TaDA: Technical Report

March 5, 2014

Abstract

To avoid data races, concurrent operations should either be at distinct times or on distinct data. *Atomicity* is the abstraction that an operation takes effect at a single, discrete instant in time, with linearisability being a well-known correctness condition which asserts that concurrent operations appear to behave atomically. *Disjointness* is the abstraction that operations act on distinct data resource, with concurrent separation logics enabling reasoning about threads that appear to operate independently on disjoint resources.

Building on separation logic with concurrent abstract predicates (CAP), we introduce TaDA, a program logic which combines the benefits of abstract atomicity and abstract disjointness. We give several examples: an atomic lock module, a CAP example with a twist, which cannot be done using linearisability; an atomic MCAS module implemented using our lock module, a classic linearisability example which cannot be done using CAP; and a double-ended queue module implemented using MCAS.

1 Introduction

In a concurrent program, a data race occurs when two threads update the same data at the same time. Data races often have poor or undefined semantics, and therefore should be avoided. To avoid data races, updates should either be at distinct times or on disjoint data.

Atomicity is the abstraction that an operation, such as a data update, takes effect at a single, discrete instant in time. Concurrent atomic updates to the same data are not data races, since they *effectively* occur at distinct times. *Linearisability* [10] is a correctness condition which asserts that the operations of a concurrent module appear to behave atomically. Various techniques have been used to prove linearisability for concurrent modules such as queues [10] and lists with fine-grained synchronisation [21].

Linearisability works well: clients of a linearisable module can be analysed with the assumption that the module operations are atomic. For instance, a client could make use of a sophisticated concurrent skip list as if its operations were simple atomic operations on a set data structure. This fiction of atomicity is fragile, however. The addition of new operations may break the linearisability of the module as a whole. Consequently, a client of a linearisable module should only use the operations provided by that module. Combining and extending modules is non-trivial. For example, a client that uses *heap* operations to dereference pointers obtained from a *list* module may be able to observe non-atomicity of list operations. Although the heap module and list module are individually linearisable, their combination is not.

Disjointness is the abstraction that operations act on distinct data resources. By treating the data as resource, disjointness can ensure that data races do not occur. *Separation logics* for concurrency (both the original CSL [13] and subsequent developments [20, 5, 4, 17, 16]) embody this principle. With separation logic, we can reason about threads that operate on disjoint resources *independently*, and combine overall effects. When threads share resources, the operations on these shared resources *must be atomic*, and they must preserve some form of invariant property.

Concurrent abstract predicates (CAP) [4] give the benefit of disjoint reasoning despite sharing, achieved by abstracting underlying invariant properties to present a *fiction of disjointness* for concurrent modules. Operations on *shared* resources can thus be specified as if they operate on *disjoint* resources. Such reasoning has been applied to, for example, locks [4] and concurrent indexes [2].

CAP works well: when threads use the abstract resources disjointly, they can be analysed independently. However, if we want to share the resources between threads, we have a problem: with the CAP approach, shared resources may only be accessed by *primitive atomic* operations. The operations provided by concurrent modules, however, are rarely primitive atomic.

Linearisability and CAP have complementary virtues and weaknesses. Linearisability gives strong specifications in the form of abstract atomicity, but requires isolation between clients and modules to preserve its fragility; typically, it is not possible for resources to be transferred or shared between a module and a client. CAP on the other hand is good at handling transfer and sharing of resources, and can exploit atomicity, but tends to produce weaker specifications; such specifications can be ad hoc and require convoluted proofs.

Concurrent indexes, for example, can have simple CAP specifications that support elegant disjoint reasoning for clients, abstracting the interference between threads [2]. However, clients are constrained to follow some protocol imposed by the specification (for instance, to only remove certain keys from the index). These specifications are relatively simple to prove for an atomic implementation. For a more concrete implementation (such as a B-tree), proving such specifications requires dealing with the client's protocol at the level of the module implementation. This leads to numerous cases in the proof, which must be handled separately. It would be more convenient to prove a simpler abstract-atomic specification, and use this to derive (independently of the implementation) the appropriate abstract-disjoint specification needed by the client.

We introduce TaDA, a logic for Time and Data Abstraction, which combines the benefits of abstract atomicity and abstract disjointness in one program logic. In TaDA, the atomicity of an operation is defined with respect to the data abstraction used in its specification. The logic allows us to build up levels of abstraction. At one level, an operation on a concurrent data structure is seen as multiple atomic accesses of shared memory; at a higher level, the memory is abstracted so that the operation appears to occur at a single one of the underlying atomic accesses. This gives a flexible approach to modularity: a module can be straightforwardly extended with any other atomic operations that use the same abstraction, without breaking the fiction of atomicity. Atomic and non-atomic operations may even safely coexist, with their specifications constraining how a client may use them and expect reasonable interactions.

Where a client uses a module by its abstract specification alone, it is, of course, straightforward to substitute any valid implementation.

TaDA extends the Hoare logic of CAP [4] (including some features from iCAP [16]) with an *abstract-atomic* judgement and proof rules for deriving and using these judgements. The simplest form of the judgement is

$$\vdash \langle p \rangle \; \mathbb{C} \; \langle q \rangle$$

where p and q are separation-logic-style assertions and $\mathbb{C}$ is a program. This judgement is read as "$\mathbb{C}$ atomically updates p to q". The program may actually take multiple steps, but each step before the update from p to q must preserve the assertion p. Before the atomic update occurs, the concurrent environment may also update the state, provided that the assertion p is preserved. As soon as the atomic update has been done, the environment can do what it likes. The concurrent environment is not constrained to maintain q, and the program $\mathbb{C}$ may no longer access the resources of q.

The atomicity of $\mathbb{C}$ is *only* with respect to the abstraction defined by p. If the environment makes an observation at a lower level of abstraction, it may perceive multiple updates rather than this single atomic update. For example, suppose that a set module, which provides an atomic remove operation, is implemented using a linked list. The implementation might first mark a node as deleted, before removing it from the list. The environment can observe the change from "marked" to "removed". This low-level step does not change the abstract set; the change already occurred when the node was marked.

We illustrate our reasoning using several examples. First we give an atomic specification for a lock. This specification implicitly imposes constraints on a client, which linearisability typically does not do. We show how the non-atomic CAP specification can be derived from our atomic specification. This strategy, of deriving the strongest atomic specification then weakening it for the client, simplifies proofs. For example, it drastically cuts the case analysis required to prove that a concurrent B-tree algorithm is correct with respect to the concurrent index specification [2]. We also give an atomic specification of a library implementing multiple-compare-and-swap (MCAS) operations, showing that an implementation based on our lock module is correct. The example is straightforwardly linearisable, but is not amenable to a meaningful specification in CAP. However, we also extend the module with an operation that involves resource transfer in the context of an atomic operation, which is not possible with either CAP or linearisability alone. Finally, we specify a double-ended queue (deque), showing that a fine-grained implementation using MCAS is correct.

In TaDA, we have a logic that combines the benefits of abstract atomicity and abstract disjointness, and in doing so goes further than linearisability or CAP alone. Within a single logical framework, we can build up implementations that mix both flavours of abstraction. TaDA's approach allows subtle and expressive atomic specifications, which are nevertheless simple to use.

1.1 Related Work

TaDA inherits from a family of logics deriving from concurrent separation logic [13]: RGSep [20], Deny-Guarantee [5], CAP [4], Higher-Order CAP (HO-CAP) [17] and Impredicative CAP (iCAP) [16]. In particular, it makes use of dynamic *shared regions* with capability resources (called *guards* in TaDA) that determine how the regions may be updated. Following iCAP, TaDA eschews the use of boxed assertions to describe the state of shared regions and instead represents regions by *abstract states*. The protocol for updating the region is specified as a transition system on these abstract states, labelled by guards. This use of transition systems to describe protocols derives from previous work by Dreyer *et al.* [7], and also appears in Turon *et al.* [19] as "local life stories".

By treating the abstract state-space of a region as a separation algebra, it is possible to localise updates on it. We use this approach to implement a double compare-and-swap operation (§2.2). Such locality is in the spirit of local life stories, and can be seen as an instance of Ley-Wild and Nanevski's "subjective auxiliary state" [12].

While HOCAP and iCAP do not support abstract atomic specifications, they support an approach to atomicity introduced by Jacobs and Piessens [11] that achieves similar effects. In their work, operations may be parametrised by an update to auxiliary state that is performed when the abstract atomic operation appears to take effect. This update is performed atomically by the implementation, and can therefore involve shared regions. This approach is inherently higher-order, which has the disadvantage of leading to complex specifications. TaDA takes a first-order approach, leading to simpler specifications.

There have been extensive work understanding and generalising linearisability, especially in light of work on separation logic. Filipovic *et al.* demonstrated that linearisability can be viewed as a particular proof technique for context refinement [8]. Vafeiadis combines the ownership given by his RGSep reasoning with linearisability [20]. Gotsman and Yang generalised linearisability to include ownership transfer of memory between a client and a data structure [9]. We take this approach further, allowing ownership transfer of memory between a client and an abstraction, which a data structure is an instance. Turon *et al.* [18] have introduced CaReSL which combines refinement and Hoare-style reasoning about higher-order concurrent programs and generalises linearisability in their logic with granularity abstraction. Our notion of atomicity is more general than theirs as it allows the client to break the abstraction of atomicity.

2 Motivating Examples

We introduce TaDA by showing how two simple concurrent interfaces can be specified, implemented, and used: lock and multiple compare-and-swap.

2.1 Lock

We define a lock module with the operations `lock(x)` and `unlock(x)` and a constructor `makeLock()`.

2.1.1 Atomic Lock Specification.

The lock operations are specified in terms of abstract predicates [14] that represent the state of a lock: $\mathsf{L}(x)$ and $\mathsf{U}(x)$ assert the existence of a lock addressed by x that is in the locked and unlocked state, respectively. These predicates confer ownership of the lock: it is not possible to have more than one $\mathsf{L}(x)$ or

$\mathsf{U}(x)$ for the same value of x. This contrasts with the style of specification given with CAP [4], but we shall see how the CAP specification can be derived using the atomic specification given here.

The makeLock() operation has the simplest specification:

$$\vdash \{\mathsf{emp}\}\ \mathtt{x} := \mathtt{makeLock}()\ \{\mathsf{U}(\mathtt{x})\}$$

It simply allocates a new lock, which is initially unlocked, and returns its address. The specification says nothing about the granularity of the operation. In fact, the granularity is hardly relevant, since no concurrent environment can meaningfully observe the effects of makeLock until its return value is known — that is, once the operation has completed.

The specification for the unlock(x) operation uses an *atomic* triple:

$$\vdash \langle\mathsf{L}(\mathtt{x})\rangle\ \mathtt{unlock}(\mathtt{x})\ \langle\mathsf{U}(\mathtt{x})\rangle$$

Intuitively, this specification means that unlock(x) will *atomically* take the lock x from the locked to unlocked state. This atomicity means that the resources in the specification may be *shared* — that is, concurrently accessible by multiple threads. Sharing in this way is not possible with ordinary triples, since they make no guarantee about preserving invariants on the shared resource in intermediate steps. The atomic triple, by contrast, makes a strong guarantee: as long as the concurrent environment guarantees that the (possibly) shared resource $\mathsf{L}(\mathtt{x})$ is available, the unlock(x) operation will preserve $\mathsf{L}(\mathtt{x})$ until it transforms it into $\mathsf{U}(\mathtt{x})$; after the transformation, the operation no longer requires $\mathsf{U}(\mathtt{x})$, and is consequently oblivious to subsequent transformations by the environment (such as another thread acquiring the lock).

It is significant that the notion of atomicity is tied to the abstraction in the specification. The predicate $\mathsf{L}(\mathtt{x})$ could abstract multiple underlying states in the implementation. If we were to observe the underlying state, the operation might no longer appear to be atomic.

Specifying lock(x) is more subtle. It can be called whether the lock is in the locked or unlocked state, and always results in setting it to the locked state (if it ever terminates). A first attempt at a specification might therefore be:

$$\vdash \langle\mathsf{L}(\mathtt{x}) \vee \mathsf{U}(\mathtt{x})\rangle\ \mathtt{lock}(\mathtt{x})\ \langle\mathsf{L}(\mathtt{x})\rangle$$

This specification has two significant flaws. Firstly, it allows lock(x) to do nothing at all when the lock is already locked. This is contrary to what it should do, which is wait for it to become unlocked and then (atomically) lock it. Secondly, as the level of abstraction given by the precondition is $\mathsf{L}(\mathtt{x}) \vee \mathsf{U}(\mathtt{x})$, an implementation could change the state of the lock arbitrarily *without appearing to have done anything.* In particular, an implementation could transition between the two states any number of times, so long as it is in the $\mathsf{L}(\mathtt{x})$ state when it finishes.

A second attempt to overcome these issues might be:

$$\vdash \langle\mathsf{L}(\mathtt{x})\rangle\ \mathtt{lock}(\mathtt{x})\ \langle\mathsf{false}\rangle \qquad \vdash \langle\mathsf{U}(\mathtt{x})\rangle\ \mathtt{lock}(\mathtt{x})\ \langle\mathsf{L}(\mathtt{x})\rangle$$

Here we have two specifications. In the first, the lock is initially locked; the implementation may not terminate, nor change the state of the lock. In the second, the lock is initially unlocked; the implementation may only make one atomic transformation from unlocked to locked. These specifications also have a subtle flaw. They both assume that the environment will not change the state of the lock. This would prevent us from having multiple threads competing to acquire the lock, which is the essential purpose of a lock.

To give a correct specification, we need to express that the environment is not constrained to preserve the state of the lock, but that the operation is constrained to perform only its atomic update. We achieve this with a more general form of atomic specification:

$$\vdash \mathbb{V} l \in \mathbb{B}.\, \big\langle (\mathsf{L}(\mathtt{x}) \land \neg l) \lor (\mathsf{U}(\mathtt{x}) \land l) \big\rangle \ \mathtt{lock(x)} \ \big\langle \mathsf{L}(\mathtt{x}) \land l \big\rangle$$

This specification introduces a logical variable l (ranging over the booleans) that is scoped across both the pre- and postcondition. This variable is simply used to record the state of the lock when the atomic operation takes effect. In particular, it cannot take effect unless the lock is already unlocked.

The special role of l (indicated by the pseudo-quantifier $\mathbb{V}$) is in distinguishing the constraints on the environment and on the thread before the atomic operation takes effect. Specifically, the environment is at liberty to change the value of l for which the precondition holds (that is, lock and unlock the lock), but the thread executing the operation must preserve the value of l (that is, it cannot lock or unlock the lock except by performing the atomic operation).

2.1.2 CAP Lock Specification.

We show how a CAP-style lock specification [4] can be derived from the above atomic specification. This illustrates a typical use of a TaDA specification: first prove a strong abstract-atomic specification, then weaken to whatever is required by the client.

The CAP specification uses two abstract predicates: $\mathsf{isLock}(x)$, which is a non-exclusive resource that allows a thread to compete for the lock; and $\mathsf{Locked}(x)$, which is an exclusive resource that represents that the thread has acquired the lock and allows it to release the lock. The lock is specified as follows (we omit `makelock` for brevity):

$$\vdash \big\{\mathsf{Locked}(\mathtt{x})\big\} \ \mathtt{unlock(x)} \ \big\{\mathsf{emp}\big\}$$

$$\vdash \big\{\mathsf{isLock}(\mathtt{x})\big\} \ \mathtt{lock(x)} \ \big\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\big\}$$

$$\mathsf{isLock}(x) \iff \mathsf{isLock}(x) * \mathsf{isLock}(x)$$

$$\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{false}$$

To implement this specification, we must provide an interpretation for the abstract predicates. For this, we need to introduce a shared region. As in CAP, a shared region encapsulates some resource that is available to multiple threads. In our example, this resource will be the predicates $\mathsf{L}(x)$ and $\mathsf{U}(x)$, plus some additional guard (capability) resource. A shared region is associated with a protocol, which determines how its contents change over time. Following iCAP, the state of a shared region is abstracted, and protocols are expressed as transition systems over these abstract states. A thread may only change the abstract state of a region when it has the *guard* resource associated with the transition to be performed. An interpretation function associates each abstract state of a region with a concrete assertion. In summary, to specify a region we must supply the guards for the region, an abstract state transition system that is labelled by these guards, and a function interpreting abstract states as assertions.

