w_h, w_a. \hat{c} \prec (w_h, w_a) \Rightarrow \text{ast}_{w_h \odot w_a}(r) \notin X_2 & \text{if } \widehat{O} = (r, X_1 \rightarrow_k X_2) \\ v \in X_1 \setminus X_2 & \text{if } \hat{c} = (_, _, _, v) \wedge \widehat{O} = (X_1 \rightarrow_k X_2) \end{cases}$$

$$\text{locheld}(O, \hat{c}) \triangleq \exists w_h, w_a, r. \hat{c} \prec (w_h, w_a) \wedge \theta_{w_h}(r) \sqsupseteq O$$

Finally, for layer m from the context of \mathbb{S} , the $\text{goodenv}_m(\hat{\tau})$ predicate checks whether the environment is satisfying the liveness assumptions of the specification:

$$\text{goodenv}_m(\hat{\tau}) \triangleq \forall \widehat{O} \in \text{POb}_{< m}. \text{if } \forall O \in \text{AOB}_{< \text{lay}(\widehat{O})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}(j)) \\ \text{then } \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{envheld}(\widehat{O}, \hat{\tau}(j))$$

Given \mathbb{S} as in (18), let $p_h = \mathcal{W}[[P_h]]_{\mathcal{A}}^{\sigma_0}$, $\forall v \in \text{AVal}' . p_a(v) = \mathcal{W}[[v \in X \wedge P_a(v)]]_{\mathcal{A}}$. We define the trace semantics $[\mathbb{S}] \subseteq \text{Trace}$ of \mathbb{S} as the set:

$$[\mathbb{S}] \triangleq \left\{ (\sigma_0, h_0) \tau \left| \begin{array}{l} \forall v_0 \in X. \text{if } h_0 \in [[p_h * p_a(v_0) * \text{True}]]_{\lambda} \\ \text{then } \exists \mathbb{T}. (\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T} \\ \wedge \forall \hat{\tau} \in \mathbb{T}. \text{goodenv}_m(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau}) \end{array} \right. \right\}$$

The intuition behind Definition E.23 is as follows. Once it has been established that there is a way to instrument the trace to justify why the local steps satisfy the safety constraints of \mathbb{S} , we consider the set of valid instrumentations \mathbb{T} . Each instrumentation should either be terminating, in which case the trace is accepted, or, if it is non-terminating, we must check that the non-termination was due to the environment not satisfying the liveness assumptions of \mathbb{S} . The predicate $\text{goodenv}_m(\hat{\tau})$ holds for a specification trace $\hat{\tau}$ if the environment keeps every pseudo-obligation live. This condition uses layers in a subtle way. The idea is that an environment is considered good, if it keeps any pseudo-obligation with layer k live, *provided no obligation of layer $< k$ is constantly held by the local thread*. This effectively encodes the acyclicity of the layered termination argument. Each of the threads t is allowed to keep an obligation constantly unfulfilled, as long as it can blame some other thread t' by showing an obligation of strictly lower level that is kept constantly unfulfilled by t' . Since layers are well-founded there needs to be some thread that will not be justified in not fulfilling some of its obligations.

Definition E.24 (Semantic Triple). The semantic judgment $\vDash_{\Phi} \mathbb{C} : \mathbb{S}$ indicates that the command \mathbb{C} , under a functional context Φ , satisfies the specification \mathbb{S} . Formally, define for some $F \subseteq \text{FName}$,

$$\mathcal{F}_F[\Phi] \triangleq \left\{ \varphi \left| \text{dom}(\varphi) = F \wedge \forall f \in F. \Phi(f) = (\vec{x}_f, \mathbb{S}_f) \wedge \varphi(f) = (\vec{x}_f, \mathbb{C}_f) \wedge [\mathbb{C}_f]_{\varphi} \subseteq [\mathbb{S}_f] \right. \right\}.$$

The semantic judgement is defined by $\vDash_{\Phi} \mathbb{C} : \mathbb{S} \Leftrightarrow [\mathbb{C}]_{\varphi} \subseteq [\mathbb{S}]$ for all $\varphi \in \mathcal{F}_{\text{fn}(\mathbb{C})}[\Phi]$. Note that when \mathbb{C} has no free function names, the judgement $\vDash_{\Phi} \mathbb{C} : \mathbb{S}$ simply means $[\mathbb{C}] \subseteq [\mathbb{S}]$.

F SOUNDNESS

In this section, we provide the details of the soundness of three rules: **LIVEC**, **PAR**, **WHILE**, **FRAME**, **LIVEO** and **LIVEA**. These are the only proof rules in TaDA-Live that bring in non-trivial liveness information. All other proof rules follow in the same way as for TaDA, with the liveness constraints on the traces being identical between the antecedent and consequent of such rules or being trivial in the case of command axioms. We will focus particularly on the liveness argument for these rules.

F.1 Environment Liveness Judgement Semantics

First we define give semantics to the judgement defined in Fig. 7.

Definition F.1 (Environment Liveness Judgement Semantics). The semantic environmental liveness judgement:

$$m; \lambda; \mathcal{A} \vDash L \xrightarrow{M} T$$

holds if for some arbitrary $\sigma \in \text{Store}$, for

$$t = \mathcal{W}[[T * \text{True}]]_{\mathcal{A}}^{\sigma}$$

$$l = \mathcal{W}[[L * \text{True}]]_{\mathcal{A}}^{\sigma}$$

$$l(\alpha) = \mathcal{W}[[L * M(\alpha) * \text{True}]]_{\mathcal{A}}^{\sigma}$$

and for arbitrary $h, h' \in [[l]]_{\lambda}$, if $(h, h') \vDash_{\lambda; \mathcal{A}} l \rightarrow l$, then:

- $\lambda; \mathcal{A} \vdash L \Rightarrow \exists \alpha. L * M(\alpha)$
- $\forall \alpha, \alpha'. h \in [[l(\alpha)]]_{\lambda} \wedge h' \in [[l(\alpha')]]_{\lambda} \Rightarrow \alpha \geq \alpha' \vee h \in [[t]]_{\lambda}$.
- For arbitrary $\alpha, \alpha' \leq \alpha$, there exists $r \in \text{Rld}$ and

$$\widehat{\mathcal{O}} \in \text{Oblig}_{<k} \uplus \{ (r, \text{live}(\mathcal{A}, r)) \mid r \in \text{dom}(\mathcal{A}), \text{lay}(\text{live}(\mathcal{A}, r)) < k \}$$

such that for any $w \in l(\alpha)$, $w' \in l(\alpha')$, if $h \in [w]_{\lambda} \wedge h' \in [w']_{\lambda}$, then the following hold:

- if $\widehat{\mathcal{O}} \in \text{Oblig}$, $\xi_w(r) \sqsupseteq \widehat{\mathcal{O}}$ and $\xi_{w'}(r) \not\sqsupseteq \widehat{\mathcal{O}}$, then $\alpha > \alpha'$ or $h \in [[t]]_{\lambda}$.
- if $\widehat{\mathcal{O}} = (r, X \rightarrow_k X')$, $\rho_w(r) \notin X'$ and $\rho_{w'}(r) \in X'$, then $\alpha > \alpha'$ or $h \in [[t]]_{\lambda}$.

Recall the definition of $\text{impr}_{\mathcal{A}}$ (Definition B.1): the condition $\text{impr}_{\mathcal{A}}(t_r^{\lambda}, L, L', R, R', T)$ holds if and only if, $R' \neq \emptyset$ and for all $w_1, w_2 \in \text{World}_{\mathcal{A}}$, α_1, α_2 , and $(x_1, x_2) \in R$:

$$\begin{aligned} (w_1 \vDash_{\mathcal{A}} L(\alpha_1) * t_r^{\lambda}(x_1) \wedge w_2 \vDash_{\mathcal{A}} L'(\alpha_2) * t_r^{\lambda}(x_2)) \\ \Rightarrow ((\alpha_2 \leq \alpha_1 \wedge ((x_1, x_2) \in R' \Rightarrow \alpha_2 < \alpha_1)) \vee w_2 \vDash_{\mathcal{A}} T * \text{True}) \end{aligned}$$

LEMMA F.2. *If $m; \lambda; \mathcal{A} \vdash L \xrightarrow{M} T$ then $m; \lambda; \mathcal{A} \vDash L \xrightarrow{M} T$.*

PROOF. Any derivation tree using the rules of Fig. 7 has the goal judgement as the root, as leaves applications of rules **LIVET** to **LIVEA**, and all internal nodes are applications are of rule **ECASE**. Thus, for any $h \in [[l * \text{True}]]_{\lambda}$ some of the leaves would apply, as they are restricted to some subset of l and all of l is covered. The conditions of the semantic judgment then follow directly from the side conditions of the leaf that applies. \square

F.2 Soundness of LIVEC

THEOREM F.3. *Let*

$$\mathbb{S} = \mathbb{W}x \in X \rightarrow_k X'. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{m; \lambda; \mathcal{A}}$$

$$\mathbb{S}' = \mathbb{W}x \in X. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{m; \lambda; \mathcal{A}}$$

If $m, k \geq n$ and

$$m; \lambda; \mathcal{A} \vDash \exists x \in X. P_a(x) \xrightarrow{M} \exists x' \in X'. P_a(x') \quad (19)$$

then $[[\mathbb{S}]] \subseteq [[\mathbb{S}']]$.

PROOF. Take $(\sigma_0, h_0)\tau \in [[\mathbb{S}]]$. Let:

$$p_h = \mathcal{W}[[P_h]]_{\mathcal{A}}^{\sigma_0}$$

$$p_a(v) = \mathcal{W}[[P_a(v)]]_{\mathcal{A}}^{\sigma_0}$$

$$m(\alpha) = \mathcal{W}[[M(\alpha)]]_{\mathcal{A}}^{\sigma_0}$$

To show $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S}' \rrbracket$, assume for some $v_0 \in X$, $h_0 \in \llbracket p_h * p_a(v_0) * \text{True} \rrbracket_\lambda$. Then, given this and $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S} \rrbracket$, for some $\mathbb{T} \in \mathcal{P}(\text{STrace})$:

$$(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T} \wedge \forall \hat{\tau} \in \mathbb{T}. \text{goodenv}_m(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$$

Given that the definition of $(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T}$ does not depend on the good states, X' , then $(\sigma_0, h_0) \tau \vDash_{\mathbb{S}'} p_h, p_a, v_0 : \mathbb{T}$ holds. To finish off, must show:

$$\forall \hat{\tau} \in \text{STrace}. \text{goodenv}_{\mathbb{S}'}(\hat{\tau}) \Rightarrow \text{goodenv}_{\mathbb{S}}(\hat{\tau})$$

As $\text{POb}_{< m_{\mathbb{S}'}}(\vec{X}_{\mathbb{S}'}, \mathcal{A}_{\mathbb{S}'}) = \text{POb}_{< m_{\mathbb{S}}}(\vec{X}_{\mathbb{S}}, \mathcal{A}_{\mathbb{S}}) \cup \{X \rightarrow_k X'\}$, it is sufficient to show that for arbitrary $\hat{\tau} \in \text{STrace}$, assuming

$$\text{goodenv}_{\mathbb{S}}(\hat{\tau}) \wedge \forall O \in \text{AOB}_{< k}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}(j))$$

then $\forall i \in \mathbb{N}. \exists j \geq i. \neg \text{envhld}(X \rightarrow_k X', \hat{\tau}(j))$ holds.