In CAP, guards consist of (parametrised) names, associated with fractional permissions. In TaDA, we are more general, effectively allowing guards to be taken from any separation algebra. This gives us more flexibility in specifying complex usage patterns for regions. For the CAP lock, however, we need only a very simple notion of guards: there is a single, indivisible guard named K (for 'key'), as well as the empty guard **0**. As a separation algebra, guard resources must have a partial composition operator that is associative and commutative. In this case, $\mathbf{0} \bullet x = x = x \bullet \mathbf{0}$ for all $x \in \{\mathbf{0}, \mathrm{K}\}$, with the only other composition $\mathrm{K} \bullet \mathrm{K}$ being undefined.

The transition system for the region will have two states: 0 and 1, corresponding to unlocked and locked states respectively. Intuitively, any thread should be allowed to lock the lock, if it is unlocked, but only the thread holding the 'key' should be able to unlock it. This is specified by the labelled transition system:

$$\begin{array}{rcl} \mathbf{0} & : & 0 \rightsquigarrow 1 \\ \mathrm{K} & : & 1 \rightsquigarrow 0 \end{array}$$

It remains to give an interpretation for the abstract states of the transition system. To do so, we must have a name for the type of region we are defining; we shall use **CAPLock**. It is possible for there to be multiple regions associated with the same region type name. To distinguish them, each region has a unique region identifier, which is typically annotated as a subscript. A region specification may take some parameters that are used in the interpretation. With **CAPLock**, for instance, the address of the lock is such a parameter. We thus specify the type name, region identifier, parameters and state of a region in the form $\mathbf{CAPLock}_r(x, s)$.

The region interpretation for **CAPLock** is given by:

$$\begin{aligned} I(\mathbf{CAPLock}_r(x, 0)) &\triangleq \mathsf{U}(x) * [\mathrm{K}]_r \\ I(\mathbf{CAPLock}_r(x, 1)) &\triangleq \mathsf{L}(x) \end{aligned}$$

With this interpretation, the guard K is in the region when it is in the unlocked state. This means that, when a thread acquires the lock, it takes ownership of the the guard by removing it from the region and, hence, uses it to subsequently release the lock.

We can now give an interpretation to the predicates $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$:

$$\begin{array}{rcl} \mathsf{isLock}(x) & \triangleq & \exists r.\, \exists s \in \{0, 1\}\,.\, \mathbf{CAPLock}_r(x, s) \\ \mathsf{Locked}(x) & \triangleq & \exists r.\, \mathbf{CAPLock}_r(x, 1) * [\mathrm{K}]_r \end{array}$$

It remains to prove the specifications for the procedures and the axioms. The key proof rule is "use atomic". A simplified version of the rule is as follows:

$$\frac{\begin{array}{c} \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \vdash \forall\!\!\!\forall x \in X.\, \left\langle I(\mathbf{t}_a(x)) * [\mathrm{G}]_a \right\rangle \ \mathbb{C} \ \left\langle I(\mathbf{t}_a(f(x))) * q \right\rangle \end{array}}{\vdash \left\{ \exists x \in X.\, \mathbf{t}_a(x) * [\mathrm{G}]_a \right\} \ \mathbb{C} \ \left\{ \exists x \in X.\, \mathbf{t}_a(f(x)) * q \right\}}$$

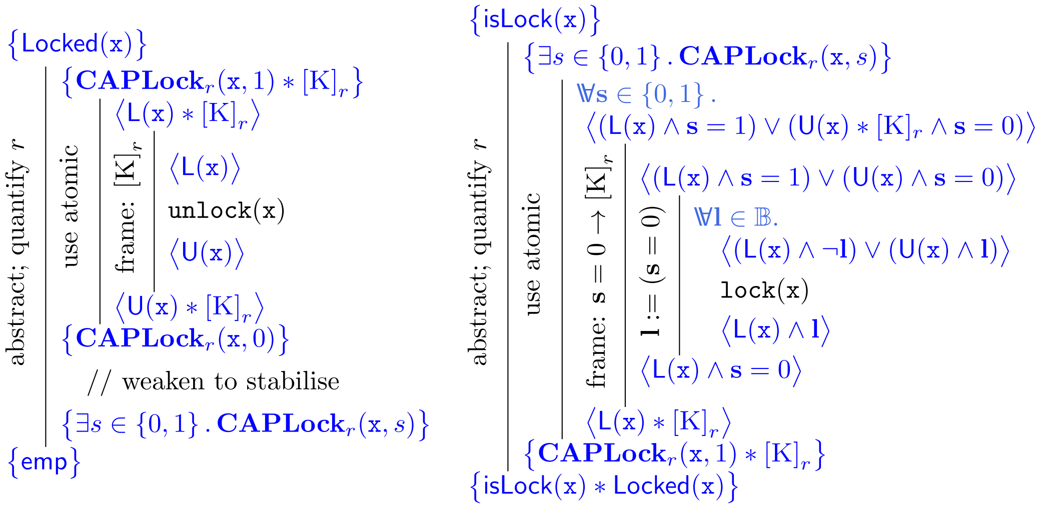
$\{\mathsf{Locked}(\mathtt{x})\}$
abstract; quantify r
$\{\mathbf{CAPLock}_r(\mathtt{x},1) * [\mathrm{K}]_r\}$
use atomic
$\langle \mathsf{L}(\mathtt{x}) * [\mathrm{K}]_r \rangle$
frame: $[\mathrm{K}]_r$
$\langle \mathsf{L}(\mathtt{x}) \rangle$
$\mathtt{unlock}(\mathtt{x})$
$\langle \mathsf{U}(\mathtt{x}) \rangle$
$\langle \mathsf{U}(\mathtt{x}) * [\mathrm{K}]_r \rangle$
$\{\mathbf{CAPLock}_r(\mathtt{x},0)\}$
// weaken to stabilise
$\{\exists s \in \{0,1\} . \mathbf{CAPLock}_r(\mathtt{x},s)\}$
$\{\mathsf{emp}\}$
$\{\mathsf{isLock}(\mathtt{x})\}$
abstract; quantify r
$\{\exists s \in \{0,1\} . \mathbf{CAPLock}_r(\mathtt{x},s)\}$
use atomic
$\mathbb{\forall} \mathbf{s} \in \{0,1\} .$
$\langle (\mathsf{L}(\mathtt{x}) \wedge \mathbf{s} = 1) \vee (\mathsf{U}(\mathtt{x}) * [\mathrm{K}]_r \wedge \mathbf{s} = 0) \rangle$
frame: $\mathbf{s} = 0 \rightarrow [\mathrm{K}]_r$
$\langle (\mathsf{L}(\mathtt{x}) \wedge \mathbf{s} = 1) \vee (\mathsf{U}(\mathtt{x}) \wedge \mathbf{s} = 0) \rangle$
$\mathbf{l} := (\mathbf{s} = 0)$
$\mathbb{\forall} \mathbf{l} \in \mathbb{B}.$
$\langle (\mathsf{L}(\mathtt{x}) \wedge \neg \mathbf{l}) \vee (\mathsf{U}(\mathtt{x}) \wedge \mathbf{l}) \rangle$
$\mathtt{lock}(\mathtt{x})$
$\langle \mathsf{L}(\mathtt{x}) \wedge \mathbf{l} \rangle$
$\langle \mathsf{L}(\mathtt{x}) \wedge \mathbf{s} = 0 \rangle$
$\langle \mathsf{L}(\mathtt{x}) * [\mathrm{K}]_r \rangle$
$\{\mathbf{CAPLock}_r(\mathtt{x},1) * [\mathrm{K}]_r\}$
$\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\}$

Figure 1: Derivation of CAP lock specifications.

This rule allows a region a, with region type $\mathbf{t}$, to be opened so that it may be updated by $\mathbb{C}$, from some state $x \in X$ to state $f(x)$. In order to do so, the precondition must include a guard G that is sufficient to perform the update to the region, in accordance with the labelled transition system — this is established by the first premiss.

The proofs of the `unlock` and `lock` operations are given in Fig. 1. In the `unlock` proof, note that the immediate postcondition of the "use atomic" is not stable, since it is possible for the environment to acquire the lock. For illustrative purposes, we weaken it minimally to a stable assertion, although it could be weakened to emp directly.

The `lock` proof uses the $\mathbb{\forall}$ quantifier in the premiss of the "use atomic" to account for the fact that, in the precondition, the lock could be in either state. Note that neither of the bad specifications for $\mathtt{lock}(\mathtt{x})$ could be used in this derivation: the first because there would be no way to express that the framed guard $[\mathrm{K}]_r$ is conditional on the state of the lock; the second because we could not combine both cases in a single derivation. The proof also uses a substitution rule to replace the boolean variable $\mathbf{l}$ recording the state of the lock when the atomic operation happens with the variable $\mathbf{s}$ representing the state of **CAPLock** region. To derive the final postcondition, we use the fact that region assertions, since they refer to shared resource, are freely duplicable: *i.e.* $\mathbf{CAPLock}_r(\mathtt{x},1) \equiv \mathbf{CAPLock}_r(\mathtt{x},1) * \mathbf{CAPLock}_r(\mathtt{x},1)$. The axiom $\mathsf{isLock}(x) \iff \mathsf{isLock}(x) * \mathsf{isLock}(x)$ similarly follows from the duplicability of region assertions. Finally, the axiom $\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{false}$ follows from the fact that $\mathrm{K} \bullet \mathrm{K}$ is undefined.

2.1.3 Spin Lock Implementation.

We consider a spin lock implementation of the atomic lock specification. The code is given in Fig. 2. We make use of three atomic operations that manipulate

```
function makeLock() {
  v := alloc(1);
  [v] := 0;
  return v;
}
```

```
function unlock(x) {
  [x] := 0;
  return ;
}
```

```
function lock(x) {
  do {
    b := CAS(x, 0, 1);
  } while (b = 0);
  return ;
}
```

Figure 2: Lock operations.

the heap. The operation $x := [y]$ reads the value of the heap position y to the variable x. The operation $[x] := y$ stores the value y in the heap position x. Finally, $\texttt{CAS}(x, v, w)$ checks if the value at heap position x is v: if so, it replaces it with w and returns 1; if not, it returns 0.

To verify this implementation against the atomic specification, we must give a concrete interpretation of the abstract predicates. To do this, we introduce a new region type, **Lock**. There is only one non-empty guard for a **Lock** region, named G (for 'guard'), much as for **CAPLock**. There are also two states for a **Lock** region: 0 and 1, representing unlocked and locked respectively. A key difference from **CAPLock** is that transitions in both directions are guarded by G. The labelled transition system is as follows:

$$\begin{aligned} \mathrm{G} &: 0 \rightsquigarrow 1 \\ \mathrm{G} &: 1 \rightsquigarrow 0 \end{aligned}$$

We also give an interpretation to each abstract state as follows:

$$\begin{aligned} I(\mathbf{Lock}_a(x, 1)) &\triangleq x \mapsto 1 \\ I(\mathbf{Lock}_a(x, 0)) &\triangleq x \mapsto 0 \end{aligned}$$

We now define the interpretation of the predicates using the regions and the guards as follows:

$$\begin{aligned} \mathsf{L}(x) &\triangleq \exists a.\, \mathbf{Lock}_a(x, 1) * [\mathrm{G}]_a \\ \mathsf{U}(x) &\triangleq \exists a.\, \mathbf{Lock}_a(x, 0) * [\mathrm{G}]_a \end{aligned}$$

The abstract predicate $\mathsf{L}(x)$ asserts there is a region with identifier a and the region is in state 1. It also states that there is a guard $[\mathrm{G}]_a$ which will be used to update the region. $\mathsf{U}(x)$ analogously states that the region is in state 0. Note that the **Lock** region here is at level 0, since it is not necessary to open any further regions while the **Lock** region is opened.

To prove the implementations against our atomic specifications, we use TaDA's "make atomic" rule. A slightly simplified version of the rule is as follows:

$$\frac{\begin{array}{c} \{(x, y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ a : x \in X \rightsquigarrow Q(x) \vdash \left\{ \begin{array}{c} \exists x \in X.\, \mathbf{t}_a(x) \\ * \, a \Mapsto \blacklozenge \end{array} \right\} \mathbb{C} \left\{ \begin{array}{c} \exists x \in X, y \in Q(x). \\ a \Mapsto (x, y) \end{array} \right\} \end{array}}{\vdash \forall\!\!\!\forall x \in X.\, \left\langle \mathbf{t}_a(x) * [\mathrm{G}]_a \right\rangle \mathbb{C} \left\langle \mathbf{t}_a(Q(x)) * [\mathrm{G}]_a \right\rangle}$$

This rule establishes that $\mathbb{C}$ atomically updates region a, from some state $x \in X$ to some state $y \in Q(x)$. To do so, it requires the guard G for the region, which must permit the update according to the transition system — this is established by the first premiss.

The second premiss introduces two new notations. The first, $a : x \in X \rightsquigarrow Q(x)$, is called the atomicity context. The atomicity context records the abstract atomic action that is to be performed. The second, $a \Mapsto -$, is the atomic tracking resource. The atomic tracking resource indicates whether the atomic update has occurred ($a \Mapsto \blacklozenge$ indicates it has not) and if so, the state of the shared region immediately before it did and after. The resource $a \Mapsto \blacklozenge$ also plays two special roles that are normally filled by guards. Firstly, it limits the interference on region a: the environment may only update the state so long as it remains in the set X (as specified by the atomicity context). Secondly, it confers permission on for the thread to update the region from state $x \in X$ to any state $y \in Q(x)$; in doing so, the thread also updates $a \Mapsto \blacklozenge$ to $a \Mapsto (x, y)$. This permission is expressed by the "update region" rule, and ensures that the atomic update only happens once.

In essence, the second premiss is capturing the notion of atomicity (with respect to the abstraction in the conclusion) and expressing it as a proof obligation. Specifically, the region must be in state x for some $x \in X$, which may be changed by the environment until, at some point, the thread updates it to some $y \in Q(x)$. The atomic tracking resource bears witness to this.