If there exists some $lin \in \mathbb{N}$ such that $\hat{\tau}(lin) = (_, _, _, \langle v, v' \rangle)$, then $\forall j \geq lin. \neg \text{envhld}(X \rightarrow_k X', \hat{\tau}(j))$, which implies the goal. Otherwise, for $i \in \mathbb{N}$, let $\hat{\tau}(i) = (\sigma^i, h^i, p_h^i, p_a^i, v^i)$.

From (19), $\lambda; \mathcal{A} \vDash \exists x \in X. P_a(x) \Rightarrow \exists x \in X, \alpha. P_a(x) * M(\alpha)$ holds, therefore, for any $w_h, w_a \in \text{View}_{\mathcal{A}}$ such that $(\sigma^i, h^i, p_h^i, p_a^i, v^i) \prec (w_h, w_a)$, then, for some α , $(\sigma^i, h^i, p_h^i, m(\alpha) * p_a^i, v^i) \prec (w_h, w_a)$ holds. By (19), there exists $r \in \text{Rld}$ and

$$\widehat{O} \in \text{Oblig}_{< k} \uplus \{ (r, \text{live}(\mathcal{A}, r)) \mid r \in \text{dom}(\mathcal{A}), \text{lay}(\text{live}(\mathcal{A}, r)) < k \}$$

such that:

- If $\widehat{O} \in \text{Oblig}_{< k}$, then $\xi_{w_h \odot w_a}(r) \supseteq \widehat{O}$
- If $Y \rightarrow_{k'} Y' \in \{ (r, \text{live}(\mathcal{A}, r)) \mid r \in \text{dom}(\mathcal{A}), \text{lay}(\text{live}(\mathcal{A}, r)) < k \}$, then $\rho_{w_h \odot w_a}(r) \notin Y'$.

In either case, $\text{goodenv}_{\mathbb{S}}(\hat{\tau})$, guarantees that there is eventually an atomic step that discharges this obligation and from (19), this will lower the metric to some $\alpha' < \alpha$. As ordinals are well founded, for arbitrary $i \in \mathbb{N}$, there exists $j \geq i$, such that $\hat{\tau}(j) = (\sigma^j, h^j, w_h^j, w_a^j, v^j)$ with $w_h^j \bullet w_a^j \in t$ and therefore $v^j \in X'$. This implies $\neg \text{envhld}(X \rightarrow_k X', \hat{\tau}(j))$ as required. \square

F.3 Soundness of PAR

Definition F.4 (Bowtie operator). The bowtie operator, \bowtie , which interleaves the subjective traces of two commands executed in parallel into a command from their combined perspective:

$$\begin{aligned} (\sigma, h) \text{env } \tau'_1 \bowtie (\sigma, h) \text{env } \tau'_2 &= (\sigma, h) \text{env } (\tau'_1 \bowtie \tau'_2) \\ (\sigma, h) \text{env } \tau'_1 \bowtie (\sigma, h) \text{loc } \tau'_2 &= (\sigma, h) \text{loc } (\tau'_1 \bowtie \tau'_2) \\ (\sigma, h) \text{loc } \tau'_1 \bowtie (\sigma, h) \text{env } \tau'_2 &= (\sigma, h) \text{loc } (\tau'_1 \bowtie \tau'_2) \end{aligned}$$

All other cases are undefined.

Definition F.5 (Specification Bowtie operator). The specification bowtie operator, $\overset{\mathbb{S}}{\bowtie}$, which interleaves the subjective specification traces of two commands executed in parallel into a command from their combined perspective:

$$\begin{aligned} (\sigma, h, p_1, \text{emp}, 1) \text{env } \tau'_1 \overset{\mathbb{S}}{\bowtie} (\sigma, h, p_2, \text{emp}, 1) \text{env } \tau'_2 &= (\sigma, h, p_1 * p_2, \text{emp}, 1) \text{env } (\tau'_1 \bowtie \tau'_2) \\ (\sigma, h, p_1, \text{emp}, 1) \text{env } \tau'_1 \overset{\mathbb{S}}{\bowtie} (\sigma, h, p_2, \text{emp}, 1) \text{loc } \tau'_2 &= (\sigma, h, p_1 * p_2, \text{emp}, 1) \text{loc } (\tau'_1 \bowtie \tau'_2) \\ (\sigma, h, p_1, \text{emp}, 1) \text{loc } \tau'_1 \overset{\mathbb{S}}{\bowtie} (\sigma, h, p_2, \text{emp}, 1) \text{env } \tau'_2 &= (\sigma, h, p_1 * p_2, \text{emp}, 1) \text{loc } (\tau'_1 \bowtie \tau'_2) \end{aligned}$$

All other cases are undefined.

LEMMA F.6. For any $\varphi \in \text{Flmpl}$:

$$\forall \tau \in \llbracket \mathbb{C}_1 \parallel \mathbb{C}_2 \rrbracket_{\varphi}. \exists \tau_1 \in \llbracket \mathbb{C}_1 \rrbracket, \tau_2 \in \llbracket \mathbb{C}_2 \rrbracket. \tau = \tau_1 \bowtie \tau_2$$

PROOF. Straightforward by induction on \rightarrow_φ . \square

LEMMA F.7. For any trace $(\sigma_0, h_0) \tau (\sigma_1, h_1) \tau' \in \llbracket \mathbb{C} \rrbracket_\varphi$, we have $\forall x \in \text{PVar} \setminus \text{mods}(\mathbb{C}). \sigma_0(x) = \sigma_1(x)$.

PROOF. Straightforward by induction on the length of the trace. \square

For the rest of the section, we name the specifications involved in the **PAR** rule as follows:

$$\mathbb{S}_1 = \{P_1\} \cdot \{Q_1\}_{m_1, \lambda; \mathcal{A}} \quad \mathbb{S}_2 = \{P_2\} \cdot \{Q_2\}_{m_2, \lambda; \mathcal{A}} \quad \mathbb{S} = \{P_1 * P_2\} \cdot \{Q_1 * Q_2\}_{m, \lambda; \mathcal{A}}$$

LEMMA F.8. For arbitrary $(\sigma_0, h_0)\tau, (\sigma_0, h_0)\tau_1, (\sigma_0, h_0)\tau_2 \in \text{Trace}$, $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{P}(\text{STrace})$, $v_1, v_2 \in \{1, \langle 1, 1 \rangle\}$, and, $p'_1, p'_2 \in \text{View}_{\mathcal{A}}$, then:

$$\left. \begin{array}{l} (\sigma_0, h_0)\tau = (\sigma_0, h_0)\tau_1 \bowtie (\sigma_0, h_0)\tau_2 \\ (\sigma_0, h_0)\tau_1 \vDash_{\mathbb{S}_1} p'_1, \text{emp}, v_1 : \mathbb{T}_1 \\ (\sigma_0, h_0)\tau_2 \vDash_{\mathbb{S}_2} p'_2, \text{emp}, v_2 : \mathbb{T}_2 \\ h_0 \in \llbracket p'_1 * p'_2 * \text{True} \rrbracket_\lambda \\ \text{term}((\sigma_0, h_0)\tau_1) \Rightarrow p'_1 = \mathcal{W} \llbracket Q_1 \rrbracket_{\mathcal{A}}^\sigma \\ \text{term}((\sigma_0, h_0)\tau_2) \Rightarrow p'_2 = \mathcal{W} \llbracket Q_2 \rrbracket_{\mathcal{A}}^\sigma \end{array} \right\} \Rightarrow \begin{array}{l} \exists \mathbb{T} \in \mathcal{P}(\text{STrace}), v \in \{1, \langle 1, 1 \rangle\}. \\ (\sigma, h)\tau \vDash_{\mathbb{S}} p'_1 * p'_2, \text{emp}, v : \mathbb{T} \wedge \\ \forall \hat{\tau} \in \mathbb{T}. \exists \hat{\tau}_1 \in \mathbb{T}_1, \hat{\tau}_2 \in \mathbb{T}_2. \hat{\tau} = \hat{\tau}_1 \bowtie \hat{\tau}_2 \wedge \\ (v_1 = \langle 1, 1 \rangle \wedge v_2 = \langle 1, 1 \rangle) \Leftrightarrow v = \langle 1, 1 \rangle \end{array}$$

PROOF. This lemma is proven by coinduction on the structure of $(\sigma_0, h_0)\tau$.

The trace either starts with a local, or an environment step. We split on the two cases:

Case $(\sigma, h)\tau = (\sigma, h) \text{env} (\sigma, h')\tau'$. Take $(\sigma_0, h_0)\tau_1, (\sigma_0, h_0)\tau_2 \in \text{Trace}$, $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{P}(\text{STrace})$, $v_1, v_2 \in \{1, \langle 1, 1 \rangle\}$, and, $p'_1, p'_2 \in \text{View}_{\mathcal{A}}$ arbitrary, and assume:

$$(\sigma_0, h_0)\tau = (\sigma_0, h_0)\tau_1 \bowtie (\sigma_0, h_0)\tau_2 \tag{20}$$

$$(\sigma_0, h_0)\tau_1 \vDash_{\mathbb{S}_1} p'_1, \text{emp}, v_1 : \mathbb{T}_1 \tag{21}$$

$$(\sigma_0, h_0)\tau_2 \vDash_{\mathbb{S}_2} p'_2, \text{emp}, v_2 : \mathbb{T}_2 \tag{22}$$

$$h_0 \in \llbracket p'_1 * p'_2 * \text{True} \rrbracket_\lambda \tag{23}$$

$$\text{term}((\sigma_0, h_0)\tau_1) \Rightarrow p'_1 = \mathcal{W} \llbracket Q_1 \rrbracket_{\mathcal{A}}^\sigma \tag{24}$$

$$\text{term}((\sigma_0, h_0)\tau_2) \Rightarrow p'_2 = \mathcal{W} \llbracket Q_2 \rrbracket_{\mathcal{A}}^\sigma \tag{25}$$

Given 20 and the definition of \bowtie :

$$(\sigma_0, h_0)\tau_1 = (\sigma_0, h_0) \text{env} (\sigma_0, h')\tau'_1$$

$$(\sigma_0, h_0)\tau_2 = (\sigma_0, h_0) \text{env} (\sigma_0, h')\tau'_2$$

$$(\sigma_0, h')\tau' = (\sigma_0, h')\tau'_1 \bowtie (\sigma_0, h')\tau'_2$$

Now to prove the goal, consider the case $v_1 = \langle 1, 1 \rangle$ and $v_2 = \langle 1, 1 \rangle$. In this case, take $v = \langle 1, 1 \rangle$, so **ENV'** must hold for the goal as well as 21 and 22. Note that this choice of v immediately satisfies the third conjunct of the goal. To show **ENV'** holds for the goal, given some $p_e, p'_e \in \text{View}_{\mathcal{A}}$, assume:

$$h_0 \in \llbracket p'_1 * p'_2 * p_e \rrbracket \wedge (h_0, h') \vDash_{\lambda; \mathcal{A}} p'_1 * p'_2 * p_e \rightarrow p'_1 * p'_2 * p'_e$$

By substitution, this implies both:

$$\exists p_e, p'_e. h_0 \in \llbracket p'_1 * p_e \rrbracket \wedge (h, h') \vDash_{\lambda; \mathcal{A}} p'_1 * p_e \rightarrow p'_1 * p'_e$$

$$\exists p_e, p'_e. h_0 \in \llbracket p'_2 * p_e \rrbracket \wedge (h, h') \vDash_{\lambda; \mathcal{A}} p'_2 * p_e \rightarrow p'_2 * p'_e$$

Given 21 and 22, these imply:

$$\begin{aligned} (\sigma_0, h')\tau'_1 &\vDash_{\mathbb{S}_1} p'_1, \text{emp}, v_1 : \mathbb{T}'_1 \\ (\sigma_0, h')\tau_2 &\vDash_{\mathbb{S}_2} p'_2, \text{emp}, v_2 : \mathbb{T}'_2 \end{aligned}$$

where:

$$\begin{aligned} \mathbb{T}_1 &= (\sigma_0, h_0, p_1, \text{emp}, \langle 1, 1 \rangle) \text{env } \mathbb{T}'_1 \\ \mathbb{T}_2 &= (\sigma_0, h_0, p_2, \text{emp}, \langle 1, 1 \rangle) \text{env } \mathbb{T}'_2 \end{aligned}$$

Assumption (23) and $(h_0, h') \vDash_{\lambda, \mathcal{A}} p'_1 * p'_2 * p_e \rightarrow p'_1 * p'_2 * p'_e$ yield:

$$h' \in \llbracket p'_1 * p'_2 * \text{True} \rrbracket_{\lambda} \quad (26)$$

Now, by using the inductive assumption, as 24 and 25 clearly imply the same assertions for $(\sigma_0, h')\tau'_1$ and $(\sigma_0, h')\tau'_2$ respectively, for some $\mathbb{T}' \in \mathcal{P}(\text{STrace})$:

$$(\sigma_0, h')\tau' \vDash_{\mathbb{S}} p'_1 * p'_2, \text{emp}, v : \mathbb{T}' \quad (27)$$

$$\forall \hat{\tau} \in \mathbb{T}'. \exists \hat{\tau}_1 \in \mathbb{T}'_1, \hat{\tau}_2 \in \mathbb{T}'_2. \hat{\tau} = \hat{\tau}_1 \overset{\mathbb{S}}{\bowtie} \hat{\tau}_2 \quad (28)$$

From this first consequence:

$$(\sigma_0, h)\tau \vDash_{\mathbb{S}} p'_1 * p'_2, \text{emp}, v : \mathbb{T}$$

holds, where $\mathbb{T} = (\sigma_0, h, p'_1 * p'_2, \text{emp}, v) \text{env } \mathbb{T}'$. This is the first conjunct of the goal.

Finally, taking $\hat{\tau} \in \mathbb{T}$ arbitrary, there exists $\hat{\tau}' \in \mathbb{T}'$ such that $\hat{\tau} = (\sigma_0, h, p'_1 * p'_2, \text{emp}, v) \text{env } \hat{\tau}'$. From the second consequence of our inductive assumption, it follows that there exist $\hat{\tau}'_1 \in \mathbb{T}'_1$ and $\hat{\tau}'_2 \in \mathbb{T}'_2$ such that $\hat{\tau}' = \hat{\tau}'_1 \overset{\mathbb{S}}{\bowtie} \hat{\tau}'_2$. Then, from the definitions of \mathbb{T}_1 and \mathbb{T}_2 , $(\sigma_0, h_0, p_1, \text{emp}, \langle 1, 1 \rangle) \text{env } \hat{\tau}'_1 \in \mathbb{T}_1$ and $(\sigma_0, h_0, p_2, \text{emp}, \langle 1, 1 \rangle) \text{env } \hat{\tau}'_2 \in \mathbb{T}_2$ hold, and $\hat{\tau} = (\sigma_0, h_0, p_1, \text{emp}, \langle 1, 1 \rangle) \text{env } \hat{\tau}'_1 \overset{\mathbb{S}}{\bowtie} (\sigma_0, h_0, p_2, \text{emp}, \langle 1, 1 \rangle) \text{env } \hat{\tau}'_2 \in \mathbb{T}_2$ holds as required.

Other cases for v_1, v_2 follow similarly.

Case $(\sigma, h)\tau = (\sigma, h) \text{loc } (\sigma, h')\tau'$. Here the variable store does not change as $\text{mods}(\mathbb{C}_1 || \mathbb{C}_2) = \emptyset$, due to lemma F.7 and the syntactic restriction on parallel commands, requiring both threads to not modify the value of any variable. To prove the goal, take $(\sigma_0, h_0)\tau_1, (\sigma_0, h_0)\tau_2 \in \text{Trace}$, $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{P}(\text{STrace})$, $v_1, v_2 \in \{1, \langle 1, 1 \rangle\}$, and, $p'_1, p'_2 \in \text{View}_{\mathcal{A}}$ arbitrary, and assume:

$$(\sigma_0, h_0)\tau = (\sigma_0, h_0)\tau_1 \bowtie (\sigma_0, h_0)\tau_2 \quad (29)$$

$$(\sigma_0, h_0)\tau_1 \vDash_{\mathbb{S}_1} p'_1, \text{emp}, v_1 : \mathbb{T}_1 \quad (30)$$

$$(\sigma_0, h_0)\tau_2 \vDash_{\mathbb{S}_2} p'_2, \text{emp}, v_2 : \mathbb{T}_2 \quad (31)$$

$$h_0 \in \llbracket p'_1 * p'_2 * \text{True} \rrbracket_{\lambda} \quad (32)$$

$$\text{term}((\sigma_0, h_0)\tau_1) \Rightarrow p'_1 = \mathcal{W}[\llbracket Q_1 \rrbracket_{\mathcal{A}}]_{\sigma}^{\sigma} \quad (33)$$

$$\text{term}((\sigma_0, h_0)\tau_2) \Rightarrow p'_2 = \mathcal{W}[\llbracket Q_2 \rrbracket_{\mathcal{A}}]_{\sigma}^{\sigma} \quad (34)$$

Given 29 and the definition of \bowtie , either:

$$(\sigma_0, h_0)\tau_1 = (\sigma_0, h_0) \text{loc } (\sigma_0, h')\tau'_1$$

$$(\sigma_0, h_0)\tau_2 = (\sigma_0, h_0) \text{env } (\sigma_0, h')\tau'_2$$

or:

$$\begin{aligned}(\sigma_0, h_0)\tau_1 &= (\sigma_0, h_0) \text{ env } (\sigma_0, h')\tau'_1 \\ (\sigma_0, h_0)\tau_2 &= (\sigma_0, h_0) \text{ loc } (\sigma_0, h')\tau'_2\end{aligned}$$

and in both cases:

$$(\sigma_0, h')\tau' = (\sigma_0, h')\tau'_1 \bowtie (\sigma_0, h')\tau'_2$$

Consider the first case, the second will follow symmetrically. Assume that the **STUTTER** rule holds for $(\sigma, h) \text{ loc } (\sigma, h')\tau_1 \vDash_{\mathbb{S}_1} p'_1, \text{ emp}, v_1 : \mathbb{T}_1$, then, for some $p''_1 \in \text{View}_{\mathcal{A}}$:

$$\begin{aligned}(h_0, h') &\vDash_{\lambda; \mathcal{A}} p'_1 \rightarrow p''_1 \\ (\sigma_0, h') \tau'_1 &\vDash_{\mathbb{S}_1} p''_1, \text{ emp}, v_1 : \mathbb{T}'_1 \\ \text{term}((\sigma_0, h')\tau'_1) &\Rightarrow v_1 = \langle 1, 1 \rangle \wedge p''_1 = \mathcal{W}[[Q_1]_{\mathcal{A}}]_{\sigma_0}^{\sigma_0}\end{aligned}$$

where $\mathbb{T}_1 = (\sigma_0, h_0, p_1, \text{ emp}, v_1) \text{ loc } \mathbb{T}'_1$. Given 32 and $(h_0, h') \vDash_{\lambda; \mathcal{A}} p'_1 \rightarrow p''_1, h' \in \llbracket p''_1 * p'_2 * \text{True} \rrbracket_{\lambda}$ holds. Given $(h_0, h') \vDash_{\lambda; \mathcal{A}} p'_1 \rightarrow p''_1, (h_0, h') \vDash_{\lambda; \mathcal{A}} p'_1 * p_2 \rightarrow p''_1 * p_2$, also holds. Using this and **ENV** or **ENV'**:

$$(\sigma, h') \tau'_2 \vDash_{\mathbb{S}_2} p'_2, \text{ emp}, v_2 : \mathbb{T}'_2$$

where $\mathbb{T}_2 = (\sigma_0, h_0, p_2, \text{ emp}, v_2) \text{ loc } \mathbb{T}'_2$. Now using the inductive assumption, as, once again, 33 and 34 clearly imply the same assertions for $(\sigma_0, h')\tau'_1$ and $(\sigma_0, h')\tau'_2$ respectively, for some $\mathbb{T}' \in \mathcal{P}(\text{STrace})$:

$$(\sigma_0, h')\tau' \vDash_{\mathbb{S}} p''_1 * p'_2, \text{ emp}, v : \mathbb{T}' \wedge \quad (35)$$

$$\forall \hat{\tau} \in \mathbb{T}'. \exists \hat{\tau}_1 \in \mathbb{T}'_1, \hat{\tau}_2 \in \mathbb{T}'_2. \hat{\tau} = \hat{\tau}_1 \bowtie \hat{\tau}_2 \wedge \quad (36)$$

$$(v_1 = \langle 1, 1 \rangle \wedge v_2 = \langle 1, 1 \rangle) \Leftrightarrow v = \langle 1, 1 \rangle \quad (37)$$