Now we can prove the `lock(x)` implementation as follows:

```
$\mathbb{V} l \in \mathbb{B}.$
$\langle (\mathsf{L}(\mathtt{x}) \land \neg l) \lor (\mathsf{U}(\mathtt{x}) \land l) \rangle$
  abstract; quantify $a$
  $\langle (\mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \land \neg l) \lor (\mathbf{Lock}_a(\mathtt{x}, 0) * [\mathrm{G}]_a \land l) \rangle$
    $y := \text{if } l \text{ then } 0 \text{ else } 1$
    $\mathbb{V} y \in \{0, 1\}.$
    $\langle \mathbf{Lock}_a(\mathtt{x}, y) * [\mathrm{G}]_a \rangle$
      make atomic
      $a : y \in \{0, 1\} \rightsquigarrow 1 \land y = 0 \vdash$
        $\{\exists y \in \{0, 1\} . \mathbf{Lock}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
        do {
          $\{\exists y \in \{0, 1\} . \mathbf{Lock}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
            update region
            $\mathbb{V} n \in \{0, 1\}.$
            $\langle \mathtt{x} \mapsto n \rangle$
            b := CAS(x, 0, 1);
            $\left\langle \begin{array}{l} (\mathtt{x} \mapsto 1 \land n = 0 * \mathtt{b} \mapsto 1) \lor \\ (\mathtt{x} \mapsto n \land n \neq 0 * \mathtt{b} \mapsto 0) \end{array} \right\rangle$
          $\left\{ \begin{array}{l} \exists y \in \{0, 1\} . \mathbf{Lock}_a(\mathtt{x}, y) * \\ (a \Mapsto (0, 1) \land \mathtt{b} = 1 \lor a \Mapsto \blacklozenge \land \mathtt{b} = 0) \end{array} \right\}$
        } while (b = 0);
        $\{a \Mapsto (0, 1) \land \mathtt{b} = 1\}$
    $\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \land y = 0 \rangle$
  $\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \land l \rangle$
$\langle \mathsf{L}(\mathtt{x}) \land l \rangle$
```

The proof first massages the specification into a form where we can apply the "make atomic" rule. The atomicity context allows the region a to be in either state, but insists that it must have been in the unlocked state (the 0) when the atomic operation takes effect. ($Q(1) = \emptyset$ while $Q(0) = \{1\}$.) The "update region" rule conditionally performs the atomic action — transitioning the region from state 0 to 1, and recording this in the atomic tracking resource — if the atomic compare-and-swap operation succeeds.

The proof of `unlock(x)` implementation is as follows:

```
$\langle \mathsf{L}(\mathtt{x}) \rangle$
  abstract; quantify $a$
  $\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \rangle$
    make atomic
    $a : 1 \rightsquigarrow 0 \vdash$
      $\{ \mathbf{Lock}_a(\mathtt{x}, 1) * a \Mapsto \blacklozenge \}$
        update region
        $\langle \mathtt{x} \mapsto 1 \rangle$
        [x] := 0;
        $\langle \mathtt{x} \mapsto 0 \rangle$
      $\{ a \Mapsto (1, 0) \}$
  $\langle \mathbf{Lock}_a(\mathtt{x}, 0) * [\mathrm{G}]_a \rangle$
$\langle \mathsf{U}(\mathtt{x}) \rangle$
```

and finally the proof of `x` := `makeLock`()

```
$\{ \mathsf{emp} \}$
v := alloc(1);
$\{ \mathtt{v} \mapsto - \}$
[v] := 0;
$\{ \mathtt{v} \mapsto 0 \}$
$\{ \mathbf{Lock}_a(\mathtt{v}, 0) * [\mathrm{G}]_a \}$
$\{ \mathsf{U}(\mathtt{v}) \}$
return v;
$\{ \mathsf{U}(\mathtt{x}) \}$
```

Remark. It is possible to prove the following alternative implementation of `unlock(x)` with the same atomic specification:

$$\vdash \langle \mathsf{L}(\mathtt{x}) \rangle \ [\mathtt{x}] := 1; [\mathtt{x}] := 0 \ \langle \mathsf{U}(\mathtt{x}) \rangle$$

The first write to `x` has no effect, since the specification asserts that the lock must be locked initially. This code would clearly not be atomic in a different context; it would not satisfy the specification $\vdash \langle \mathsf{L}(\mathtt{x}) \vee \mathsf{U}(\mathtt{x}) \rangle$ `unlock(x)` $\langle \mathsf{U}(\mathtt{x}) \rangle$, for example. This highlights the fragility of the fiction of atomicity.

Remark. While traditional linearisability approaches might be able to prove a specification similar to ours, TaDA's basis in separation logic gives our specification a subtlety: separation can guarantee that the resources used by the lock module are disjoint from those used by the client. This is essential for using the lock in practice, since if the resources were not disjoint, a client could potentially break the atomicity abstraction by performing basic heap operations. Disjointness is critical to the modularity of the specification.

2.2 Multiple Compare-and-swap (MCAS)

Abstract Specification. We look at an interface over the heap which provides atomic double-compare-and-swap (`dcas`) and triple-compare-and-swap (`3cas`) operations, in addition to the basic read, write and compare-and-swap operations. It makes use of two abstract predicates: $\mathsf{MCL}(l)$ to represent an instance of the MCAS library at address l; and $\mathsf{MCP}(l, x, v)$ to represent the "MCAS heap cell" at address x with value v, protected by instance l. There is an abstract disjointness, as we can view each heap cell as disjoint from each other at the abstract level, even if that is not the case with the implementation itself. The specification for creating the interface, transferring memory cells to and from it as well as manipulating it is given in Fig. 3.

$$\vdash \{\mathsf{emp}\}\ \mathtt{l} := \mathtt{makeMCL}()\ \{\mathsf{MCL}(\mathtt{l})\}$$

$$\vdash \forall v.\ \langle \mathtt{x} \mapsto v * \mathsf{MCL}(\mathtt{l}) \rangle\ \mathtt{makeMCP}(\mathtt{l}, \mathtt{x})\ \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCL}(\mathtt{l}) \rangle$$

$$\vdash \{\mathsf{MCP}(\mathtt{l}, \mathtt{x}, v)\}\ \mathtt{unmakeMCP}(\mathtt{l}, \mathtt{x})\ \{\mathtt{x} \mapsto v\}$$

$$\vdash \forall v.\ \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \rangle\ \mathtt{y} := \mathtt{read}(\mathtt{l}, \mathtt{x})\ \langle \mathtt{y} = v \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \rangle$$

$$\vdash \forall v.\ \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \rangle\ \mathtt{write}(\mathtt{l}, \mathtt{x}, \mathtt{w})\ \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{w}) \rangle$$

$$\vdash \forall v.\ \left\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \right\rangle\ \mathtt{b} := \mathtt{cas}(\mathtt{l}, \mathtt{x}, \mathtt{v1}, \mathtt{v2})\ \left\langle \begin{array}{r} \underline{\textbf{if}}\ v = \mathtt{v1}\ \underline{\textbf{then}}\ \mathtt{b} = 1 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{v2}) \\ \underline{\textbf{else}}\ \mathtt{b} = 0 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \end{array} \right\rangle$$

$$\begin{array}{ll} \vdash \forall v, w. & \begin{array}{c} \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) \rangle \\ \mathtt{b} := \mathtt{dcas}(\mathtt{l}, \mathtt{x}, \mathtt{y}, \mathtt{v1}, \mathtt{w1}, \mathtt{v2}, \mathtt{w2}) \\ \left\langle \begin{array}{c} \underline{\textbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1} \\ \underline{\textbf{then}}\ \mathtt{b} = 1 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{v2}) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, \mathtt{w2}) \\ \underline{\textbf{else}}\ \mathtt{b} = 0 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) \end{array} \right\rangle \end{array} \end{array}$$

$$\begin{array}{ll} \lambda \vdash \forall v, w, u. & \begin{array}{c} \langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) * \mathsf{MCP}(\mathtt{l}, \mathtt{z}, u) \rangle \\ \mathtt{b} := \mathtt{3cas}(\mathtt{l}, \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{v1}, \mathtt{w1}, \mathtt{u1}, \mathtt{v2}, \mathtt{w2}, \mathtt{u2}) \\ \left\langle \begin{array}{c} \underline{\textbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1} \land u = \mathtt{u1} \\ \underline{\textbf{then}}\ \mathtt{b} = 1 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{v2}) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, \mathtt{w2}) * \mathsf{MCP}(\mathtt{l}, \mathtt{z}, \mathtt{u2}) \\ \underline{\textbf{else}}\ \mathtt{b} = 0 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) * \mathsf{MCP}(\mathtt{l}, \mathtt{z}, u) \end{array} \right\rangle \end{array} \end{array}$$

$$\mathsf{MCP}(l, x, v) * \mathsf{MCP}(l, x, w) \implies \mathsf{false}$$

Figure 3: The abstract specification for the MCAS module.

Implementation. We give a straightforward coarse-grained implementation of the MCAS specification. The operation `makeMCL` creates a lock which is used to protect updates to pointers under the control of the library. The `makeMCP` and `unmakeMCP` operations do nothing: the transfer of resources to and from the library is purely logical. The other operations simply acquire the lock, perform the appropriate reads and writes, and then release the lock.

We interpret the abstract predicates using a single shared region, with type name **MCAS**. The abstract states of the region are *partial heaps*, which represent the part of the heap that is protected by the module. For instance, the abstract state $x \mapsto v \bullet y \mapsto w$ indicates that heap cells x and y are under the protection of the module, with logical values v and w respectively. Note that the physical values at x and y need not be the same as their logical values, specifically when the lock has been acquired and they are being modified.

For the **MCAS** region, there are five kinds of guard. The $\textsc{Own}(x)$ guard confers ownership of the heap cell at address x under the control of the region. This guard is used by all operations of the library that access the heap cell x. The following equivalence ensures that there can only be one instance of $\textsc{Own}(x)$:

$$[\textsc{Own}(x)]_m * [\textsc{Own}(x)]_m \implies \mathsf{false}$$

We amalgamate the $\textsc{Own}$ guards for heap cells that are not currently under the protection of the module into $\textsc{Rest}(X)$, where X is the set of all cells that *are* protected. We have the following equivalence:

$$[\textsc{Rest}(X)]_m \iff [\textsc{Rest}(X \uplus \{x\})]_m * [\textsc{Own}(x)]_m$$

Initially the set X will be empty. When we add an element $x \mapsto v$ to the region, we get a guard $\textsc{Own}(x)$ that allows us to manipulate the abstract state for that particular x. There can be only one $\textsc{Rest}$ guard:

$$[\textsc{Rest}(X)]_m * [\textsc{Rest}(Y)]_m \implies \mathsf{false}$$

The remaining guards are effectively used as auxiliary state. When a thread acquires the lock, it removes some heap cells from the shared region in order to access them. The $\textsc{Locked}(h)$ guard will be used to record that the heap cells in h have been removed in this way. The thread that acquired the lock will have a corresponding $\textsc{Key}(h)$ guard. When it releases the lock, the two guards will be reunited inside the region to form the $\textsc{Unlocked}$ guard. This is expressed by the following equivalence:

$$[\textsc{Unlocked}]_m \iff [\textsc{Locked}(h)]_m * [\textsc{Key}(h)]_m$$

The transition system for the region is parametric in each heap cell. It allows anyone to add the resource $x \mapsto v$ to the region. (There is no need to guard this action, as the resource is unique and as such only one thread can do it for a particular value of x.) It allows the value of x to be updated using the guard $\textsc{Own}(x)$. Finally, given the guard $\textsc{Own}(x)$, the value x can be removed from the region. We formally define the transition system as follows:

$$\begin{aligned}
\mathbf{0} &: \forall h, x, v.\, h \rightsquigarrow x \mapsto v \bullet h \\
\textsc{Own}(x) &: \forall h, v, w.\, x \mapsto v \bullet h \rightsquigarrow x \mapsto w \bullet h \\
\textsc{Own}(x) &: \forall h, x, v.\, x \mapsto v \bullet h \rightsquigarrow h
\end{aligned}$$

We define the interpretation of abstract states for the **MCAS** region:

$$\begin{aligned}
I(\mathbf{MCAS}_m(l, h)) \triangleq {} & [\textsc{Rest}(\mathrm{dom}(h))]_m * (\mathsf{U}(l) * h * [\textsc{Unlocked}]_m \lor \\
& \qquad \exists h_1, h_2.\, \mathsf{L}(l) * h_1 * [\textsc{Locked}(h_2)]_m \land h = h_1 \bullet h_2)
\end{aligned}$$

Internally, the region may be in one of two states, indicated by the disjunction. In one case, the lock l is unlocked, the heap cells corresponding to the abstract state of the region are actually in the region, and so is the $\textsc{Unlocked}$ guard. In the other, the lock l is locked, some portion h_1 of the abstract heap is in the

In the following, let $h_v = \mathtt{x} \mapsto v$.

$\mathbb{V}v.$

$\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \rangle$

abstract; quantify m

$\langle \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_v * h) * [\textsc{Own}(\mathtt{x})]_m \rangle$

make atomic

$m : h_v \bullet h \rightsquigarrow h_v \bullet h \vdash$

$\{\exists h, v.\, \mathbf{MCAS}_m(\mathtt{l}, h_v \bullet h) * m \Mapsto \blacklozenge\}$

open region

$\mathbb{V}h.$

$$\left\langle \begin{pmatrix} \mathsf{U}(\mathtt{l}) * h * [\textsc{Unlocked}]_m \ \vee \\ \mathsf{L}(\mathtt{l}) * \exists h_1, h_2.\, h = (h_1 \bullet h_2) \wedge h_1 \\ * [\textsc{Locked}(h_2)]_m \end{pmatrix} * [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge \right\rangle$$

`lock(l);` // remove from the shared region the heap cell

$$\left\langle \begin{array}{l} \exists h_1.\, \mathsf{L}(l) * h_1 * [\textsc{Locked}(h_v)]_m \wedge h = (h_1 \bullet h_v) * \\ [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \end{array} \right\rangle$$

$\{\exists h, v.\, \mathbf{MCAS}_m(l, h_v \bullet h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m * h_v\}$

`v := [x];`

$\{\exists h, v.\, \mathbf{MCAS}_m(l, h_v * h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m \wedge \mathtt{v} = v\}$

update region

$\mathbb{V}h.$

$$\left\langle \begin{array}{l} \exists h_1.\, h = (h_1 \bullet h_v) \wedge \mathsf{L}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * \\ [\textsc{Locked}(h_v)]_m * [\textsc{Key}(h_v)]_m * h_1 * h_v \wedge \mathtt{v} = v \end{array} \right\rangle$$

`unlock(l);` // put the heap cell in the shared region

$\langle \mathsf{U}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * [\textsc{Unlocked}]_m * h \rangle$

$\{\exists h, v.\, m \Mapsto (h_v \bullet h, h_v \bullet h) \wedge \mathtt{v} = v\}$

`return v;`

$\{\exists h, v.\, m \Mapsto (h_v \bullet h, h_v \bullet h) \wedge \mathsf{ret} = v\}$

$\langle \exists h, v.\, \mathbf{MCAS}_m(l, h_v \bullet h) * [\textsc{Own}(\mathtt{x})]_m \wedge \mathsf{ret} = v \rangle$

$\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{w}) \rangle$

Figure 4: Proof of the `read` implementation.

region, while the remainder, h_2, has been removed, together with the $\textsc{Key}(h_2)$ guard, leaving behind the $\textsc{Locked}(h_2)$ guard. In both cases, the $\textsc{Rest}(\mathrm{dom}(h))$ guard belongs to the state, encapsulating the $\textsc{Own}$ guards for the cells that are not in the heap.

We now give an interpretation to the predicates as follows:

$$\mathsf{MCL}(l) \triangleq \exists m, h.\, \mathbf{MCAS}_m(l, h)$$

$$\mathsf{MCP}(l, x, v) \triangleq \exists m, h.\, \mathbf{MCAS}_m(l, x \mapsto v \bullet h) * [\textsc{Own}(x)]_m$$

The predicate $\mathsf{MCL}(l)$ states the existence of the shared region, but makes no assumptions about its state. The predicate $\mathsf{MCP}(l, x, v)$ states that there is x with value v and possible other heap cells in it.

We can not prove that the specification is satisfied by the implementation, shown in Fig. 7 for `dcas` command; Fig. 4 for `read` command; Fig. 5 for `write` command; and Fig. 6 for `cas` command..

The axiom $\mathsf{MCP}(l, x, v) * \mathsf{MCP}(l, x, w) \implies \mathsf{false}$ follows from the fact that $\textsc{Own}(x) \bullet \textsc{Own}(x)$ is undefined.