The second and third consequents imply the equivalent conjuncts of the goal with the same method as in the env case and directly respectively. As we have shown $(h, h') \vDash_{\lambda; \mathcal{A}} p'_1 * p_2 \rightarrow p''_1 * p_2$ holds, using the **STUTTER** rule, to show that $(\sigma_0, h_0)\tau \vDash_{\mathbb{S}} p'_1 * p'_2, \text{ emp}, v : \mathbb{T}$ holds, where $\mathbb{T} = (\sigma_0, h_0, p'_1 * p'_2, \text{ emp}, v) \text{ loc } \mathbb{T}'$, it suffices to show:

$$\text{term}((\sigma_0, h')\tau') \Rightarrow v = \langle 1, 1 \rangle \wedge p''_1 * p'_2 = \mathcal{W}[[Q_1 * Q_2]_{\mathcal{A}}]_{\sigma_0}^{\sigma_0}$$

Assuming $\text{term}((\sigma_0, h')\tau')$ holds, then $\text{term}((\sigma_0, h')\tau'_1)$ and $\text{term}((\sigma_0, h')\tau'_2)$ hold. From this it follows that $v_1, v_2 = \langle 1, 1 \rangle$, so, due to 37, $v = \langle 1, 1 \rangle$.

Finally, due to $\text{term}((\sigma_0, h')\tau'_1)$ and $\text{term}((\sigma_0, h')\tau'_2)$, $p''_1 = \mathcal{W}[[Q_1]_{\mathcal{A}}]_{\sigma_0}^{\sigma_0}$ and $p'_2 = \mathcal{W}[[Q_2]_{\mathcal{A}}]_{\sigma_0}^{\sigma_0}$ hold respectively, yielding $p''_1 * p'_2 = \mathcal{W}[[Q_1 * Q_2]_{\mathcal{A}}]_{\sigma_0}^{\sigma_0}$, as required.

The **LINPT** rule follows similarly. □

THEOREM F.9. *Given*

$$m_1; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1\} \mathbb{C}_1 \{Q_1\} \quad (38)$$

$$m_2; \lambda; \mathcal{A} \vdash_{\Phi} \{P_2\} \mathbb{C}_2 \{Q_2\} \quad (39)$$

$$\lambda; \mathcal{A} \vdash Q_1 \triangleright m_2 \leq m \quad (40)$$

$$\lambda; \mathcal{A} \vdash Q_2 \triangleright m_1 \leq m \quad (41)$$

then:

$$m; \lambda; \mathcal{A} \vdash_{\Phi} \{P_1 * P_2\} \mathbb{C}_1 || \mathbb{C}_2 \{Q_1 * Q_2\}$$

PROOF. Take $\varphi \in \mathcal{F}_{\text{fin}(\mathbb{C})}[\Phi]$ arbitrary, from 38 and 39, $[\mathbb{C}_1]_\varphi \subseteq [\mathbb{S}_1]$ and $[\mathbb{C}_2]_\varphi \subseteq [\mathbb{S}_2]$ hold. Given an arbitrary $(\sigma_0, h_0)\tau \in [\mathbb{C}_1 \parallel \mathbb{C}_2]_\varphi$, need to show $(\sigma_0, h_0)\tau \in [\mathbb{S}]$. Let

$$\begin{aligned} p_1 &= \mathcal{W}[P_1]_{\mathcal{A}}^{\sigma_0} \\ p_2 &= \mathcal{W}[P_2]_{\mathcal{A}}^{\sigma_0} \end{aligned}$$

To reach the goal, assume $h_0 \in \llbracket p_1 * p_2 * \text{True} \rrbracket_\lambda$. Then $h_0 \in \llbracket p_1 * \text{True} \rrbracket_\lambda$ and $h_0 \in \llbracket p_2 * \text{True} \rrbracket_\lambda$ hold. From F.6 and the definition of \bowtie , there exists $(\sigma_0, h_0)\tau_1 \in [\mathbb{C}_1]_\varphi$ and $(\sigma_0, h_0)\tau_2 \in [\mathbb{C}_2]_\varphi$ such that $(\sigma_0, h_0)\tau = (\sigma_0, h_0)\tau_1 \bowtie (\sigma_0, h_0)\tau_2$. As $[\mathbb{C}_1]_\varphi \subseteq [\mathbb{S}_1]$ and $[\mathbb{C}_2]_\varphi \subseteq [\mathbb{S}_2]$, $(\sigma_0, h_0)\tau_1 \in [\mathbb{S}_1]$ and $(\sigma_0, h_0)\tau_2 \in [\mathbb{S}_2]$ hold. Now, as $h_0 \in \llbracket p_1 * \text{True} \rrbracket_\lambda$ and $h_0 \in \llbracket p_2 * \text{True} \rrbracket_\lambda$, then for some $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{P}(\text{STrace})$:

$$\begin{aligned} (\sigma_0, h_0)\tau_1 &\vDash_{\mathbb{S}} p_1, \text{emp}, 1 : \mathbb{T}_1 \\ (\sigma_0, h_0)\tau_2 &\vDash_{\mathbb{S}} p_2, \text{emp}, 1 : \mathbb{T}_2 \end{aligned}$$

and

$$\begin{aligned} \forall \hat{\tau}_1 \in \mathbb{T}_1. \text{goodenv}_{\mathbb{S}}(\hat{\tau}_1) &\Rightarrow \text{lterm}(\hat{\tau}_1) \\ \forall \hat{\tau}_2 \in \mathbb{T}_2. \text{goodenv}_{\mathbb{S}}(\hat{\tau}_2) &\Rightarrow \text{lterm}(\hat{\tau}_2) \end{aligned}$$

As all commands must take at least one step, $\neg \text{term}((\sigma_0, h_0)\tau_1)$ and $\neg \text{term}((\sigma_0, h_0)\tau_2)$ hold, therefore:

$$\begin{aligned} \text{term}((\sigma_0, h_0)\tau_1) &\Rightarrow p'_1 = \mathcal{W}[Q_1]_{\mathcal{A}}^{\sigma} \\ \text{term}((\sigma_0, h_0)\tau_2) &\Rightarrow p'_2 = \mathcal{W}[Q_2]_{\mathcal{A}}^{\sigma} \end{aligned}$$

hold. Now, using lemma F.8, there exists $\mathbb{T} \in \mathcal{P}(\text{STrace})$ such that:

$$(\sigma_0, h_0)\tau \vDash_{\mathbb{S}} p_1 * p_2, \text{emp}, 1 : \mathbb{T}$$

and for any $\hat{\tau} \in \mathbb{T}$, there exist $\hat{\tau}_1 \in \mathbb{T}_1$ and $\hat{\tau}_2 \in \mathbb{T}_2$, such that $\hat{\tau} = \hat{\tau}_1 \overset{\mathbb{S}}{\bowtie} \hat{\tau}_2$. It now suffices to show that $\forall \hat{\tau} \in \mathbb{T}. \text{goodenv}_{\mathbb{S}}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$. Take $\hat{\tau} \in \mathbb{T}$ arbitrary and $\hat{\tau}_1 \in \mathbb{T}_1$ and $\hat{\tau}_2 \in \mathbb{T}_2$ such that $\hat{\tau} = \hat{\tau}_1 \overset{\mathbb{S}}{\bowtie} \hat{\tau}_2$. From above:

$$\begin{aligned} \text{goodenv}_{\mathbb{S}_1}(\hat{\tau}_1) &\Rightarrow \text{lterm}(\hat{\tau}_1) \\ \text{goodenv}_{\mathbb{S}_2}(\hat{\tau}_2) &\Rightarrow \text{lterm}(\hat{\tau}_2) \end{aligned}$$

holds. To reach the goal, split on $\text{lterm}(\hat{\tau}_1)$ and $\text{lterm}(\hat{\tau}_2)$.

$\text{lterm}(\hat{\tau}_1) \wedge \text{lterm}(\hat{\tau}_2)$: In this case, $\text{lterm}(\hat{\tau})$ holds, therefore $\text{goodenv}_{\mathbb{S}}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$ holds trivially.

$\text{lterm}(\hat{\tau}_1) \wedge \neg \text{lterm}(\hat{\tau}_2)$: From $\neg \text{lterm}(\hat{\tau}_2)$, $\neg \text{goodenv}_{\mathbb{S}_2}(\hat{\tau}_2)$ holds:

$$\begin{aligned} \exists \hat{\mathcal{O}} \in \text{POb}_{< m_2}(\mathcal{A}_{\mathbb{S}}). (\forall \mathcal{O} \in \text{AOb}_{< \text{lay}(\hat{\mathcal{O}})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(\mathcal{O}, \hat{\tau}_2(j))) \wedge \\ (\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\hat{\mathcal{O}}, \hat{\tau}_2(j))) \end{aligned}$$

As $\text{lterm}(\hat{\tau}_1)$, there exists some $i_1 \in \mathbb{N}$, an index after which the trace $\hat{\tau}_1$ only performs env steps, in particular, for any $j \geq i_1$, $\hat{\tau}_1(j) = (\sigma, h, \mathcal{W}[Q_1]_{\mathcal{A}}^{\sigma}, \text{emp}, \langle 1, 1 \rangle)$, therefore $\hat{\tau}(j) = (\sigma, h, \mathcal{W}[Q_1]_{\mathcal{A}}^{\sigma} * p'_2, \text{emp}, \langle 1, 1 \rangle)$, where $\hat{\tau}_2(j) = (\sigma, h, p'_2, \text{emp}, \langle 1, 1 \rangle)$. Given $\lambda; \mathcal{A} \vdash Q_1 \triangleright m_2$:

$$\begin{aligned} (\forall \mathcal{O} \in \text{AOb}_{< \text{lay}(\hat{\mathcal{O}})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(\mathcal{O}, \hat{\tau}_2(j))) \Rightarrow \\ (\forall \mathcal{O} \in \text{AOb}_{< \text{lay}(\hat{\mathcal{O}})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(\mathcal{O}, \hat{\tau}(j))) \end{aligned}$$

and similarly as $\hat{\mathcal{O}} \in \text{POb}_{< m_2}(\mathcal{A}_{\mathbb{S}})$:

$$(\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\hat{\mathcal{O}}, \hat{\tau}_2(j))) \Rightarrow (\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\hat{\mathcal{O}}, \hat{\tau}(j)))$$

As $m_2 \leq m$, $\widehat{O} \in \text{POb}_{<m}(\mathcal{A}_{\mathbb{S}})$. Finally, from this $\neg \text{goodenv}_{\mathbb{S}}(\hat{\tau})$ holds, implying $\text{goodenv}_{\mathbb{S}}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$.

$\neg \text{lterm}(\hat{\tau}_1) \wedge \text{lterm}(\hat{\tau}_2)$: Similarly to the previous case.

$\neg \text{lterm}(\hat{\tau}_1) \wedge \neg \text{lterm}(\hat{\tau}_2)$: From this we can infer $\neg \text{goodenv}_{\mathbb{S}_1}(\hat{\tau}_1)$ and $\neg \text{goodenv}_{\mathbb{S}_2}(\hat{\tau}_2)$. Assume $\text{goodenv}_{\mathbb{S}}(\hat{\tau})$ for a contradiction. From $\neg \text{goodenv}_{\mathbb{S}_1}(\hat{\tau}_1)$, for some $\widehat{O} \in \text{POb}_{<m_1}(\mathcal{A}_{\mathbb{S}})$:

$$(\forall O \in \text{AOb}_{<\text{lay}(\widehat{O})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}_1(j))) \wedge (\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\widehat{O}, \hat{\tau}_1(j)))$$

From this and $\text{goodenv}_{\mathbb{S}}(\hat{\tau})$, there is some $i \in \mathbb{N}$ such that:

$$\forall j \geq i. \text{locheld}(\widehat{O}, \hat{\tau}_2(j))$$

From $\neg \text{goodenv}_{\mathbb{S}_2}(\hat{\tau}_2)$, for some $\widehat{O}' \in \text{POb}_{<m_2}(\mathcal{A}_{\mathbb{S}})$:

$$(\forall O \in \text{AOb}_{<\text{lay}(\widehat{O}')}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}_2(j))) \wedge (\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\widehat{O}', \hat{\tau}_2(j)))$$

For $\forall j \geq i. \text{locheld}(\widehat{O}, \hat{\tau}_2(j))$ and $\forall O \in \text{AOb}_{<\text{lay}(\widehat{O}')}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}_2(j))$ to hold, it must be the case that $\text{lay}(\widehat{O}) > \text{lay}(\widehat{O}')$. This argument can be repeated ad-infinitum, by the well-foundedness of layers, a contradiction ensues, therefore $\neg \text{goodenv}_{\mathbb{S}}(\hat{\tau})$ holds. This implies $\text{goodenv}_{\mathbb{S}}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$.

From these cases, we deduce that $\forall \hat{\tau} \in \mathbb{T}. \text{goodenv}_{\mathbb{S}}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$.

From this, we can infer $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S} \rrbracket$ and consequently, $\llbracket \mathbb{C}_1 \parallel \mathbb{C}_2 \rrbracket_{\varphi} \subseteq \llbracket \mathbb{S} \rrbracket$, as required. \square

F.4 Soundness of WHILE

Definition F.10 (Concrete trace sequence operator).

$$\tau = \tau_1 \mathbin{\text{\textcircled{;}}} \tau_2 \Leftrightarrow \left(\begin{array}{l} (\neg \text{lterm}(\tau) \wedge \tau = \tau_1) \vee \\ \exists \sigma \in \text{Store}, h \in \text{Heap}, \tau'_1 \text{ loc } (\sigma, h)\tau''_1, (\sigma, h)\tau'_2 \in \text{Trace}. \\ \tau_1 = \tau'_1 \text{ loc } (\sigma, h)\tau''_1 \wedge \tau_2 = (\sigma, h)\tau'_2 \wedge \text{term}((\sigma, h)\tau''_1) \wedge \tau = \tau'_1 \text{ loc } (\sigma, h) \text{ loc } (\sigma, h)\tau'_2 \end{array} \right)$$

A similarly defined overloading of this operator exists for specification traces, $\hat{\tau}_1 \mathbin{\text{\textcircled{;}}} \hat{\tau}_2$ and the obvious lifting to sets $\mathbb{T}_1 \mathbin{\text{\textcircled{;}}} \mathbb{T}_2$.

LEMMA F.11. *For arbitrary φ , $(\sigma_0, h_0)\tau \in \llbracket \text{while}(\mathbb{B})\{\mathbb{C}\} \rrbracket_{\varphi}$, either $\neg \mathcal{B}[\mathbb{B}]_{\sigma_0}$, or there exists $(\sigma_0, h_0)\tau' \in \llbracket \mathbb{C} \rrbracket_{\varphi}$ and $\tau'' \in \llbracket \text{while}(\mathbb{B})\{\mathbb{C}\} \rrbracket_{\varphi}$, such that $(\sigma_0, h_0)\tau = (\sigma_0, h_0)\text{loc}(\sigma_0, h_0)\tau' \mathbin{\text{\textcircled{;}}} \tau''$.*

PROOF. Straightforward by induction on \Rightarrow_{φ} . \square

LEMMA F.12. *Given an arbitrary specification*

$$\mathbb{S} = \mathbb{W}x \in X \rightarrow X'. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{\lambda; \mathcal{A}}$$

for an arbitrary trace $(\sigma_0, h_0)\tau \in \text{Trace}$, let

$$p_h = \mathcal{W}[\llbracket P_h \rrbracket]_{\mathcal{A}}^{\sigma_0} \qquad p_a(v) = \mathcal{W}[\llbracket P_a(v) \rrbracket]_{\mathcal{A}}^{\sigma_0}$$

If for some $v \in X$ and $\mathbb{T} \in \mathcal{P}(\text{STrace})$, $h_0 \in \llbracket p_h * p_a(v) * \text{True} \rrbracket$ and $(\sigma_0, h_0)\tau \vDash_{\mathbb{S}} p_h, p_a, v : \mathbb{T}$, then:

$$\forall \hat{\tau} \in \mathbb{T}. \forall i \in \mathbb{N}. \text{term}(\hat{\tau}/i) \Rightarrow \begin{array}{l} \exists h \in \text{Heap}, \sigma \in \text{Store}, \quad \hat{\tau}(i) = (\sigma, h, p_h, \text{emp}, \langle v, v' \rangle) \wedge \\ p_h \in \text{View}_{\mathcal{A}}, v \in X, v' \in \text{AVal}. \quad p_h = \mathcal{W}[\llbracket Q_h(v, v') \rrbracket]_{\mathcal{A}}^{\sigma} \wedge h \in \llbracket p_h \rrbracket_{\lambda} \end{array}$$

PROOF. Straightforward by induction on the specification semantics rules. \square

Let

$$\mathbb{S}'(\beta, b) = \{P(\beta) * (b \doteq T(\beta)) \wedge \mathbb{B}\} \cdot \{\exists \gamma. P(\gamma) \wedge \gamma \leq \beta * (b \doteq \gamma < \beta)\}_{m;\lambda;\mathcal{A}}$$

$$\mathbb{S}(\beta_0) = \{P(\beta_0) * L\} \cdot \{\exists \beta. P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta_0 \geq \beta\}_{m;\lambda;\mathcal{A}}$$

LEMMA F.13. Take φ and β_0 arbitrary and take $(\sigma_0, h_0)\tau \in \llbracket \text{while}(\mathbb{B})\{\mathbb{C}\} \rrbracket_\varphi$ such that $\mathcal{B}[\mathbb{B}]_{\sigma_0}$. Let

$$\begin{aligned} p'(\beta, b) &= \mathcal{W}[\llbracket P(\beta) * (b \doteq T) \rrbracket_{\mathcal{A}}]_{\sigma_0}^{\sigma_0} \\ l &= \mathcal{W}[\llbracket L \rrbracket_{\mathcal{A}}]_{\sigma_0}^{\sigma_0} \end{aligned}$$

Clearly, $\forall \beta. p'(\beta, \text{True}) \subseteq p'(\beta, \text{False})$. As $\mathcal{B}[\mathbb{B}]_{\sigma_0}$, by lemma F.11, there exists $(\sigma_0, h_0)\tau' \in \llbracket \mathbb{C} \rrbracket_\varphi$ and $(\sigma_1, h_1)\tau'' \in \llbracket \text{while}(\mathbb{B})\{\mathbb{C}\} \rrbracket_\varphi$, such that $(\sigma_0, h_0)\tau = (\sigma_0, h_0)\tau' \ ; \ (\sigma_1, h_1)\tau''$. If, for arbitrary $\beta' \leq \beta \leq \beta_0$ and $\mathbb{T}', \mathbb{T}'' \in \mathcal{P}(\text{STrace})$, there exists $b \in \text{Bool}$, such that:

$$\begin{aligned} h_0 &\in \llbracket p'(\beta, b) * l \rrbracket_\lambda \\ (\sigma_0, h_0)\tau' &\vDash_{\mathbb{S}'(\beta, b)} p'(\beta, b), \text{emp}, 1 : \mathbb{T}' \\ (\sigma_1, h_1)\tau'' &\vDash_{\mathbb{S}(\beta')} p'(\beta', \text{False}) * l, \text{emp}, 1 : \mathbb{T}'' \\ b &\Leftrightarrow h_0 \in \llbracket p'(\beta, \text{True}) * l \rrbracket_\lambda \end{aligned}$$

then:

$$(\sigma_0, h_0)\tau \vDash_{\mathbb{S}(\beta, \text{False})} p'(\beta', \text{False}) * l, \text{emp}, 1 : \mathbb{T}' \ ; \ \mathbb{T}''$$

and one of the following hold:

$$\begin{aligned} &\text{lterm}((\sigma_0, h_0)\tau) \\ &\forall \hat{\tau} \in \mathbb{T}' \ ; \ \mathbb{T}'' . \neg \text{goodenv}_{\mathbb{S}}(\hat{\tau}) \\ &\forall \hat{\tau} \in \mathbb{T}' \ ; \ \mathbb{T}'' . \forall i \in \mathbb{N}. \exists j \geq i, \beta. \hat{\tau}(j) = (\sigma, h, p'(\beta, \text{False}), \text{emp}, 1) \end{aligned}$$