In the following, let $h_v = \mathtt{x} \mapsto v$ and $h_{\mathtt{w}} = \mathtt{x} \mapsto \mathtt{w}$.

```
$\mathbb{A} v.$
$\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \rangle$
  abstract; quantify $m$
  $\langle \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_v * h) * [\textsc{Own}(\mathtt{x})]_m \rangle$
    make atomic
    $m : h_v \bullet h \rightsquigarrow h_{\mathtt{w}} \bullet h \vdash$
      $\{ \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_v \bullet h) * m \Mapsto \blacklozenge \}$
        open region
        $\mathbb{A} h.$
        $\left\langle \left( \begin{array}{c} \mathsf{U}(\mathtt{l}) * h * [\textsc{Unlocked}]_m \;\lor \\ \mathsf{L}(\mathtt{l}) * \exists h_1, h_2.\, h = (h_1 \bullet h_2) \land h_1 \\ * [\textsc{Locked}(h_2)]_m \end{array} \right) * [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge \right\rangle$
        lock(l); // remove from the shared region the heap cell
        $\left\langle \begin{array}{l} \exists h_1.\, \mathsf{L}(l) * h_1 * [\textsc{Locked}(h_v)]_m \land h = (h_1 \bullet h_v) * \\ [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \end{array} \right\rangle$
      $\{ \exists h.\, \mathbf{MCAS}_m(l, h_v \bullet h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \}$
      [x] := w;
      $\{ \exists h.\, \mathbf{MCAS}_m(l, h_v * h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_v)]_m * h_{\mathtt{w}} \}$
        update region
        $\mathbb{A} h.$
        $\left\langle \begin{array}{l} \exists h_1.\, h = (h_1 \bullet h_v) \land \mathsf{L}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * \\ [\textsc{Locked}(h_v)]_m * [\textsc{Key}(h_v)]_m * h_1 * h_{\mathtt{w}} \end{array} \right\rangle$
        unlock(l); // put the heap cell in the shared region and update
                   // its abstract state
        $\langle \mathsf{U}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * [\textsc{Unlocked}]_m * h[\mathtt{x} \mapsto \mathtt{w}] \rangle$
      $\{ \exists h.\, m \Mapsto (h_v \bullet h, h_{\mathtt{w}} \bullet h) \}$
      return ;
  $\langle \exists h.\, \mathbf{MCAS}_m(l, h_{\mathtt{w}} \bullet h) * [\textsc{Own}(\mathtt{x})]_m \rangle$
$\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{w}) \rangle$
```

Figure 5: Proof of the `write` implementation.

In the following, let $h_v = \mathtt{x} \mapsto v$ and $h_{\mathtt{v2}} = \mathtt{x} \mapsto \mathtt{v2}$.

```
$\mathbb{V}v.$
$\left\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \right\rangle$
abstract; quantify $m$
  $\left\langle \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_v * h) * [\textsc{Own}(\mathtt{x})]_m \right\rangle$
  make atomic
    $m : h_v \bullet h \rightsquigarrow \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ h_{\mathtt{v2}} \bullet h\ \underline{\mathbf{else}}\ h_v \bullet h \vdash$
      $\left\{ \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_v \bullet h) * m \Rrightarrow \blacklozenge \right\}$
      open region
        $\mathbb{V}h.$
        $\left\langle \begin{pmatrix} \mathsf{U}(\mathtt{l}) * h * [\textsc{Unlocked}]_m \ \vee \\ \mathsf{L}(\mathtt{l}) * \exists h_1, h_2.\, h = (h_1 \bullet h_2) \wedge h_1 \\ * [\textsc{Locked}(h_2)]_m \end{pmatrix} * [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Rrightarrow \blacklozenge \right\rangle$
        lock(l); // remove from the shared region the heap cell
        $\left\langle \begin{matrix} \exists h_1.\, \mathsf{L}(l) * h_1 * [\textsc{Locked}(h_v)]_m \wedge h = (h_1 \bullet h_v) * \\ [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Rrightarrow \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \end{matrix} \right\rangle$
      $\left\{ \exists h.\, \mathbf{MCAS}_m(l, h_v \bullet h) * m \Rrightarrow \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \right\}$
      v := [x]; // the environment cannot access the cell
      $\left\{ \exists h.\, \mathbf{MCAS}_m(l, h_v \bullet h) * m \Rrightarrow \blacklozenge * [\textsc{Key}(h_v)]_m * h_v \wedge \mathtt{v} = v \right\}$
      if (v = v1) { // perform conditional update on the heap cell
        [x] := v2;   r := 1;
      } else {   r := 0;   }
      $\left\{ \begin{matrix} \exists h.\, \mathbf{MCAS}_m(l, h_v * h) * m \Rrightarrow \blacklozenge * [\textsc{Key}(h_v)]_m \wedge \mathtt{v} = v * \\ \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ \mathtt{r} = 1 \wedge h_{\mathtt{v2}}\ \underline{\mathbf{else}}\ \mathtt{r} = 0 \wedge h_v \end{matrix} \right\}$
      update region
        $\mathbb{V}h.$
        $\left\langle \begin{matrix} \exists h_1.\, h = (h_1 \bullet h_v) \wedge \mathsf{L}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * \\ [\textsc{Locked}(h_v)]_m * [\textsc{Key}(h_v)]_m * h_1 * \\ \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ \mathtt{r} = 1 \wedge h_{\mathtt{v2}}\ \underline{\mathbf{else}}\ \mathtt{r} = 0 \wedge h_v \end{matrix} \right\rangle$
        unlock(l); // put the heap cell in the shared region and update
                   // its abstract state if the heap cell were modified
        $\left\langle \begin{matrix} \mathsf{U}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * [\textsc{Unlocked}]_m * \\ \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ h[\mathtt{x} \mapsto \mathtt{v2}]\ \underline{\mathbf{else}}\ h \end{matrix} \right\rangle$
      $\left\{ \begin{matrix} \exists h.\, \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ m \Rrightarrow (h_v \bullet h, h_{\mathtt{v2}} \bullet h) * \mathtt{r} = 1 \\ \underline{\mathbf{else}}\ m \Rrightarrow (h_v \bullet h, h_v \bullet h) * \mathtt{r} = 0 \end{matrix} \right\}$
      return r;
  $\left\langle \begin{matrix} (\underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ \mathsf{ret} = 1 \wedge \exists h.\, \mathbf{MCAS}_m(l, h_{\mathtt{v2}} \bullet h) \\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \wedge \exists h.\, \mathbf{MCAS}_m(l, h_v \bullet h)) * [\textsc{Own}(\mathtt{x})]_m \end{matrix} \right\rangle$
$\left\langle \begin{matrix} \underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ \mathsf{ret} = 1 \wedge \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{v2}) \\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \wedge \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) \end{matrix} \right\rangle$
```

Figure 6: Proof of the cas implementation.

In the following, let $h_{v,w} = \mathtt{x} \mapsto v \bullet \mathtt{y} \mapsto w$ and $h_{\mathtt{v2},\mathtt{w2}} = \mathtt{x} \mapsto \mathtt{v2} \bullet \mathtt{y} \mapsto \mathtt{w2}$.

```
$\forall v, w.$
$\langle \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) \rangle$
  abstract; quantify $m$
  $\langle \exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_{v,w} * h) * [\textsc{Own}(\mathtt{x})]_m * [\textsc{Own}(\mathtt{y})]_m \rangle$
    make atomic
    $m : h_{v,w} \bullet h \rightsquigarrow \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ h_{\mathtt{v2},\mathtt{w2}} \bullet h\ \underline{\mathbf{else}}\ h_{v,w} \bullet h \vdash$
      $\{\exists h.\, \mathbf{MCAS}_m(\mathtt{l}, h_{v,w} \bullet h) * m \Mapsto \blacklozenge\}$
        open region
        $\forall h.$
        $\left\langle \left( \begin{array}{c} \mathsf{U}(\mathtt{l}) * h * [\textsc{Unlocked}]_m \ \lor \\ \mathsf{L}(\mathtt{l}) * \exists h_1, h_2.\, h = (h_1 \bullet h_2) \land h_1 \\ * [\textsc{Locked}(h_2)]_m \end{array} \right) * [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge \right\rangle$
        lock(l); // remove from the shared region the two heap cells
        $\left\langle \begin{array}{l} \exists h_1.\, \mathsf{L}(l) * h_1 * [\textsc{Locked}(h_{v,w})]_m \land h = (h_1 \bullet h_{v,w}) * \\ [\textsc{Rest}(\mathrm{dom}(h))]_m * m \Mapsto \blacklozenge * [\textsc{Key}(h_{v,w})]_m * h_{v,w} \end{array} \right\rangle$
      $\{\exists h.\, \mathbf{MCAS}_m(l, h_{v,w} \bullet h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_{v,w})]_m * h_{v,w}\}$
      v := [x];   w := [y]; // the environment cannot access either cell
      $\{\exists h.\, \mathbf{MCAS}_m(l, h_{v,w} \bullet h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_{v,w})]_m * h_{v,w} \land \mathtt{v} = v \land \mathtt{w} = w\}$
      if (v = v1 and w = w1) { // perform conditional update on the heap cells
        [x] := v2;   [y] := w2;   r := 1;
      } else {   r := 0;   }
      $\left\{ \begin{array}{l} \exists h.\, \mathbf{MCAS}_m(l, h_{v,w} * h) * m \Mapsto \blacklozenge * [\textsc{Key}(h_{v,w})]_m \land \mathtt{v} = v \land \mathtt{w} = w * \\ \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ \mathtt{r} = 1 \land h_{\mathtt{v2},\mathtt{w2}}\ \underline{\mathbf{else}}\ \mathtt{r} = 0 \land h_{v,w} \end{array} \right\}$
        update region
        $\forall h.$
        $\left\langle \begin{array}{l} \exists h_1.\, h = (h_1 \bullet h_{v,w}) \land \mathsf{L}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * \\ [\textsc{Key}(h_{v,w})]_m * [\textsc{Key}(h_{v,w})]_m * h_1 * \\ \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ \mathtt{r} = 1 \land h_{\mathtt{v2},\mathtt{w2}}\ \underline{\mathbf{else}}\ \mathtt{r} = 0 \land h_{v,w} \end{array} \right\rangle$
        unlock(l); // put the heap cells in the shared region and update
                   // its abstract state if the heap cells were modified
        $\left\langle \begin{array}{l} \mathsf{U}(l) * [\textsc{Rest}(\mathrm{dom}(h))]_m * [\textsc{Unlocked}]_m * \\ \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ h[\mathtt{x} \mapsto \mathtt{v2}, \mathtt{y} \mapsto \mathtt{w2}]\ \underline{\mathbf{else}}\ h \end{array} \right\rangle$
      $\left\{ \begin{array}{l} \exists h.\, \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ m \Mapsto (h_{v,w} \bullet h, h_{\mathtt{v2},\mathtt{w2}} \bullet h) * \mathtt{r} = 1 \\ \quad \underline{\mathbf{else}}\ m \Mapsto (h_{v,w} \bullet h, h_{v,w} \bullet h) * \mathtt{r} = 0 \end{array} \right\}$
      return r;
  $\left\langle \begin{array}{l} (\underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ \mathsf{ret} = 1 \land \exists h.\, \mathbf{MCAS}_m(l, h_{\mathtt{v2},\mathtt{w2}} \bullet h) \\ \quad \underline{\mathbf{else}}\ \mathsf{ret} = 0 \land \exists h.\, \mathbf{MCAS}_m(l, h_{v,w} \bullet h)) * [\textsc{Own}(\mathtt{x})]_m * [\textsc{Own}(\mathtt{y})]_m \end{array} \right\rangle$
$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ v = \mathtt{v1} \land w = \mathtt{w1}\ \underline{\mathbf{then}}\ \mathsf{ret} = 1 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, \mathtt{v2}) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, \mathtt{w2}) \\ \quad \underline{\mathbf{else}}\ \mathsf{ret} = 0 \land \mathsf{MCP}(\mathtt{l}, \mathtt{x}, v) * \mathsf{MCP}(\mathtt{l}, \mathtt{y}, w) \end{array} \right\rangle$
```

Figure 7: Proof of the dcas implementation.

2.3 Resource Transfer

Consider an addition to the MCAS library: the `readTo` operation takes an MCAS heap cell and an ordinary heap cell and copies the value of the former into the latter. Such an operation could be implemented as follows:

$$\texttt{function readTo}(\texttt{l}, \texttt{x}, \texttt{y}) \ \{ \quad \texttt{v} := \texttt{read}(\texttt{l}, \texttt{x}); \quad [\texttt{y}] := \texttt{v}; \quad \}$$

This implementation atomically reads the MCAS cell at `x`, then writes the value to the cell at `y`. The overall effect is non-atomic in the sense that a concurrent environment could update `x` and then witness `y` being updated to the old value of `x`. However, if the environment's interaction is confined to the MCAS cell, the effect *is* atomic.

TaDA allows us to specify this kind of limited atomicity by splitting the pre- and postcondition of an atomic judgement into a *private* and a *public* part. The private part will contain resources that are particular to the thread — in this example, the heap cell at `y`. When the atomic triple is used to update a region (*e.g.* with the "use atomic" rule), these private resources cannot form part of the region's invariant. The public part will contain resources that can form part of a region's invariant — in this example, the MCAS cell at `x`.

The generalised form of our atomic judgements is:

$$\vdash \mathbb{V}\mathbf{x} \in X. \left\langle p_p \,\middle|\, p(\mathbf{x}) \right\rangle \ \mathbb{C} \quad \exists\mathbf{y} \in Y. \left\langle q_p(\mathbf{x}, \mathbf{y}) \,\middle|\, q(\mathbf{x}, \mathbf{y}) \right\rangle$$

Here, p_p is the private precondition, $p(\mathbf{x})$ is the public precondition, $q_p(\mathbf{x}, \mathbf{y})$ is the private postcondition, and $q(\mathbf{x}, \mathbf{y})$ is the public postcondition. The private precondition is independent of $\mathbf{x}$, since the environment is assumed to change $\mathbf{x}$. The two parts of the postcondition are linked by $\mathbf{y}$, which is chosen arbitrarily by the implementation when the atomic operation appears to take effect.

The `readTo` operation can be specified as follows:

$$\vdash \mathbb{V}v, w. \left\langle \texttt{y} \mapsto w \,\middle|\, \mathsf{MCP}(\texttt{l}, \texttt{x}, v) \right\rangle \ \texttt{readTo}(\texttt{l}, \texttt{x}, \texttt{y}) \ \left\langle \texttt{y} \mapsto v \,\middle|\, \mathsf{MCP}(\texttt{l}, \texttt{x}, v) \right\rangle$$

One way of understanding such specifications is in terms of ownership transfer between a client and a data structure, as in [9]: ownership of the private precondition is transferred from the client; ownership of the private postcondition is transferred to the client. In this example, the same resources (albeit modified) are transferred in and out, but this need not be the case in general. For instance, an operation could allocate a fresh location in which to store the retrieved value, which is then transferred to the client.

While it should be clear that this judgement generalises our original atomic judgement, it is revealing that it also generalises the non-atomic judgement. Indeed, $\left\{p\right\} \ \mathbb{C} \ \left\{q\right\}$ is equivalent to $\langle p | \mathsf{true} \rangle \ \mathbb{C} \ \langle q | \mathsf{true} \rangle$.

3 Logic

We give an overview of the key TaDA proof rules that deal with atomicity in Fig. 8. Here, we do not formally define the syntax and semantics of our assertions, although we describe how they are modelled in §5.

We implicitly require the pre- and postcondition assertions in our judgements to be *stable*: that is, they must account for any updates other threads could have sufficient resources to perform.

Frame rule

$$\dfrac{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, q(x,y) \right\rangle}{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle r' * p_p \,\middle|\, r(x) * p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle r' * q_p(x,y) \,\middle|\, r(x) * q(x,y) \right\rangle}$$

Substitution rule

$$\dfrac{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, q(x,y) \right\rangle \quad f : X' \to X}{\lambda; \mathcal{A} \vdash \mathbb{\forall} x' \in X'.\, \left\langle p_p \,\middle|\, p(f(x')) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(f(x'),y) \,\middle|\, q(f(x'),y) \right\rangle}$$

Atomicity weakening rule

$$\dfrac{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, p' * p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, q'(x,y) * q(x,y) \right\rangle}{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p * p' \,\middle|\, p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) * q'(x,y) \,\middle|\, q(x,y) \right\rangle}$$

Open region rule

$$\dfrac{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, I(\mathbf{t}_a^\lambda(x)) * p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, I(\mathbf{t}_a^\lambda(x)) * q(x,y) \right\rangle}{\lambda + 1; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, \mathbf{t}_a^\lambda(x) * p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, \mathbf{t}_a^\lambda(x) * q(x,y) \right\rangle}$$

Use atomic rule

$$\dfrac{\begin{array}{c} a \notin \mathcal{A} \quad \forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \lambda; \mathcal{A} \vdash \mathbb{\forall} x {\in} X.\, \left\langle p_p \,\middle|\, I(\mathbf{t}_a^\lambda(x)) * p(x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \;\; \mathbb{\exists} y {\in} Y.\, \left\langle q_p(x,y) \,\middle|\, I(\mathbf{t}_a^\lambda(f(x))) * q(x,y) \right\rangle \end{array}}{\lambda + 1; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, \mathbf{t}_a^\lambda(x) * p(x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \;\; \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, \mathbf{t}_a^\lambda(f(x)) * q(x,y) \right\rangle}$$

Update region rule

$$\dfrac{\lambda; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, I(\mathbf{t}_a^\lambda(x)) * p(x) \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Y.\, \left\langle q_p(x,y) \,\middle|\, \begin{array}{c} I(\mathbf{t}_a^\lambda(Q(x))) * q_1(x,y) \\ \vee\, I(\mathbf{t}_a^\lambda(x)) * q_2(x,y) \end{array} \right\rangle}{\lambda{+}1; a : x {\in} X \rightsquigarrow Q(x), \mathcal{A} \vdash \begin{array}{c} \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \right\rangle \\ \mathbb{C} \\ \mathbb{\exists} y {\in} Y.\, \left\langle q_p(x,y) \,\middle|\, \begin{array}{c} \exists z \in Q(x).\, \mathbf{t}_a^\lambda(z) * q_1(x,y) * a \Mapsto (x,z) \\ \vee\, \mathbf{t}_a^\lambda(x) * q_2(x,y) * a \Mapsto \blacklozenge \end{array} \right\rangle \end{array}}$$

Make atomic rule

$$\dfrac{\begin{array}{c} a \notin \mathcal{A} \quad \{(x,y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \lambda'; a : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash \begin{array}{c} \left\{ p_p * \exists x \in X.\, \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \right\} \\ \mathbb{C} \\ \left\{ \exists x \in X, y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y) \right\} \end{array} \end{array}}{\lambda'; \mathcal{A} \vdash \mathbb{\forall} x \in X.\, \left\langle p_p \,\middle|\, \mathbf{t}_a^\lambda(x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \quad \mathbb{\exists} y \in Q(x).\, \left\langle q_p(x,y) \,\middle|\, \mathbf{t}_a^\lambda(y) * [\mathrm{G}]_a \right\rangle}$$

Figure 8: Selected proof rules of TaDA.

Until now, we have elided a detail of the proof system: region levels. Each judgement of TaDA includes a region level λ in the context. This level is simply a number that indicates that only regions below level λ may be opened in the derivation of the judgement. For this to be meaningful, each region is associated with a level (indicated as a superscript) and rules that open regions require that the level of the judgement is higher than the level of the region being opened. The purpose of the levels is to ensure that a region can never be opened twice in a single branch of the proof tree, which could unsoundly duplicate resources. The rules that open regions enforce this by requiring the level of the conclusion $(\lambda + 1)$ to be above the level of the region (λ), which is also the level of the premiss. For our examples, the level of each module's regions just needs to be greater than the levels of modules that it uses.

In all of our examples, the atomicity context describes an update to a single region. In the logic, there is no need to restrict in this way, and an atomicity context $\mathcal{A}$ may describe updates to multiple regions (although only one update to each). Both atomic and non-atomic judgements may have atomicity contexts.

The *frame rule*, as in separation logic, allows us to add the same resources to the pre- and postcondition, which are untouched by the command. Our frame rule separately adds to both the private and public parts. Note that the frame for the public part may be parametrised by the $\mathbb{V}$-bound variable x. (We exploited this fact in deriving the CAP lock specification.)

The *substitution rule* allows us to change the domain of $\mathbb{V}$-bound variables. A consequence of this rule is that we can instantiate $\mathbb{V}$-variables much like universally quantified variables, simply by choosing X' to be a single-element set.

The *atomicity weakening rule* allows us to convert private state from the conclusion into public state in the premiss.

The next three rules allow us to access the content of a shared region by using an atomic command. With all of the rules, the update to the shared region must be atomic, so its interpretation is in the public part in the premiss. (The region is in the public part in the conclusion also, but may be moved by applying atomicity weakening.)

The *open region* rule allows us to access the contents of a shared region without updating its abstract state. The command may change the concrete state of the region, so long as the abstract state is preserved. This is exemplified by its use in the DCAS proof in Fig. 7, where concretely the lock becomes locked, but the abstract state of the **MCAS** region is not affected.

The *use atomic* rule allows us to update the abstract state of a shared region. To do so, it is necessary to have a guard for the region being updated, such that the change in state is permitted by this guard according to the transition system associated with the region. This rule takes a $\mathbb{C}$ which (abstractly) atomically updates the region a from some state $x \in X$ to the state $f(x)$. It requires the guard G for the region, which allows the update according to the transition system, as established by one of the premisses. Another premiss states that the command $\mathbb{C}$ performs the update described by the transition system of region a in an atomic way. This allows us to conclude that the region a is updated atomically by the command $\mathbb{C}$. Note that the command is not operating at the same level of abstraction as the region a. Instead it is working at a lower level of abstraction, which means that if it is atomic at that level it will also be atomic at the region a level.

The *update region* rule similarly allows us to update the abstract state of a shared region, but this time the authority comes from the atomicity context instead of a guard. In order to preform such an update, the atomic update to the region must not already have happened, indicated by $a \Mapsto \blacklozenge$ in the precondition of the conclusion. In the postcondition, there are two cases: either the appropriate update happened, or no update happened. If it did happen, the new state of the region is some $z \in Q(x)$, and both x and z are recorded in the atomicity tracking resource. If it did not, then both the region's abstract state and the atomicity tracking resource are unchanged. The premiss requires the command to make a corresponding update to the concrete state of the region. The atomicity context and tracking resource are not present in the premiss; their purpose is rather to record information about the atomic update that is performed for use further down the proof tree.

It is necessary for the update region rule to account for both the case where the update occurs and where it does not. One might expect that the case with no update could be dealt with by the open region rule, and the results combined using a disjunction rule. However, a general disjunction rule is not sound for atomic triples. (If we have $\langle p_1 \rangle\, \mathbb{C}\, \langle q \rangle$ and $\langle p_2 \rangle\, \mathbb{C}\, \langle q \rangle$, we may not have $\langle p_1 \vee p_2 \rangle\, \mathbb{C}\, \langle q \rangle$ since $\mathbb{C}$ might rely on the environment not changing between p_1 and p_2.) The proof of the atomic specification for the spin lock uses the conditional nature of the update region rule.

Finally, we revisit the *make atomic* rule, which elaborates on the version presented in §2.1.3. As before, a guard in the conclusion must permit the update in accordance with the transition system for the region. This is replaced in the premiss by the atomicity context and atomicity tracking resource, which tracks the occurrence of the update. One difference is the inclusion of the private state, which is effectively preserved between the premiss and the conclusion. A second difference is the $\exists\!\!\!\exists$-binding of the resulting state of the atomic update. This allows the private state to reflect the result of the update.

4 Case Study: Concurrent Deque

We show how to use TaDA to specify a double-ended queue (deque) and verify a fine-grained implementation that makes use of MCAS. A deque has operations that allow elements to be inserted and removed from both ends of a list.

4.1 Abstract specification

We represent the deque state by the abstract predicate $\mathsf{Deque}(d, vs)$. It asserts that there is a deque at address d with list of elements vs. The `makeDeque()` operation creates an empty deque and returns its address. It has the following specification:

$$\lambda \vdash \left\{\mathsf{emp}\right\}\ \mathtt{d} := \mathtt{makeDeque}()\ \left\{\mathsf{Deque}(\mathtt{d}, [])\right\}$$

Statement rules

$$\frac{\lambda; \mathcal{A} \vdash \big\{P\big\}\ \mathbb{C}_1\ \big\{R\big\} \quad \lambda; \mathcal{A} \vdash \big\{R\big\}\ \mathbb{C}_2\ \big\{Q\big\}}{\lambda; \mathcal{A} \vdash \big\{P\big\}\ \mathbb{C}_1;\ \mathbb{C}_2\ \big\{Q\big\}}$$

$$\frac{\lambda; \mathcal{A} \vdash \big\{P \land \mathbb{B}\big\}\ \mathbb{C}_1\ \big\{Q\big\} \quad \lambda; \mathcal{A} \vdash \big\{P \land \neg\mathbb{B}\big\}\ \mathbb{C}_2\ \big\{Q\big\}}{\lambda; \mathcal{A} \vdash \big\{P\big\}\ \mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \big\{Q\big\}}$$

$$\frac{}{\vdash \big\{\mathtt{x} = n\big\}\ \mathtt{x} := \mathbb{B}\ \big\{\mathtt{x} = \mathbb{B}[n/\mathtt{x}]\big\}}$$

$$\frac{}{\vdash \forall\!\!\!\forall n.\, \big\langle \mathbb{E} \mapsto n \land \mathtt{x} = m \big\rangle\ \mathtt{x} := [\mathbb{E}]\ \big\langle \mathbb{E}[m/\mathtt{x}] \mapsto n \land \mathtt{x} = n \big\rangle}$$

$$\frac{}{\vdash \big\langle \exists n.\, \mathbb{E}_1 \mapsto n \big\rangle\ [\mathbb{E}_1] := \mathbb{E}_2\ \big\langle \mathbb{E}_1 \mapsto \mathbb{E}_2 \big\rangle}$$

$$\frac{}{\vdash \Big\{\textsf{emp}\Big\}\ \mathtt{x} := \mathtt{alloc}(\mathbb{E})\ \Big\{\exists y.\, \mathtt{x} = y \land \mathop{\circledast}\limits_{0 \le i \le \mathbb{E}-1} (y+i) \mapsto -\Big\}}$$

$$\frac{}{\vdash \forall\!\!\!\forall n.\, \left\langle \mathbb{E}_1 \mapsto n \right\rangle\ \mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)\ \left\langle \begin{array}{c} (n = \mathbb{E}_2 \land \mathtt{x} = 1 \land \mathbb{E}_1 \mapsto \mathbb{E}_3) \lor \\ (n \neq \mathbb{E}_2 \land \mathtt{x} = 0 \land \mathbb{E}_1 \mapsto n) \end{array} \right\rangle}$$

Weakening rules

$$\frac{\lambda_1 \le \lambda_2 \quad \lambda_1; \mathcal{A} \vdash \forall\!\!\!\forall x \in X.\, \big\langle p(x) \big\rangle\ \mathbb{C}\ \big\langle q(x) \big\rangle}{\lambda_2; \mathcal{A} \vdash \forall\!\!\!\forall x \in X.\, \big\langle p(x) \big\rangle\ \mathbb{C}\ \big\langle q(x) \big\rangle}$$

$$\frac{\lambda; \mathcal{A} \vdash \forall\!\!\!\forall x \in X.\, \big\langle p_p \,\big|\, p(x) \big\rangle\ \mathbb{C}\ \big\langle q_p(x) \,\big|\, q(x) \big\rangle}{\forall x \in X.\, \lambda; \mathcal{A} \vdash \big\{p_p * p(x)\big\}\ \mathbb{C}\ \big\{q_p(x) * q(x)\big\}}$$

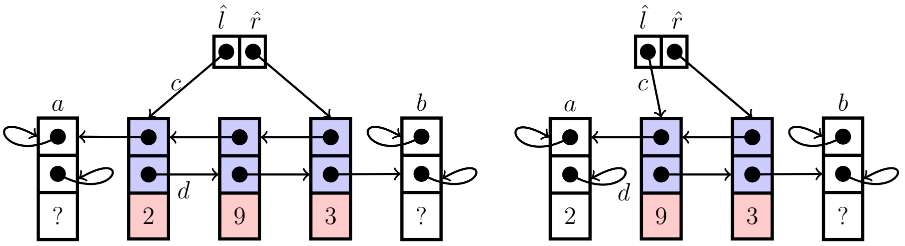

Figure 9: Examples of a deque before and after performing `popLeft`, which uses `dcas` to updated pointers c and d.

The operations `pushLeft`(d, v) and `popLeft`(d) are specified to update the state of the deque atomically:

$$\lambda \vdash \Forall vs.\ \left\langle \mathsf{Deque}(\mathtt{d}, vs) \right\rangle\ \mathtt{pushLeft}(\mathtt{d}, \mathtt{v})\ \left\langle \mathsf{Deque}(\mathtt{d}, \mathtt{v} : vs) \right\rangle$$

$$\lambda \vdash \Forall vs. \quad \begin{array}{c} \left\langle \mathsf{Deque}(\mathtt{d}, vs) \right\rangle \\ \mathtt{v} := \mathtt{popLeft}(\mathtt{d}) \\ \left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathtt{v} = 0 \wedge \mathsf{Deque}(\mathtt{d}, vs) \\ \underline{\mathbf{else}}\ vs = v : vs' \wedge \mathtt{v} = v \wedge \mathsf{Deque}(\mathtt{d}, vs') \end{array} \right\rangle \end{array}$$

The `pushLeft`(d, v) operation adds the value `v` to the left of the deque. The `popLeft`(d) operation tries to remove an element from the left end of the deque. However, if the deque is empty, then it returns 0 and does not change its state. Otherwise, it removes the element at the left, updating the state of the deque, and returns the removed valued. The `pushRight` and `popRight` operations have analogous specifications, operating on the right end of the deque.

4.2 The “Snark” Linked-list Deque Implementation

We consider an implementation that represents the deque as a doubly-linked list of nodes, based on *Snark* [6]. An example of the shape of the data structure is shown in Fig. 9. Each node consists of a left link pointer, a right link pointer, and a value. There are two anchor variables, *left hat* and *right hat* (nodes $\hat{l}$ and $\hat{r}$ in the figure), that generally point to the leftmost node and the rightmost node in the list, except when the deque is empty. When the deque is not empty, its leftmost node's left link points to a so-called *dead* node — a node whose left and right links point to itself (e.g. node a in the figure). Symmetrically, the rightmost node's right link points to a dead node. When the deque is empty, then the left hat and the right hat point to dead nodes.

We will focus on the `popLeft` operation. It first reads the left hat. If it points to a dead node, the list might be empty. It is necessary to recheck the left hat to confirm, since the node might have died since the left hat was first read. If the deque is indeed empty, the operation returns 0; otherwise it is restarted. If the left node is not dead, it tries to atomically update the left hat to point to the node to its right, and, at the same time, update the left node to be dead. (This could fail, in which case the operation restarts.) An example of such update is shown in Fig. 9. In order to update three pointers atomically, the implementation makes use of the `3cas` command described in §2.2.