PROOF. This lemma is proven by coinduction on the structure of $(\sigma_0, h_0)\tau$. First, assume:

$$h_0 \in \llbracket p'(\beta, b) * l \rrbracket_\lambda \quad (42)$$

$$(\sigma_0, h_0)\tau' \vDash_{\mathbb{S}'(\beta, b)} p'(\beta, b), \text{emp}, 1 : \mathbb{T}' \quad (43)$$

$$(\sigma_1, h_1)\tau'' \vDash_{\mathbb{S}(\beta')} p'(\beta', \text{False}) * l, \text{emp}, 1 : \mathbb{T}'' \quad (44)$$

$$b \Leftrightarrow h_0 \in \llbracket p'(\beta, \text{True}) * l \rrbracket_\lambda \quad (45)$$

Using 42, 43 and 44, we can derive:

$$(\sigma_0, h_0)\tau \vDash_{\mathbb{S}(\beta, \text{False})} p'(\beta', \text{False}) * l, \text{emp}, 1 : \mathbb{T}' \ ; \ \mathbb{T}''$$

Now, split on $\text{lterm}((\sigma_0, h_0)\tau)$. If $\text{lterm}((\sigma_0, h_0)\tau)$, then the goal holds, otherwise, split again on $\text{lterm}((\sigma_0, h_0)\tau')$. If $\text{lterm}((\sigma_0, h_0)\tau')$, then $\mathbb{T}' \ ; \ \mathbb{T}'' = \mathbb{T}'$, so from 43, $\forall \hat{\tau} \in \mathbb{T}' \ ; \ \mathbb{T}'' . \text{goodenv}_{\mathbb{S}(\beta)}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$, therefore the goal holds. Otherwise, $\neg \text{lterm}((\sigma_1, h_1)\tau'')$. To not terminate, the while loop must iterate at least one more time, as $(\sigma_1, h_1)\tau''$ is a fair trace, therefore $\mathcal{B}[\mathbb{B}]_{\sigma_1}$ holds. We can then use lemma F.11 and our coinductive assumption to obtain:

$$h_1 \in \llbracket p'(\beta, b) * l \rrbracket_\lambda$$

and one of:

$$\forall \hat{\tau} \in \mathbb{T}'' . \neg \text{goodenv}_{\mathbb{S}(\beta')}(\hat{\tau})$$

$$\forall \hat{\tau} \in \mathbb{T}'' . \forall i \in \mathbb{N}. \exists j \geq i, \beta. \hat{\tau}(j) = (\sigma, h, p'(\beta, \text{False}), \text{emp}, 1)$$

holds. In the first case $\forall \hat{\tau} \in \mathbb{T}' \ ; \ \mathbb{T}''$. $\text{goodenv}_{\mathbb{S}(\beta)}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$, so the goal holds as required and in the second, from $h_1 \in \llbracket p'(\beta, b) * l \rrbracket_\lambda$:

$$\forall \hat{\tau} \in \mathbb{T}' \ ; \ \mathbb{T}'' . \forall i \in \mathbb{N} . \exists j \geq i . \beta . \hat{\tau}(j) = (\sigma, h, p'(\beta, \text{False}), \text{emp}, 1)$$

This implies the goal, as required. \square

THEOREM F.14. *Given*

$$\forall \beta \leq \beta_0 . \forall b \in \{0, 1\} . m; \lambda; \mathcal{A} \models \{P(\beta) * (b \Rightarrow T) \wedge \mathbb{B}\} \mathbb{C} \{\exists \gamma . P(\gamma) \wedge \gamma \leq \beta * (b \Rightarrow \gamma < \beta)\} \quad (46)$$

$$\forall \beta \leq \beta_0 . m(\beta); \lambda; \mathcal{A} \models L \xrightarrow{M} T \quad (47)$$

$$\forall \alpha . \mathcal{A} \models \exists \alpha' . L * M(\alpha') \wedge \alpha' \leq \alpha \text{ stable} \quad (48)$$

$$\mathcal{A} \models L \text{ stable} \quad (49)$$

$$\forall \beta \leq \beta_0 . \vdash_{\mathcal{A}} P(\beta) \triangleright m(\beta) \geq m \quad (50)$$

$$\text{pv}(T, L, M) \cap \text{mod}(\mathbb{C}) = \emptyset \quad (51)$$

then:

$$m; \lambda; \mathcal{A} \models \{P(\beta_0) * L\} \text{ while}(\mathbb{B})\{\mathbb{C}\} \{\exists \beta . P(\beta) * L \wedge \neg \mathbb{B} \wedge \beta_0 \geq \beta\}$$

PROOF. Take $\varphi \in \mathcal{F}_{\text{fin}(\mathbb{C})}[\llbracket \Phi \rrbracket]$ arbitrary. Take $(\sigma_0, h_0)\tau \in \llbracket \text{while}(\mathbb{B})\{\mathbb{C}\} \rrbracket_{\varphi}$, need to show $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S}(\beta_0) \rrbracket$. Let

$$\begin{aligned} p'(\beta, b) &= \mathcal{W}[\llbracket P(\beta) * (b \Rightarrow T) \rrbracket]_{\mathcal{A}}^{\sigma_0} \\ l &= \mathcal{W}[\llbracket L \rrbracket]_{\mathcal{A}}^{\sigma_0} \end{aligned}$$

To reach the goal, assume $h_0 \in \llbracket p'(\beta_0, \text{False}) * l \rrbracket_\lambda$. By lemma F.13, in the case that $\mathcal{B}[\llbracket \mathbb{B} \rrbracket]_{\sigma_0}$, and our assumptions, there exists \mathbb{T} :

$$(\sigma_0, h_0)\tau \models_{\mathbb{S}} \mathcal{W}[\llbracket P(\beta_0) * L \rrbracket]_{\mathcal{A}}^{\sigma_0}, \text{emp}, 1 : \mathbb{T}$$

and one of the following hold:

$$\text{lterm}((\sigma_0, h_0)\tau)$$

$$\forall \hat{\tau} \in \mathbb{T} . \neg \text{goodenv}_{\mathbb{S}}(\hat{\tau})$$

$$\forall \hat{\tau} \in \mathbb{T} . \forall i \in \mathbb{N} . \exists j \geq i . \beta . \hat{\tau}(j) = (\sigma, h, p'(\beta, \text{False}), \text{emp}, 1)$$

In the first case, $\forall \hat{\tau} \in \mathbb{T} . \text{lterm}(\hat{\tau})$, therefore, $\forall \hat{\tau} \in \mathbb{T} . \text{goodenv}_{\mathbb{S}(\beta_0)}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$, as required. In the second, $\forall \hat{\tau} \in \mathbb{T} . \text{goodenv}_{\mathbb{S}(\beta_0)}(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau})$ clearly also holds. Finally, we consider the third case. Take $\hat{\tau} \in \mathbb{T}$ arbitrary and assume $\text{goodenv}_{\mathbb{S}(\beta_0)}(\hat{\tau})$. Now, for a contradiction, assume $\neg \text{lterm}(\hat{\tau})$. In this case, due to 47, with an argument similar to that in the soundness of **LIVEC**, at every point, every $\hat{\tau} \in \mathbb{T}$ eventually reaches a state satisfying T . This must eventually be stable due to the metric stably decreasing due to assumption 48, holding till the next iteration, at which point, the loop variant decreases due to 46 with $b = \text{True}$. In this case, by well-foundedness of ordinals, the while loop must eventually terminate if $\text{goodenv}(\hat{\tau})$ holds, as required. \square

F.5 Soundness of **FRAME**

Definition F.15 ($\hat{\tau}$ -unframe $_{r_h, r_a}$).

$$\hat{\tau}\text{-unframe}_{r_h, r_a}((\sigma, h, p_h, p_a, v) : \hat{\tau}) \triangleq \begin{cases} (\sigma, h, p_h * r_h, p_a * r_h, v) : \hat{\tau}\text{-unframe}_{r_h, r_a}(\hat{\tau}) & v \in \text{AVal} \\ (\sigma, h, p_h * r_h, \text{emp}, v) : \hat{\tau}\text{-unframe}_{r_h, r_a}(\hat{\tau}) & v = \langle _ , _ \rangle \wedge \forall v \in \text{AVal} . p_a(v) = \text{emp} \end{cases}$$

Let

$$\mathbb{S} = \forall x \in \vec{X}. \langle P_h \mid P_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) \mid Q_a(x, y) \rangle_{m; \lambda; \mathcal{A}}$$

$$\mathbb{S}' = \forall x \in \vec{X}. \langle P_h * R_h \mid P_a(x) * R_a(x) \rangle \cdot \exists y. \langle Q_h(x, y) * R_h \mid Q_a(x, y) * R_a(x) \rangle_{m; \lambda; \mathcal{A}}$$

LEMMA F.16. For arbitrary $\lambda \in \text{Lvl}$, \mathcal{A} and atomicity context, $h_0, h_1 \in \text{Heap}$, $p, q \in \mathcal{P}(\text{World}_{\mathcal{A}})$ and $r \in \text{View}_{\mathcal{A}}$:

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p \rightarrow q \Rightarrow (h_0, h_1) \vDash_{\lambda; \mathcal{A}} p * r \rightarrow q * r$$

PROOF. Assume $(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p \rightarrow q$, which is equivalent:

$$\forall f \in \text{View}_{\mathcal{A}}. h_0 \in \llbracket p * f \rrbracket \Rightarrow h_1 \in \llbracket q * f \rrbracket$$

$$\forall f \in \text{View}_{\mathcal{A}}. h \in \llbracket (p * f) \cap W_1 \rrbracket_{\lambda} \Rightarrow \exists a_2, O_2. ((a_1, O_1), (a_2, O_2)) \in \mathcal{T}_t \wedge h' \in \llbracket (q * f) \cap W_2 \rrbracket_{\lambda}$$

for all $\mathbf{t} \in \text{RType}$, $r \in \text{Rld}$, $\lambda' \in \text{Lvl}$, $a_1 \in \text{AState}$, $O_1 \in \mathcal{O}_t$, where

$W_i \triangleq \{w \in \text{World}_{\mathcal{A}} \mid \rho_w(r) = (\mathbf{t}, \lambda', a_i), \theta_w(r) = O_i\}$. Both of the assertions apply substituting $f = r * f'$ for arbitrary f' , from which $h_0 \dashv > h_1 \mid = p * r \dashv > q * r$ follows. \square

LEMMA F.17. For arbitrary $\lambda \in \text{Lvl}$, \mathcal{A} and atomicity context, $h_0, h_1 \in \text{Heap}$ and $p, q \in \mathcal{P}(\text{World}_{\mathcal{A}})$:

$$h_0 \in \llbracket p * \text{True} \rrbracket \wedge (h_0, h_1) \vDash_{\lambda; \mathcal{A}} p \rightarrow q \Rightarrow h_1 \in \llbracket q * \text{True} \rrbracket$$

PROOF. To start off, assume:

$$h_0 \in \llbracket p * \text{True} \rrbracket$$

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p \rightarrow q$$

Clearly, this second assumption entails $(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p \dashv^* q$, which is equivalent to:

$$\forall f \in \text{View}_{\mathcal{A}}. h_0 \in \llbracket p * f \rrbracket \Rightarrow h_1 \in \llbracket q * f \rrbracket$$

Choosing f to be True and applying the first assumption yields $h_1 \in \llbracket q * \text{True} \rrbracket$ as required. \square

LEMMA F.18. For arbitrary $(\sigma_0, h_0)\tau \in \text{Trace}$, $p_h, r_h \in \text{View}_{\mathcal{A}}$, $v, v' \in \text{AVal}'$ and $\mathbb{T} \in \mathcal{P}(\text{STrace})$, then

$$h_0 \in \llbracket p_h * r_h * \text{True} \rrbracket \wedge (\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : \mathbb{T} \Rightarrow$$

$$(\sigma_0, h_0) \tau \vDash_{\mathbb{S}'} p_h * r_h, \text{emp}, \langle v, v' \rangle : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T})$$

holds.