To verify the `popLeft`, we introduce a new region type, **Deque**. The region has two parameters, d standing for the deque address and L for the MCAS address. There is only one non-empty guard for the region, named G. We represent the abstract state by a tuple (ns, ds) where: ns is a list of pairs of node addresses and values, the values representing the elements stored in the deque; and ds is a list of pairs of nodes addresses and values that were part of the deque, but are now dead. In order to change the abstract state of the deque, we require the guard G. The labelled transition system is as follows:

$$\begin{array}{l}
[\mathrm{G}]_a : \forall n, v, ns, ds.\, (ns, ds) \rightsquigarrow ((n, v) : ns, ds) \\
[\mathrm{G}]_a : \forall n, v, ns, ds.\, (ns, ds) \rightsquigarrow (ns : (n, v), ds) \\
[\mathrm{G}]_a : \forall n, v, ns, ds.\, ((n, v) : ns, ds) \rightsquigarrow (ns, (n, v) : ds) \\
[\mathrm{G}]_a : \forall n, v, ns, ds.\, (ns : (n, v), ds) \rightsquigarrow (ns, (n, v) : ds)
\end{array}$$

In order to provide an interpretation for the abstract state, we must first define the data structure invariant and will define predicates to aid the proof.

We use fields in this implementation. We shall use $E.\mathtt{field}$ as a shorthand for $E + \mathsf{offset}(\mathtt{field})$. Here, $\mathsf{offset}(\mathtt{left}) = 0$, $\mathsf{offset}(\mathtt{right}) = 1$, and $\mathsf{offset}(\mathtt{value}) = \mathsf{offset}(\mathtt{mcl}) = 2$.

A node at address n in the deque will make use of the MCAS interface:

$$\mathsf{node}(L, n, l, r, v) \equiv \mathsf{MCP}(L, n.\mathtt{left}, l) * \mathsf{MCP}(L, n.\mathtt{right}, r) * n.\mathtt{value} \mapsto v$$

Here l and r are the left and right link addresses. The L parameter is the address of MCAS lock, which is global for the whole deque. A dead node is defined as:

$$\mathsf{dead}(L, n, v) \equiv \mathsf{node}(L, n, n, n, v)$$

We also define a predicate to stand for the doubly-linked list that contains all the elements in the list, (i.e. the shaded nodes in the figure).

$$\begin{array}{l}
\mathsf{dlseg}(L, l, r, n, m, ns) \equiv ns = [] \land l = m \land r = n \lor \\
\qquad \exists v, ns', p.\, ns = (l : v) : ns' \land \mathsf{node}(L, l, n, p, v) * \mathsf{dlseg}(L, p, r, l, m, ns')
\end{array}$$

We define a predicate to include the dead nodes (ds):

$$\begin{array}{l}
\mathsf{dls}(L, l, r, ns, ds) \equiv \\
\qquad \exists a, b.\, (a, -), (b, -) \in ds \land \mathsf{dlseg}(L, l, r, a, b, ns) * \underset{(n,v) \in ds}{\circledast} \mathsf{dead}(L, n, v)
\end{array}$$

Note that there must be at least one dead node in ds.

Our last auxiliary predicate to represent the whole deque: the double linked list; the anchors left hat and right hat; and the reference to the MCAS interface.

$$\begin{array}{l}
\mathsf{deque}(d, L, ns, ds) \equiv \exists n, l, r.\, \mathsf{dls}(L, l, r, ns, ds) * \\
\qquad \mathsf{MCP}(L, d.\mathtt{left}, l) * \mathsf{MCP}(L, d.\mathtt{right}, r) * d.\mathtt{mcl} \mapsto L * \mathsf{MCL}(L)
\end{array}$$

We now define the interpretation of abstract states as follows:

$$I(\mathbf{Deque}_a(d, L, ns, ds)) \triangleq \mathsf{deque}(d, L, ns, ds)$$

We define the interpretation of the Deque predicate as follows:

$$\mathsf{Deque}(d, vs) \triangleq \exists a, L, ns, ds.\, \mathbf{Deque}_a(d, L, ns, ds) * [\mathrm{G}]_a \land vs = \mathsf{snds}(ns)$$

where $\mathsf{snds}(ns)$ maps the second projection over the list of pairs ns.

To prove the implementation against our atomic specifications, we use the "make atomic" rule again. We show the proof of the `popLeft` operation in Fig. 10.

This example shows that TaDA can scale to multiple levels of abstraction: the deque uses MCAS, which uses the lock, which is based on primitive atomic heap operations. This proof development would not be possible with CAP, since atomicity is central to the abstractions at each level. It would also not be possible using traditional approaches to linearisability, since separation of resources between and within abstraction layers is also crucial.

5 Semantics

5.1 Operational Semantics

The operational semantics of our language are given in Figs. 11 and 12.

5.2 Model

Guards and Guard Algebras. We assume a set Guard that will contain all guards that we might wish to use. A *guard algebra* $\zeta = (\mathcal{G}, \bullet, \mathbf{0}, \mathbf{1})$ consists of:

- a carrier set $\mathcal{G} \subseteq \mathsf{Guard}$,
- an associative, commutative partial binary operator $\bullet : \mathcal{G} \times \mathcal{G} \rightharpoonup \mathcal{G}$,
- an identity element $\mathbf{0} \in \mathcal{G}$, with $\mathbf{0} \bullet g = g$ for all $g \in \mathcal{G}$, and
- a maximal element $\mathbf{1} \in \mathcal{G}$, with $x \leq \mathbf{1}$ for all $g \in \mathcal{G}$,

where

$$x \leq y \overset{\text{def}}{\iff} \exists z.\, x \bullet z = y.$$

We denote by GAlg the set of all guard algebras.

Note that a guard algebra is a separation algebra (in the sense of [3]) with a single unit, $\mathbf{0}$.

Abstract States and Transition Systems. We assume a set AState that will contain all abstract region states that we might wish to use. For a given guard algebra ζ, a *guard-labelled transition system* $\mathcal{T} : \mathcal{G}_\zeta \rightarrow_{mon} \mathcal{P}(\mathsf{AState} \times \mathsf{AState})$ is a mapping from guards to relations. The mapping is monotone with respect to the resource ordering ($\leq_\zeta$) and subset ordering ($\subseteq$), meaning that having more guard resource permits more transitions. Although we make no restriction on the transition relation, in general, we shall use the reflexive-transitive closure $\mathcal{T}(g)^*$. We denote by ASTS_ζ the set of all ζ-labelled transition systems.

Abstract Region Types. We assume a set RTName of region type names. An abstract region typing

$$t \in \mathsf{ARType} \overset{\text{def}}{=} \mathsf{RTName} \rightarrow \coprod_{\zeta \in \mathsf{GAlg}} \mathsf{ASTS}_\zeta$$

maps region type names to pairs of guard algebras and guard-labelled transition systems.

```
$\forall vs.$
$\langle \mathsf{Deque}(\mathtt{d}, vs) \rangle$
  abstract; quantify $a$, $L$, $ns$, $ds$
  $\langle \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * [\mathrm{G}]_a \land vs = \mathsf{snds}(ns) \rangle$
    make atomic
    $a : (ns, ds) \rightsquigarrow \underline{\mathbf{if}}\ ns = []\ \underline{\mathbf{then}}\ (ns, ds)\ \underline{\mathbf{else}}\ (ns', (n, v) : ds) \land ns = (n, v) : ns' \vdash$
      $\{\exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge\}$
      L := [d.mcl];
      while (true) {
        $\{\exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge \land \mathtt{L} = L\}$
        lh := read(L, l.left); lhR := read(L, lh.right); lhL := read(L, lh.left);
        $\left\{ \begin{array}{l} \exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge \land \mathtt{L} = L \land \\ \underline{\mathbf{if}}\ \mathtt{lh} = \mathtt{lhL}\ \underline{\mathbf{then}}\ (\mathtt{lh}, -) \in ds \\ \quad \underline{\mathbf{else}}\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns \mathbin{+\!\!+} ds \end{array} \right\}$
        if (lhL = lh) { // left hat seems dead
          $\{\exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge \land \mathtt{L} = L \land (\mathtt{lhL}, -) \in ds\}$
            update region
            $\forall ns, ds.$
            $\langle \mathsf{deque}(\mathtt{d}, L, ns, ds) \land \mathtt{L} = L \land (\mathtt{lhL}, -) \in ds \rangle$
            lh2 := read(L, d.left);
            $\left\langle \begin{array}{l} \mathsf{deque}(\mathtt{d}, L, ns, ds) \land \mathtt{L} = L \land \\ (\mathtt{lh2} = \mathtt{lhL} \rightarrow ns = []) \end{array} \right\rangle$
          $\left\{ \begin{array}{l} \exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) \land \mathtt{L} = L \land \\ \underline{\mathbf{if}}\ \mathtt{lh2} = \mathtt{lhL}\ \underline{\mathbf{then}}\ a \Mapsto ([], ds), ([], ds)\ \underline{\mathbf{else}}\ a \Mapsto \blacklozenge \end{array} \right\}$
          if (lh2 = lhL) { // left hat confirmed dead
            return 0;
            $\{\exists ds.\, \mathsf{ret} = 0 * a \Mapsto ([], ds), ([], ds)\}$
          } // left hat not dead — try again
        } else {
          $\left\{ \begin{array}{l} \exists ns, ds.\, \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge \land \mathtt{L} = L \land \\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns \mathbin{+\!\!+} ds \end{array} \right\}$
            update region
            $\forall ns, ds.$
            $\left\langle \begin{array}{l} \mathsf{deque}(\mathtt{d}, L, ns, ds) \land \mathtt{L} = L \land \\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns \mathbin{+\!\!+} ds \end{array} \right\rangle$
            b := 3cas(L, d.left, lh.right, lh.left, lh, lhR, lhL, lhR, lh, lh);
            $\left\langle \begin{array}{l} \exists ns', v.\, \underline{\mathbf{if}}\ \mathtt{b} = 1\ \underline{\mathbf{then}} \left( \begin{array}{c} \mathsf{deque}(\mathtt{d}, L, ns', (\mathtt{lh}, v) : ds) \land \\ \mathtt{L} = L \land (\mathtt{lh}, v) \in ds \land ns = (\mathtt{lh}, v) : ns' \end{array} \right) \\ \quad \underline{\mathbf{else}}\ \mathsf{deque}(\mathtt{d}, L, ns, ds) \land \mathtt{L} = L \end{array} \right\rangle$
          $\left\{ \begin{array}{l} \exists ns, ds, v.\, \underline{\mathbf{if}}\ \mathtt{b} = 1\ \underline{\mathbf{then}} \left( \begin{array}{r} a \Mapsto ((\mathtt{lh}, \mathtt{v}) : ns, ds), (ns, (\mathtt{lh}, \mathtt{v}) : ds) \\ \land \mathtt{L} = L \land (\mathtt{lh}, v) \in ds \end{array} \right) \\ \quad \underline{\mathbf{else}}\ \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * a \Mapsto \blacklozenge \land \mathtt{L} = L \end{array} \right\}$
          if (b = 1) {
            v := [lh.value]; return v;
            $\{\exists ns, ds.\, \mathsf{ret} = \mathtt{v} * a \Mapsto ((\mathtt{lh}, \mathtt{v}) : ns, ds), (ns, (\mathtt{lh}, \mathtt{v}) : ds)\}$
      } } }
  $\left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathsf{ret} = 0 * \mathbf{Deque}_a(\mathtt{d}, L, ns, ds) * [\mathrm{G}]_a \\ \quad \underline{\mathbf{else}} \left( \begin{array}{c} \exists ns', v.\, ns = (n, v) : ns' \land \mathsf{ret} = v * \\ \mathbf{Deque}_a(\mathtt{d}, L, ns', (n, v) : ds) * [\mathrm{G}]_a \land vs' = \mathsf{snds}(ns') \end{array} \right) \end{array} \right\rangle$
$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathsf{ret} = 0 * \mathsf{Deque}(\mathtt{d}, vs) \\ \quad \underline{\mathbf{else}}\ \exists vs', v.\, vs = v : vs' \land \mathsf{ret} = v * \mathsf{Deque}(\mathtt{d}, vs') \end{array} \right\rangle$
```

Figure 10: Proof of the `popLeft` implementation.

$$\frac{\langle s, \mathbb{C}_1\rangle \xrightarrow{\alpha} \langle s', \mathbb{C}_1'\rangle}{\langle s, \mathbb{C}_1; \mathbb{C}_2\rangle \xrightarrow{\alpha} \langle s', \mathbb{C}_1'; \mathbb{C}_2\rangle} \qquad \frac{}{\langle s, \texttt{skip}; \mathbb{C}\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}\rangle}$$

$$\frac{\mathcal{B}\llbracket \mathbb{B} \rrbracket_s}{\langle s, \texttt{if } (\mathbb{B})\ \mathbb{C}_1 \texttt{ else } \mathbb{C}_2\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}_1\rangle} \qquad \frac{\neg\mathcal{B}\llbracket \mathbb{B} \rrbracket_s}{\langle s, \texttt{if } (\mathbb{B})\ \mathbb{C}_1 \texttt{ else } \mathbb{C}_2\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}_2\rangle}$$

$$\frac{\mathcal{B}\llbracket \mathbb{B} \rrbracket_s}{\langle s, \texttt{while } (\mathbb{B})\ \mathbb{C}\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}; \texttt{while } (\mathbb{B})\ \mathbb{C}\rangle} \qquad \frac{\neg\mathcal{B}\llbracket \mathbb{B} \rrbracket_s}{\langle s, \texttt{while } (\mathbb{B})\ \mathbb{C}\rangle \xrightarrow{\mathsf{id}} \langle s, \texttt{skip}\rangle}$$

$$\frac{\mathcal{E}\llbracket \overrightarrow{\mathbb{E}} \rrbracket_s = s'(\mathit{vars}(\gamma(f)))}{\langle s, \texttt{x} := f(\overrightarrow{\mathbb{E}})\rangle \xrightarrow{\mathsf{id}} \langle s, \texttt{x} := \langle s', \mathit{code}(\gamma(f))\rangle\rangle}$$

$$\frac{}{\langle s, \texttt{do } \mathbb{C} \texttt{ while } (\mathbb{B})\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}; \texttt{while } (\mathbb{B})\ \mathbb{C}\rangle} \qquad \frac{\tau \xrightarrow{\alpha} \tau'}{\langle s, \texttt{x} := \tau\rangle \xrightarrow{\alpha} \langle s, \texttt{x} := \tau'\rangle}$$

$$\frac{}{\langle s, \texttt{x} := \langle s', \texttt{return } \mathbb{E}; \mathbb{C}\rangle\rangle \xrightarrow{\mathsf{id}} \langle s[\texttt{x} \mapsto \mathcal{E}\llbracket \mathbb{E} \rrbracket_{s'}], \texttt{skip}\rangle}$$

$$\frac{}{\langle s, \texttt{x} := \mathbb{E}\rangle \xrightarrow{\mathsf{id}} \langle s[\texttt{x} \mapsto \mathcal{E}\llbracket \mathbb{E} \rrbracket_s], \texttt{skip}\rangle}$$

$$\frac{}{\langle s, \texttt{x} := [\mathbb{E}]]\rangle \xrightarrow{\mathsf{read}(\mathcal{E}\llbracket \mathbb{E} \rrbracket_s, v)} \langle s[\texttt{x} \mapsto v], \texttt{skip}\rangle}$$

$$\frac{}{\langle s, [\mathbb{E}_1] := \mathbb{E}_2\rangle \xrightarrow{\mathsf{write}(\mathcal{E}\llbracket \mathbb{E}_1 \rrbracket_s, \mathcal{E}\llbracket \mathbb{E}_2 \rrbracket_s)} \langle s, \texttt{skip}\rangle}$$

$$\frac{}{\langle s, \texttt{x} := \texttt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)\rangle \xrightarrow{\mathsf{cas}(\mathcal{E}\llbracket \mathbb{E}_1 \rrbracket_s, \mathcal{E}\llbracket \mathbb{E}_2 \rrbracket_s, \mathcal{E}\llbracket \mathbb{E}_3 \rrbracket_s, v)} \langle s[\texttt{x} \mapsto v], \texttt{skip}\rangle}$$

$$\frac{}{\langle s, \texttt{x} := \texttt{alloc}(\mathbb{E})\rangle \xrightarrow{\mathsf{alloc}(\mathcal{E}\llbracket \mathbb{E} \rrbracket, v)} \langle s[\texttt{x} \mapsto v], \texttt{skip}\rangle}$$

$$\frac{}{\langle s, \texttt{fork } f(\overrightarrow{\mathbb{E}})\rangle \xrightarrow{\mathsf{spawn}(f, \mathcal{E}\llbracket \overrightarrow{\mathbb{E}} \rrbracket_s)} \langle s, \texttt{skip}\rangle}$$

Figure 11: Small-step operational semantics for threads, $\xrightarrow{\alpha}_\gamma$. The parameter γ is fixed, and not shown.

$$\frac{}{T \parallel \langle s, \texttt{skip}\rangle \xrightarrow{\mathsf{id}} T} \qquad \frac{}{T \parallel \langle s, \texttt{return}\ \mathbb{E}; \mathbb{C}\rangle \xrightarrow{\mathsf{id}} T}$$

$$\frac{\tau \xrightarrow{\mathsf{spawn}(f, \overrightarrow{v})} \tau' \quad s(\mathit{vars}(\gamma(f))) = \overrightarrow{v}}{T \parallel \tau \xrightarrow{\mathsf{id}} T \parallel \tau' \parallel \langle s, \mathit{code}(\gamma(f))\rangle}$$

$$\frac{\tau \xrightarrow{\alpha} \tau' \quad \alpha \notin \{\mathsf{spawn}(f, \overrightarrow{v}) \mid f, \overrightarrow{v}\}}{T \parallel \tau \xrightarrow{\alpha} T \parallel \tau'}$$

Figure 12: Small-step operational semantics for thread pools, $\xrightarrow{\alpha}_{\gamma}$.

Heaps. We assume a set Val of program values, which includes a set $\mathsf{Loc} \subseteq \mathsf{Val}$ of program locations. A heap $h \in \mathsf{Heap} \stackrel{\text{def}}{=} \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$ is a finite partial function from locations to values. Heaps form a separation algebra $(\mathsf{Heap}, \uplus, \emptyset)$, where $\uplus$ is the disjoint union of partial functions, and $\emptyset$ is the partial function with the empty domain. Heaps are ordered by resource ordering: $h_1 \leq h_2 \stackrel{\text{def}}{\iff} \exists h_3.\, h_1 \uplus h_3 = h_2$.

Abstract Predicates. We assume a set APName of abstract predicate names. An abstract predicate $\mathsf{a} \in \mathsf{APName} \times \mathsf{Val}^*$ consists of an abstract predicate name and a list of parameters. An abstract predicate bag $b \in \mathsf{APBag} \stackrel{\text{def}}{=} \mathcal{M}_{\mathit{fin}}(\mathsf{APName} \times \mathsf{Val}^*)$ is a finite multiset of abstract predicates. Abstract predicate bags form a separation algebra $(\mathsf{APBag}, \cup, \emptyset)$, where $\cup$ is multiset union, and $\emptyset$ is the empty multiset. Abstract predicate bags are ordered by the usual subset order $\subseteq$, which corresponds to the resource order.

Levels. A level $\lambda \in \mathsf{Level} \stackrel{\text{def}}{=} \mathbb{N}$ is simply a natural number. Levels are ordered by the usual well-founded ordering on natural numbers.

Note. It would be possible to take the levels from a more general well-founded order. This might be useful if we need some kind of unbounded nesting of regions. I cannot see any obvious use for this, though.

Region Assignments. We assume a (countably infinite) set of region identifiers, RId. A region assignment $r \in \mathsf{RAss} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{\mathit{fin}} \mathsf{Level} \times \mathsf{RTName} \times \mathsf{Val}^*$ is a finite partial function from region identifiers to levels and parametrised region type names. Region assignments are ordered by extension ordering: $r_1 \leq r_2 \stackrel{\text{def}}{\iff} \forall a \in \mathrm{dom}(r_1).\, r_2(a) = r_1(a)$.

For the following semantic definitions, we assume a fixed abstract region typing $t \in \mathsf{ARType}$.

Guard Assignments. Given a region assignment, r, a guard assignment

$$\gamma \in \mathsf{GAss}_r \stackrel{\text{def}}{=} \prod_{a \in \mathrm{dom}(r)} \mathcal{G}_{\zeta(t(r(a)))}$$

is a mapping from the regions declared in r to guards of the appropriate type for each region. Guard assignments form a separation algebra $(\mathsf{GAss}_r, \bullet, \lambda a.\, \mathbf{0}_{\zeta(t(r(a)))})$ where $\bullet$ is the pointwise lift of the guard combination operators:

$$\gamma_1 \bullet \gamma_2 \stackrel{\text{def}}{=} \lambda a.\, \gamma_1(a) \bullet \gamma_2(a)$$

For $\gamma_1 \in \mathsf{GAss}_{r_1}$, $\gamma_2 \in \mathsf{GAss}_{r_2}$ with $r_1 \leq r_2$, guards assignments are ordered pointwise-extensionally:

$$\gamma_1 \leq \gamma_2 \overset{\text{def}}{\iff} \forall a \in \operatorname{dom}(\gamma_1).\, \gamma_1(a) \leq \gamma_2(a).$$

Region States. Given a region assignment, r, a region state

$$\rho \in \mathsf{RState}_r \stackrel{\text{def}}{=} \operatorname{dom}(r) \to \mathsf{AState}$$

is a mapping from the regions declared in r to abstract states. For $\rho_1 \in \mathsf{RState}_{r_1}$, $\rho_2 \in \mathsf{RState}_{r_2}$ with $r_1 \leq r_2$, region states are ordered extensionally: $\rho_1 \leq \rho_2 \overset{\text{def}}{\iff} \forall a \in \operatorname{dom}(\rho_1).\, \rho_1(a) = \rho_2(a)$.

Worlds. A *world*

$$w \in \mathsf{World} \stackrel{\text{def}}{=} \coprod_{r \in \mathsf{RAss}} (\mathsf{Heap} \times \mathsf{APBag} \times \mathsf{GAss}_r \times \mathsf{RState}_r)$$

consists of a region assignment, a heap, an abstract predicate bag, a guard assignment and a region state.

Worlds can be combined, provided they agree on the region assignment and region state, by combining the remaining components in the appropriate separation algebras. Thus, worlds form a (multi-unit) separation algebra $(\mathsf{World}, \cdot, \mathsf{emp})$ where

$$(r, h_1, b_1, \gamma_1, \rho) \cdot (r, h_2, b_2, \gamma_2, \rho) \stackrel{\text{def}}{=} (r, h_1 \uplus h_2, b_1 \cup b_2, \gamma_1 \bullet \gamma_2, \rho)$$

$$\mathsf{emp} \stackrel{\text{def}}{=} \left\{ (r, \emptyset, \emptyset, \lambda a.\, \mathbf{0}_{\zeta(t(r(a)))}, \rho) \;\middle|\; r \in \mathsf{RAss}, \rho \in \mathsf{RState}_r \right\}$$

Worlds are also ordered by the product order. If $w_1 \leq w_2$, then w_2 may be obtained from w_1 by introducing new regions (with arbitary associated type name and state) and adding heap, abstract-predicate and guard resources.

World Predicates. A world predicate $p \in \mathsf{WPred} \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathsf{World})$ is a set of worlds that is upwards closed with respect to the world ordering. That is, if $w \in p$ and $w \leq w'$ then $w' \in p$.

The composition operator on worlds is lifted to world predicates:

$$p_1 * p_2 \stackrel{\text{def}}{=} \{ w \mid \exists w_1 \in p_1, w_2 \in p_2.\, w = w_1 \bullet w_2 \}$$

(That the results is upwards closed is not difficult to check: any extension to the composition of two worlds can be tracked back and applied to one of the components.) The $*$ operator is associative and commutative with identity World. To denote $*$ iterated over a finite set X, we write $\circledast_{x \in X}\, p(x)$.

Worlds with Atomic Tracking. The atomic tracking separation algebra is defined to be $((\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\}, \bullet, (\mathsf{AState} \times \mathsf{AState}) \cup \{\lozenge\})$, where $\bullet$ is defined by

$$\blacklozenge \bullet \lozenge = \blacklozenge = \lozenge \bullet \blacklozenge$$
$$\lozenge \bullet \lozenge = \lozenge$$
$$(x, y) \bullet (x, y) = (x, y)$$

and undefined in all other cases. The resource ordering on this separation algebra is characterised by the two rules: $k \leq k$ (for all $k \in (\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\}$) and $\lozenge \leq \blacklozenge$.

Given a finite set of region identifiers $\mathcal{R} \subseteq_{\mathrm{fin}} \mathsf{RId}$, a world with atomic tracking $\varphi \in \mathsf{AWorld}_{\mathcal{R}} \stackrel{\text{def}}{=} \mathsf{World} \times (\mathcal{R} \to (\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\})$ consists of a world together with a mapping that associates atomic tracking resources with each region in $\mathcal{R}$. The mapping records if an atomic update has taken place on a region, and, if so, what state change the region underwent in the update. Specifically, $\lozenge$ and $\blacklozenge$ record that the atomic update has not yet happened, while (x, y) records that the update has happened, and it entailed updating the abstract state from x to y. The difference between $\lozenge$ and $\blacklozenge$ is that $\blacklozenge$ embodies a right to perform the update, while $\lozenge$ does not.

By lifting $\bullet$ to maps, the maps form a separation algebra. Consequently, by combining the operators of its components, $\mathsf{AWorld}_{\mathcal{R}}$ is also an ordered separation algebra.

We consider that $\mathsf{World} = \mathsf{AWorld}_{\emptyset}$.

As with worlds, we consider predicates over worlds with atomic tracking $p \in \mathsf{AWPred}_{\mathcal{R}} \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathsf{AWorld}_{\mathcal{R}})$ to be upwards-closed sets. These predicates similarly have a $*$ operator.

Atomicity Context. An atomicity context $\mathcal{A} \in \mathsf{AContext} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{\mathit{fin}} \mathcal{P}(\mathsf{AState} \times \mathsf{AState})$ is a (finite) partial mapping from region identifiers to relations on abstract states. In the context of proving that an operation is abstractly atomic, the atomicity context records the abstract operation to be performed. This has implications in terms of both how the thread performing the operation and the environment can update the region mentioned in the context.

Rely Relation. Interference by the environment is abstracted by the rely relation. For a given atomicity context $\mathcal{A} \in \mathsf{AContext}$, with $\mathcal{R} = \mathrm{dom}(\mathcal{A})$, the rely relation $\mathrm{R}_{\mathcal{A}} \subseteq \mathsf{AWorld}_{\mathcal{R}} \times \mathsf{AWorld}_{\mathcal{R}}$ is the smallest reflexive-transitive relation that satisfies the following rules:

$$\frac{g \mathbin{\#} g' \quad (s, s') \in \mathcal{T}_{t(n)}(g')^{*} \quad (d(a) \in \{\blacklozenge, \lozenge\} \Rightarrow s' \in \mathrm{dom}(\mathcal{A}(a)))}{(r[a \mapsto n], h, b, \gamma[a \mapsto g], \rho[a \mapsto s], d) \; \mathrm{R}_{\mathcal{A}} \; (r[a \mapsto n], h, b, \gamma[a \mapsto g], \rho[a \mapsto s'], d)}$$

$$\frac{(s, s') \in \mathcal{A}(a)}{(r[a \mapsto n], h, b, \gamma, \rho[a \mapsto s], d[a \mapsto \lozenge]) \; \mathrm{R}_{\mathcal{A}} \; (r[a \mapsto n], h, b, \gamma, \rho[a \mapsto s'], d[a \mapsto (s, s')])}$$

The first rule expresses that the environment may make any update to a region for which it can have a guard that permits it in the corresponding transition system. (It can only have such a guard if it is compatible with the guard held by the thread, expressed as $g \mathbin{\#} g'$.) The exception to this is that, if an atomic update is pending then the environment must not take the state outside of those on which the atomic operation is set to perform.

The second rule expresses that having the ⧫ entitles one to perform an update corresponding to that expressed in the atomicity context.

Note that interference is explicitly confined to the shared regions and atomic tracking resources. Furthermore, extending the atomicity context decreases the possible interference of the environment.

Stable Predicates. Given an atomicity context $\mathcal{A} \in \mathsf{AContext}$, the stable predicates are those which are closed under the associated rely relation. That is, we define the stability judgement as follows:

$$\mathcal{A} \vDash p \; \mathsf{stable} \overset{\text{def}}{\iff} \mathrm{R}_{\mathcal{A}} \, (p) \subseteq p.$$

We call the stable predicates *views* (as in [3]) and denote the set of views (in atomicity context $\mathcal{A}$) by $\mathsf{View}_{\mathcal{A}}$. We drop the subscript when the empty atomicity context is intended.

If $\mathcal{A}'$ is an extension of $\mathcal{A}$, we have a coercion from $\mathsf{View}_{\mathcal{A}}$ to $\mathsf{View}_{\mathcal{A}'}$ by extending the atomicity tracking component for the additional regions in every possible way.

Stable predicates are closed under $*$. That is

$$\mathcal{A} \vDash p \; \mathsf{stable} \wedge \mathcal{A} \vDash q \; \mathsf{stable} \implies \mathcal{A} \vDash p * q \; \mathsf{stable}$$

Region Interpretation. A region interpretation $I \in \mathsf{RInterp} \overset{\text{def}}{=} \mathsf{Level} \times \mathsf{RTName} \times \mathsf{Val}^* \times \mathsf{RId} \times \mathsf{AState} \to \mathsf{View}$ associates a view with each abstract state of each parametrised region type. The parameters are used to specify, for example, the address of a datastructure contained in the region. The region identifier is often a necessary parameter as it is common for a region interpretation to refer to guards for the region.[1]

Abstract Predicate Interpretation. An abstract predicate interpretation $\iota \in \mathsf{APInterp} \overset{\text{def}}{=} \mathsf{APName} \times \mathsf{Val}^* \to \mathsf{View}$ associates a view with each abstract predicate.

For the following, assume a fixed region interpretation I and abstract predicate interpretation ι.

Region Collapse. Given a level $\lambda \in \mathsf{Level}$, the region collapse of a world $\varphi \in \mathsf{AWorld}_{\mathcal{R}'}$ is a set of worlds given by:

$$\varphi{\downarrow_\lambda} \overset{\text{def}}{=} \left\{ \varphi \cdot (w', \emptyset) \;\middle|\; w' \in \circledast_{\{a \;\mid\; \exists \lambda' < \lambda . r_\varphi(a) = (\lambda', -, -)\}} I(r_\varphi(a), a, \rho_\varphi(a)) \right\}$$

This operation is lifted to predicates in a straightforward manner: $p{\downarrow_\lambda} \overset{\text{def}}{=} \bigcup_{\varphi \in p} \varphi{\downarrow_\lambda}$.