PROOF. Taking $(\sigma_0, h_0)\tau \in \text{Trace}$, $p_h, r_h \in \text{View}_{\mathcal{A}}$, $v, v' \in \text{AVal}'$ and $\mathbb{T} \in \mathcal{P}(\text{STrace})$ arbitrary, to start off, assume:

$$h_0 \in \llbracket p_h * r_h * \text{True} \rrbracket \tag{52}$$

$$(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : \mathbb{T} \tag{53}$$

The proof proceeds by coinduction on the structure of τ . Only three rules can apply from the trace safety judgement: **STUTTER**, **ENV'** and **ENV_z**.

– *Case STUTTER*. In this case, $(\sigma_0, h_0)\tau = (\sigma_0, h_0) \text{ loc } (\sigma_1, h_1)\tau'$ and $\mathbb{T} = (\sigma_0, h_0, p_h, \text{emp}, \langle v, v' \rangle) \text{ loc } \mathbb{T}'$. From 53, the following hold:

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h \rightarrow p'_h \quad (54)$$

$$(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}} p'_h, \text{emp}, \langle v, v' \rangle : \mathbb{T}' \quad (55)$$

$$\text{term}(\tau') \Rightarrow p'_h = \mathcal{W}[\llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_1}] \quad (56)$$

Given that $r_h \in \text{View}_{\mathcal{A}}$, using lemma F.16, (54) implies $(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h * r_h \rightarrow p'_h * r_h$, this in turn implies $h_1 \in \llbracket p'_h * r_h * \text{True} \rrbracket$, using lemma F.16. Using the inductive assumption, (55) implies $(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}'} p'_h * r_h, \text{emp}, \langle v, v' \rangle : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}')$. Finally, assuming $\text{term}(\tau')$, $p'_h = \mathcal{W}[\llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_1}]$ holds, therefore $p'_h * r_h = \mathcal{W}[\llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_1}] * \mathcal{W}[\llbracket R_h \rrbracket_{\mathcal{A}}^{\sigma_1}] = \mathcal{W}[\llbracket Q_h(v, v') * R_h \rrbracket_{\mathcal{A}}^{\sigma_1}]$ holds, as required.

– *Case ENV'*. In this case, $(\sigma_0, h_0)\tau = (\sigma_0, h_0) \text{ env } (\sigma_1, h_1)\tau'$ and $\mathbb{T} = (\sigma_0, h_0, p_h, \text{emp}, \langle v, v' \rangle) \text{ env } \mathbb{T}'$. From 53, the following hold:

if $\exists p_e, p'_e. h_0 \in \llbracket p_h * p_e \rrbracket_{\lambda} \wedge (h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_e \rightarrow p'_e$ **then** $(\sigma_1, h_1) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : \mathbb{T}'$ **else** $\mathbb{T}' = \emptyset$

To reach the goal, assume for some $\bar{p}_e, \bar{p}'_e \in \text{View}_{\mathcal{A}}$, that

$$\begin{aligned} h_0 &\in \llbracket p_h * r_h * \bar{p}_e \rrbracket_{\lambda} \\ (h_0, h_1) &\vDash_{\lambda; \mathcal{A}} \bar{p}_e \rightarrow \bar{p}'_e \end{aligned}$$

From these assumptions, we can infer that $h_1 \in \llbracket p_h * r_h * \text{True} \rrbracket$, using lemmas F.16 and F.17. Setting $p_e = r_h * \bar{p}_e$ and $p'_e = r_h * \bar{p}'_e$:

$$h_0 \in \llbracket p_h * p_e \rrbracket_{\lambda}$$

and as $r_h \in \text{View}_{\mathcal{A}}$:

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_e \rightarrow p'_e$$

holds. Given these results and the assumptions, $(\sigma_1, h_1) \tau \vDash_{\mathbb{S}} p_h, \text{emp}, \langle v, v' \rangle : \mathbb{T}'$ holds, yielding $(\sigma_1, h_1) \tau \vDash_{\mathbb{S}'} p_h * r_h, \text{emp}, \langle v, v' \rangle : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}')$ by coinductive assumption.

– *Case ENV_‡*. This case is trivially true. □

LEMMA F.19. For arbitrary $(\sigma_0, h_0)\tau \in \text{Trace}$, $p_h, r_h \in \text{View}_{\mathcal{A}}$, $p_a, r_a \in \text{AVal} \rightarrow \text{View}_{\mathcal{A}}$, $v_0 \in \text{AVal}$ and $\mathbb{T} \in \mathcal{P}(\text{STrace})$, then

$$\begin{aligned} h_0 \in \llbracket p_h * r_h * p_a(v_0) * r_a(v_0) * \text{True} \rrbracket \wedge (\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T} \Rightarrow \\ (\sigma_0, h_0) \tau \vDash_{\mathbb{S}'} p_h * r_h, p_a * r_a, v_0 : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}) \end{aligned}$$

holds.

PROOF. Taking $(\sigma_0, h_0)\tau \in \text{Trace}$, $p_h, r_h \in \text{View}_{\mathcal{A}}$, $p_a, r_a \in \text{AVal} \rightarrow \text{View}_{\mathcal{A}}$, $v_0 \in \text{AVal}'$ and $\mathbb{T} \in \mathcal{P}(\text{STrace})$ arbitrary, too start off, assume:

$$h_0 \in \llbracket p_h * r_h * p_a(v_0) * r_a(v_0) * \text{True} \rrbracket \quad (57)$$

$$(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T} \quad (58)$$

The proof proceeds by coinduction on the structure of τ . Only four rules can apply from the trace safety judgement: *STUTTER*, *LINPT*, *ENV* and *ENV_‡*.

– *Case STUTTER*. In this case, $(\sigma_0, h_0)\tau = (\sigma_0, h_0) \text{ loc } (\sigma_1, h_1)\tau'$ and $\mathbb{T} = (\sigma_0, h_0, p_h, p_a, v) \text{ loc } \mathbb{T}'$. From 58, the following hold:

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h * p_a(v) \rightarrow p'_h * p_a(v) \quad (59)$$

$$(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}} p'_h, p_a, v : \mathbb{T}' \quad (60)$$

$$\neg \text{term}(\tau') \quad (61)$$

Given that $r_h, r_a(v_0) \in \text{View}_{\mathcal{A}}$, using lemma F.16, (59) implies $(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h * r_h * p_a(v) * r_a(v) \rightarrow p'_h * r_h * p_a(v) * r_a(v)$, this in turn implies $h_1 \in \llbracket p'_h * r_h * p_a(v) * r_a(v) * \text{True} \rrbracket$ using lemma F.17. Using the inductive assumption, (60) implies $(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}'} p'_h * r_h, p_a * r_a, v : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}')$. Finally, given that $\neg \text{term}(\tau')$ holds, $\text{term}(\tau') \Rightarrow p'_h * r_h = \mathcal{W} \llbracket Q_h(v, v') * R_h \rrbracket_{\mathcal{A}}^{\sigma_1}$ holds as required.

– *Case LINPT*. In this case, $(\sigma_0, h_0)\tau = (\sigma_0, h_0) \text{ loc } (\sigma_1, h_1)\tau'$ and $\mathbb{T} = (\sigma_0, h_0, p_h, p_a, v) \text{ loc } \mathbb{T}'$. From 58, the following hold:

$$(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h * p_a(v) \rightarrow q'_h * \mathcal{W} \llbracket Q_a(v, v') \rrbracket_{\mathcal{A}} \quad (62)$$

$$(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}} q'_h, \text{emp}, \langle v, v' \rangle : \mathbb{T}' \quad (63)$$

$$\text{term}(\tau') \Rightarrow q'_h = \mathcal{W} \llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_2} \quad (64)$$

Given that $r_h, r_a(v_0) \in \text{View}_{\mathcal{A}}$, using lemma F.16, (62) implies $(h_0, h_1) \vDash_{\lambda; \mathcal{A}} p_h * r_h * p_a(v) * r_a(v) \rightarrow q'_h * r_h * \mathcal{W} \llbracket Q_a(v, v') \rrbracket_{\mathcal{A}} * r_a(v)$, this in turn implies $h_1 \in \llbracket q'_h * r_h * \mathcal{W} \llbracket Q_a(v, v') \rrbracket_{\mathcal{A}} * r_a(v) * \text{True} \rrbracket$ using lemma F.17. Using lemma F.18, (63) implies $(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}'} q'_h * r_h, \text{emp}, \langle v, v' \rangle : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}')$. Finally, assuming $\text{term}(\tau')$, $q'_h = \mathcal{W} \llbracket Q_h(v, v') \rrbracket_{\mathcal{A}}^{\sigma_2}$ holds, therefore $\text{term}(\tau') \Rightarrow q'_h * r_h = \mathcal{W} \llbracket Q_h(v, v') * R_h \rrbracket_{\mathcal{A}}^{\sigma_1}$ holds as required.

– *case ENV*. In this case, $(\sigma_0, h_0)\tau = (\sigma_0, h_0) \text{ env } (\sigma_1, h_1)\tau'$ and $\mathbb{T} = \bigcup \{ (\sigma, h_1, p_h, p_a, v) \text{ env } \mathbb{T}'_{v'} \mid v' \in X, E(v') \}$. From 58, the following holds:

$$\forall v' \in X. E(v') \Rightarrow (\sigma, h_2) \tau \vDash_{\mathbb{S}} p_h, p_a, v' : \mathbb{T}_{v'}$$

Taking $v' \in X$ arbitrary and, assuming $E(v')$ given some $\overline{p_e}, \overline{p'_e}$, for the goal specification:

$$h_1 \in \llbracket p_h * r_h * p_a(v) * r_a(v) * \overline{p_e} \rrbracket_{\lambda}$$

$$(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_a(v) * r_a(v) * \overline{p_e} \rightarrow p_a(v') * r_a(v') * \overline{p'_e}$$

, it suffices to show that $(\sigma_1, h_1) \tau' \vDash_{\mathbb{S}'} p_h * r_h, p_a * r_a, v' : \mathbb{T}_{v'}$ holds. This follows from lemma F.16 and the substitution:

$$p_e = \overline{p_e} * r_h * r_a(v)$$

$$p'_e = \overline{p'_e} * r_h * r_a(v')$$

yielding:

$$h_1 \in \llbracket p_h * p_a(v) * p_e \rrbracket_{\lambda}$$

$$(h_1, h_2) \vDash_{\lambda; \mathcal{A}} p_a(v) * p_e \rightarrow p_a(v') * p'_e$$

as required.

– Case ENV_4^i . This case is trivially true. \square

THEOREM F.20 (SOUNDNESS OF FRAME). *If*

$$\vdash_{\mathcal{A}} R_h * R_a(x) \triangleright m \quad (65)$$

$$\mathcal{A} \models R_h \text{ stable} \quad (66)$$

$$\forall x \in X. \mathcal{A} \models R_a(x) \text{ stable} \quad (67)$$

$$\text{fv}(R_h, R_a(x)) \cap \text{mod}(\mathbb{C}) = \emptyset \quad (68)$$

then:

$$\llbracket \mathbb{S} \rrbracket \subseteq \llbracket \mathbb{S}' \rrbracket$$

PROOF. To start off, as $\mathcal{A} \models R_h$ stable, clearly $P_h * R_h \in \text{Stable}_{\mathcal{A}}$ and therefore, $\mathbb{S}' \in \text{Spec}$.

Take $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S} \rrbracket$ arbitrary, sufficient to show $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S}' \rrbracket$.

Let

$$\begin{aligned} p_h &= \mathcal{W} \llbracket P_h \rrbracket_{\mathcal{A}}^{\sigma_0} \\ r_h &= \mathcal{W} \llbracket R_h \rrbracket_{\mathcal{A}}^{\sigma_0} \\ p_a(v) &= \mathcal{W} \llbracket P_a(v) \rrbracket_{\mathcal{A}} \\ r_a(v) &= \mathcal{W} \llbracket R_a(v) \rrbracket_{\mathcal{A}} \end{aligned}$$

To show $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S}' \rrbracket$, for some arbitrary $v_0 \in X$, assume $h_0 \in \llbracket p_h * r_h * p_a(v_0) * r_a(v_0) * \text{True} \rrbracket_{\lambda}$. Then, clearly $h_0 \in \llbracket p_h * p_a(v_0) * \text{True} \rrbracket_{\lambda}$. Then, as $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S} \rrbracket$, for some $\mathbb{T} \subseteq \text{STrace}$:

$$(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h, p_a, v_0 : \mathbb{T} \quad (69)$$

$$\forall \hat{\tau} \in \mathbb{T}. \text{goodenv}_m(\hat{\tau}) \Rightarrow \text{lterm}(\hat{\tau}) \quad (70)$$

From F.19 and (69), $(\sigma_0, h_0) \tau \vDash_{\mathbb{S}} p_h * r_h, p_a * r_a, v_0 : \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T})$. To reach the goal now, it suffices to show that for some arbitrary $\hat{\tau}' \in \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T})$:

$$\text{goodenv}_m(\hat{\tau}') \Rightarrow \text{lterm}(\hat{\tau}')$$

To reach this goal, assume $\neg \text{lterm}(\hat{\tau}')$. Clearly, from the definition of $\hat{\tau}\text{-unframe}_{r_h, r_a}$, this implies $\neg \text{lterm}(\hat{\tau})$, so from (70), $\neg \text{goodenv}_m(\hat{\tau})$, which is equivalent to:

$$\forall O \in \text{AOB}_{<\text{lay}(\widehat{O})}. \forall i \in \mathbb{N}. \exists j \geq i. \neg \text{locheld}(O, \hat{\tau}(j)) \quad (71)$$

$$\exists i \in \mathbb{N}. \forall j \geq i. \text{envheld}(\widehat{O}, \hat{\tau}(j)) \quad (72)$$

for some $\widehat{O} \in \text{POb}_{<m}$.

To reach the goal, it is sufficient to show that the same holds for $\hat{\tau}'$. To show (71) holds for $\hat{\tau}'$, take $O \in \text{AOB}_{<\text{lay}(\widehat{O})}$ and $i \in \mathbb{N}$ arbitrary. Using (71), there is some $j \geq i$ such that $\neg \text{locheld}(O, \hat{\tau}(j))$, which implies:

$$\forall w_h, w_a, r. \neg(\hat{\tau}(j) \prec (w_h, w_a)) \vee \theta_{w_h}(r) \not\subseteq O$$

Let $\hat{\tau}(j) = (\sigma_j, h_j, p_h^j, p_a^j, v_j)$, clearly, if for some arbitrary w_h, w_a and r , $\hat{\tau}'(j) \prec (w_h, w_a)$, then there exists some w'_h and wr'_h such that:

$$\begin{aligned} w_h &= w'_h \odot wr'_h \\ w'_h &\in p_h^j \\ wr'_h &\in r_h \end{aligned}$$

and $\hat{\tau}(j) \prec (w'_h, w'_a)$, therefore, from above, $\theta_{w'_h}(r) \not\sqsupseteq O$. Assume $\theta_{w'_h}(r) \sqsupseteq O$ for a contradiction. As O is an atom, $\theta_{wr'_h}(r) \sqsupseteq O$ holds, which in turn implies that $\text{lay}(O) \geq \text{lay}(\theta_{wr'_h}(r))$. As $\text{lay}(O) < \text{lay}(\widehat{O}) < m$, this forms a contradiction with (65), therefore $\theta_{w'_h}(r) \sqsupseteq O$ holds, which is sufficient to show $\neg \text{locheld}(O, \hat{\tau}(j))$, as required.

To show (72) for $\hat{\tau}'$, given (72) for some $i \in \mathbb{N}$, taking $j \geq i$ arbitrary, $\text{envheld}(\widehat{O}, \hat{\tau}(j))$ holds. This breaks down into three cases depending on \widehat{O} :

– $O \in \text{Oblig}$. In this case, $\text{envheld}(\widehat{O}, \hat{\tau}(j))$ is equivalent to:

$$\forall w_h, w_a. \hat{\tau}(j) \prec (w_h, w_a) \Rightarrow \exists r. \xi_{w_h \odot w_a}(r) \sqsupseteq \widehat{O}$$

Let $\hat{\tau}(j) = (\sigma_j, h_j, p_h^j, p_a^j, v_j)$, to show $\text{envheld}(\widehat{O}, \hat{\tau}(j))$, take w_h, w_a such that $\hat{\tau}(j) \prec (w_h, w_a)$. As above, we can split these into w'_h, wr'_h, w'_a and wr'_a such that:

$$\begin{array}{llll} w_h = w'_h \odot wr'_h & w_a = w'_a \odot wr'_a & & \\ w'_h \in p_h^j & wr'_h \in r_h & w'_a \in p_a^j(v_j) & wr'_a \in r_a(v_j) \end{array}$$

From $\text{envheld}(\widehat{O}, \hat{\tau}(j))$, $\xi_{w'_h \odot w'_a}(r) \sqsupseteq \widehat{O}$ holds. From the definition of \odot , $\xi_{w'_h \odot wr'_h \odot w'_a \odot wr'_a}(r) \bullet \theta_{wr'_h}(r) \bullet \theta_{wr'_a}(r) = \xi_{w'_h \odot w'_a}(r)$. Assuming $\xi_{w'_h \odot wr'_h \odot w'_a \odot wr'_a}(r) \sqsupseteq \widehat{O}$ for a contradiction, as \widehat{O} is an atom, $\theta_{wr'_h}(r) \bullet \theta_{wr'_a}(r) \sqsupseteq \widehat{O}$, which forms a contradiction with (65). This is sufficient to show $\text{envheld}(\widehat{O}, \hat{\tau}(j))$, as required.

– $O = (r, X_1 \rightarrow_k X_2)$, $O = X_1 \rightarrow_k X_2$. These cases are trivial as the frame extension does not change the state of regions or the value of the pseudo-quantified variable.

From this, (72) and (71) for $\hat{\tau}'$. Therefore $\neg \text{goodenv}_m(\hat{\tau}')$ holds, as required. From this we infer that:

$$\forall \hat{\tau}' \in \hat{\tau}\text{-unframe}_{r_h, r_a}(\mathbb{T}). \text{goodenv}_m(\hat{\tau}') \Rightarrow \text{lterm}(\hat{\tau}')$$

This suffices to prove $(\sigma_0, h_0)\tau \in \llbracket \mathbb{S}' \rrbracket$, as required. \square

G 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.

G.1 Specifications

LiLi is based on proving refinements, so specifications are themselves programs. To specify an abstractly atomic blocking operation with an atomic specification, LiLi introduces a primitive-blocking specification construct: $\text{await}(\mathbb{B})\{\mathbb{C}\}$ executes \mathbb{C} atomically if scheduled in a state where \mathbb{B} is true. As both fair- and spin-lock acquisition are specified using $\text{await}(l = 0)\{l := 1\}$ 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 $\text{await}(\mathbb{B})\{\mathbb{C}\}_{\text{SF/DF}}$, 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 $(\text{await}(l = 0)\{l := 1\})_{\text{SF}}$ 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 enqueueing 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 \rightarrow) and bounded impedance assumptions (through 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 awkward 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(B){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.

G.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".¹³ 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 $\text{lock}(x1)$ 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 ω 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 x_i is associated with an obligation \mathbf{R}_i to release it and the two obligations are declared dependent using the layers: $\text{lay}(\mathbf{R}_1) > \text{lay}(\mathbf{R}_2)$. 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 \mathbf{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.

¹³Note 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.