[1] Here, we have avoided having region interpretations directly referring to region interpretations. Impredicative CAP [16] does support this by constructing the relevant domains in the topos of trees. We opt for a simpler, if less powerful, alternative: breaking self-reference by indirection through region type names.

Abstract Predicate Collapse. The one-step abstract predicate collapse of a world is a set of worlds given by:

$$(r, h, b, \gamma, \rho, d)\lfloor_1 \overset{\text{def}}{=} \left\{ (r, h, \emptyset, \gamma, \rho, d) \cdot (w, \emptyset) \;\middle|\; w \in \mathop{\circledast}_{a \in b} \iota(a) \right\}$$

This is lifted to predicates: $p\lfloor_1 \overset{\text{def}}{=} \bigcup_{\varphi \in p} \varphi\lfloor_1$. The one-step collapse is iterated to give the multi-step collapse: $p\lfloor_{n+1} \overset{\text{def}}{=} (p\lfloor_n)\lfloor_1$.

The abstract predicate collapse of a predicate applies the multi-step collapse to collapse all abstract predicates:

$$p\lfloor \overset{\text{def}}{=} \{\varphi \mid \exists n.\, \varphi \in p\lfloor_n \wedge b_\varphi = \emptyset\}$$

Note. This approach to interpreting abstract predicates is different from the usual one. It effectively gives a step-indexed interpretation to the predicates: the concrete interpretation is given by the finite unfoldings. If a predicate cannot be made fully concrete by finite unfolding, then its semantics will be false.

Reification. The reification operation on worlds collapses the regions and the abstract predicates, and then considers only the heap portion:

$$\lfloor \varphi \rfloor_\lambda \overset{\text{def}}{=} \{h_{\varphi'} \mid \varphi' \in \varphi{\downarrow_\lambda}\lfloor\}$$

This operation is lifted to predicates in the usual manner.

Guarantee Relation. Given a level $\lambda \in \mathsf{Level}$, and atomicity context $\mathcal{A} \in \mathsf{AContext}$, the guarantee relation $\mathrm{G}_{\lambda;\mathcal{A}} \subseteq \mathsf{AWorld}_{\mathcal{R}'} \times \mathsf{AWorld}_{\mathcal{R}'}$ is defined as:

$$\begin{aligned} \varphi \,\mathrm{G}_{\lambda;\mathcal{A}}\, \varphi' \overset{\text{def}}{\iff} & \forall a.\, (\exists \lambda' \geq \lambda.\, r_\varphi(a) = (\lambda', -, -)) \implies \rho_\varphi(a) = \rho_{\varphi'}(a) \wedge \\ & \forall a \in \operatorname{dom} \mathcal{A}. \left(\begin{array}{r} (d_\varphi(a) = d_{\varphi'}(a) \wedge \rho_\varphi(a) = \rho_{\varphi'}(a)) \vee \\ \left(\begin{array}{r} d_\varphi(a) = \blacklozenge \wedge d_{\varphi'}(a) = (\rho_\varphi(a), \rho_{\varphi'}(a)) \\ \wedge\, (\rho_\varphi(a), \rho_{\varphi'}(a)) \in \mathcal{A}(a) \end{array} \right) \end{array} \right) \end{aligned}$$

The guarantee relation enforces that regions with level λ or higher cannot be modified. It also enforces that regions mentioned in the atomicity context can only be updated using the atomicity context.

Note. It will be necessary to enforce that each execution step preserves regions above a certain level, because these regions will simply be dropped by the reification. If we didn't constrain them in this way, a thread could change them as it liked (resources permitting) without even making a concrete update!

5.2.1 Semantic Judgements

In the Views Framework [3], primitive atomic actions are abstracted to relations on views by means of an atomic satisfaction judgement. Here, we have an analogous judgement, but which is more complex as it expresses the role of an action in performing an abstractly-atomic operation. To express this role, we conceptually divide the view into a private and a public part. A thread is at liberty to do as it pleases with the private part (subject to preserving all stable frames). The public part, however, must be maintained invariant by the thread until it performs its abstract atomic action, at which point it updates the public part accordingly and thereafter loses access to it. The primitive atomic satisfaction judgement therefore incorporates five assertions: p_p, the precondition for the private part; p, the precondition for the public part; p'_p, the postcondition for the private part where the atomic update does not happen; q, the postcondition for the public part (when an atomic update does happen — otherwise p plays the role); and q_p, the postcondition for the private part where the atomic update does happen.

Definition 1 (Primitive Atomic Satisfaction Judgement)**.** The primitive atomic satisfaction judgement $\lambda; \mathcal{A} \vDash \langle p_p \mid p \rangle\ \alpha\ \langle p'_p \mid - \rangle + \langle q_p \mid q \rangle$, where $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, $\alpha \in \mathsf{AAction}$ and $p_p, p, p'_p, q, q_p \in \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$, is defined as:

$$\begin{aligned}
&\lambda; \mathcal{A} \vDash \langle p_p \mid p \rangle\ \alpha\ \langle p'_p \mid - \rangle + \langle q_p \mid q \rangle \overset{\text{def}}{\iff} \\
&\qquad \forall r \in \mathsf{View}_{\mathcal{A}}.\, \forall \varphi \in p_p * p * r.\, \forall h \in \lfloor \varphi \rfloor_\lambda.\, \forall h' \in [\![\alpha]\!](h). \\
&\qquad\qquad \exists \varphi'.\, \varphi\ \mathrm{G}_{\lambda;\mathcal{A}}\ \varphi' \wedge h' \in \lfloor \varphi' \rfloor_\lambda \wedge \varphi' \in (p'_p * p * r) \cup (q_p * q * r)
\end{aligned}$$

Definition 2 (Primitive Atomic Satisfaction Judgement)**.**

$$\begin{aligned}
&\lambda; \mathcal{A} \vDash \langle p \rangle \alpha \langle q \rangle \overset{\text{def}}{\iff} \\
&\qquad \forall r \in \mathsf{View}_{\mathcal{A}}.\, \forall \varphi \in p * r.\, \forall h \in \lfloor \varphi \rfloor_\lambda.\, \forall h' \in [\![\alpha]\!](h). \\
&\qquad\qquad \exists \varphi'.\, \varphi\ \mathrm{G}_{\lambda;\mathcal{A}}\ \varphi' \wedge h' \in \lfloor \varphi' \rfloor_\lambda \wedge \varphi' \in q * r.
\end{aligned}$$

Definition 3 (Semantic Judgement)**.** The semantic judgement

$$\lambda; \mathcal{A}; \Omega \vDash \forall\!\!\!\forall \mathbf{x} \in X.\, \langle p_p \mid p(\mathbf{x}) \rangle\ \mathbb{C}\ \exists\!\!\!\exists \mathbf{y} \in Y.\, \langle q_p(\mathbf{x}, \mathbf{y}) \mid q(\mathbf{x}, \mathbf{y}) \rangle$$

where

- $\lambda \in \mathsf{Level}$ is a level strictly greater than that of any region that will be affected by the program;
- $\mathcal{A} \in \mathsf{AContext}$ is the atomicity context, which constrains updates to regions on which an abstractly atomic update is to be performed;
- $\Omega \in X \times Y \to \mathsf{Val} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ is the postcondition on return, which is parametrised by the value returned;
- $p_p \in \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ is the private part of the precondition, which does not correspond to resources in some opened shared region, and is parametrised by the valuation of program variables;
- $p \in X \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ is the public part of the precondition, which may correspond to resources from some opened shared regions, and is parametrised by $\mathbf{x} \in X$ that tracks the precondition at the linearisation point;
- $\mathbb{C} \in \mathsf{Command}$ is the program under consideration;
- $q_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ is the private part of the postcondition, which is parametrised by $\mathbf{x} \in X$ that tracks the precondition at the linearisation point, by $\mathbf{y} \in Y$ that tracks the postcondition at the linearisation point, and by the valuation of program variables;

- $q \in X \times Y \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ is the public part of the postcondition, which is similarly parametrised by $\mathbf{x} \in X$ and $\mathbf{y} \in Y$,

is defined to be the most-general judgement that holds when the following conditions hold:

- For all $s, s' \in \mathsf{Store}$, $\mathbb{C}' \in \mathsf{Command}$, $\alpha \in \mathsf{AAction}$ with $\langle \mathbb{C}, s\rangle \xrightarrow{\alpha} \langle \mathbb{C}', s'\rangle$, for all $\mathbf{x} \in X$, there exist $p'_p \in \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$, $p''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ such that

$$\lambda; \mathcal{A} \vDash \big\langle p_p(s) * p(\mathbf{x})\big\rangle \alpha \big\langle p'_p(s') * p(\mathbf{x}) \vee \exists \mathbf{y} \in Q(\mathbf{x}).\, p''_p(\mathbf{x}, \mathbf{y}, s') * q(\mathbf{x}, \mathbf{y})\big\rangle$$
$$\lambda; \mathcal{A}; \Omega \vDash \forall\!\!\forall \mathbf{x} \in X.\, \langle p'_p | p(\mathbf{x})\rangle\ \mathbb{C}'\ \exists\!\!\exists \mathbf{y} \in Y.\, \langle q_p(\mathbf{x}, \mathbf{y}) | q(\mathbf{x}, \mathbf{y})\rangle,$$
$$\text{and for all } \mathbf{y} \in Q(\mathbf{x}),\ \lambda; \mathcal{A}; \Omega(\mathbf{x}, \mathbf{y}) \vDash \big\{p''_p(\mathbf{x}, \mathbf{y})\big\}\ \mathbb{C}'\ \big\{q_p(\mathbf{x}, \mathbf{y})\big\}\,.$$

- For all $s, s' \in \mathsf{Store}$, $\mathbb{C}' \in \mathsf{Command}$, f, $\overrightarrow{v}$ with $\langle \mathbb{C}, s\rangle \xrightarrow{\mathsf{fork}(f, \overrightarrow{v})} \langle \mathbb{C}', s'\rangle$, for all $\mathbf{x} \in X$, there exist $p'_p \in \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$, $p''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$ and $p_f \in \mathsf{Store} \to \mathsf{View}$ such that for all $s_f \in \mathsf{Store}$ with $s_f(\mathit{vars}(\gamma(f))) = \overrightarrow{v}$,

$$\lambda; \mathcal{A} \vDash \big\langle p_p(s) * p(\mathbf{x})\big\rangle \mathsf{id} \big\langle p'_p(s') * p_f(s_f) * p(\mathbf{x}) \vee \exists \mathbf{y} \in Q(\mathbf{x}).\, p''_p(s') * p_f(s_f) * q(\mathbf{x}, \mathbf{y})\big\rangle\,,$$
$$\lambda; \mathcal{A}; \Omega \vDash \forall\!\!\forall \mathbf{x} \in X.\, \langle p'_p | p(\mathbf{x})\rangle\ \mathbb{C}'\ \exists\!\!\exists \mathbf{y} \in Y.\, \langle q_p(\mathbf{x}, \mathbf{y}) | q(\mathbf{x}, \mathbf{y})\rangle,$$
$$\text{for all } \mathbf{y} \in Q(\mathbf{x}),\ \lambda; \mathcal{A}; \Omega(\mathbf{x}, \mathbf{y}) \vDash \big\{p''_p(\mathbf{x}, \mathbf{y})\big\}\ \mathbb{C}'\ \big\{q_p(\mathbf{x}, \mathbf{y})\big\}\,,$$
$$\text{and } \lambda; \emptyset; \mathsf{true} \vDash \big\{p_f\big\}\ \mathit{code}(\gamma(f))\ \big\{\mathsf{true}\big\}\,.$$

- If $\mathbb{C} = \texttt{skip}$ then, for all $s \in \mathsf{Store}$, $\mathbf{x} \in X$, there exists $\mathbf{y} \in Y$ such that

$$\lambda; \mathcal{A} \vDash \langle p_p(s) \mid p(\mathbf{x})\rangle\ \mathsf{id}\ \langle \mathsf{false} \mid -\rangle + \langle q_p(\mathbf{x}, \mathbf{y}, s) \mid q(\mathbf{x}, \mathbf{y})\rangle.$$

- If $\mathbb{C} = \texttt{return}\ \mathbb{E}; \mathbb{C}'$ then, for all $s \in \mathsf{Store}$, $\mathbf{x} \in X$, there exists $\mathbf{y} \in Y$ such that

$$\lambda; \mathcal{A} \vDash \langle p_p(s) \mid p(\mathbf{x})\rangle\ \mathsf{id}\ \langle \mathsf{false} \mid -\rangle + \langle \Omega(\mathbf{x}, \mathbf{y}, \mathcal{E}[\![\mathbb{E}]\!]_s) \mid q(\mathbf{x}, \mathbf{y})\rangle.$$

Here, we adopt the syntax $\lambda; \mathcal{A}; \Omega \vDash \big\{p\big\}\ \mathbb{C}\ \big\{q\big\}$ as shorthand for $\lambda; \mathcal{A}; \Omega \vDash \forall\!\!\forall \mathbf{x} \in \mathbf{1}.\, \langle p | \mathsf{true}\rangle\ \mathbb{C}\ \exists\!\!\exists \mathbf{y} \in \mathbf{1}.\, \langle q | \mathsf{true}\rangle$.

The semantic judgement breaks down into four mutually-exclusive cases: two progressing and two terminating. The first case covers normal progress, where the thread performs some atomic action (possibly id). The action may or may not perform the linearisation point: the two new private views express the outcome of each case. In the case where the linearisation point is not performed, the continuation takes up this obligation. In the case where the linearisation point is performed, the continuation loses responsibility for the public part.

The second case covers forking a new thread. This is just like the first case, taking the action id, but with an additional obligation on the semantics of the new thread: we must split the private part to give a precondition for both the continuation and the newly-forked thread. Since it is not possible to explicitly join on forked threads, we take their postcondition to be simply true. Note that the forked thread does not participate in the atomic action of the original thread.

The third case covers ordinary termination. In this case, the atomic action must be performed by the id action (since the thread is not going to perform any further actions).

The fourth case covers termination by return. This is similar to the previous case, except that the return postcondition, Ω, is used.

5.3 Soundness

We give some of the interesting proof steps in the soundness proof.

Lemma 1. *If, for $p \in \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $q, \omega \in \coprod_{x \in X} Q(x) \to \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $x \in X$, $y \in Q(x)$*

$$\lambda; a : x \in X \rightsquigarrow Q(x), \mathcal{A}; \exists x, y.\, \omega(x,y) * a \Mapsto (x,y) \vDash \begin{array}{c} \{p * a \Mapsto (x,y)\} \\ \mathbb{C} \\ \{\exists x, y.\, q(x,y) * a \Mapsto (x,y)\} \end{array}$$

then

$$\lambda; \mathcal{A}; \omega(x,y) \vDash \{p\}\ \mathbb{C}\ \{q(x,y)\}$$

Lemma 2 (Make Atomic Rule)**.** *Suppose that*

$$\{(x,y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_a(\mathrm{G})^*$$

$$\lambda; \mathcal{A}; \Omega \vDash \begin{array}{c} \left\{p_p * \exists x \in X.\, \mathbf{t}_a^{\lambda'}(x) * a \Mapsto \blacklozenge\right\} \\ \mathbb{C} \\ \{\exists x \in X, y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y)\} \end{array}$$

where

$$\mathcal{A} = a^{\lambda'} : x \in X \rightsquigarrow Q(x), \mathcal{A}'$$
$$\Omega(ret) = \exists x \in X, y \in Q(x).\, \omega(x,y,ret) * a \Mapsto (x,y)$$

and $a \notin \mathcal{A}'$. Then

$$\lambda; \mathcal{A}'; \omega \vDash \forall x \in X.\, \left\langle p_p \middle| \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a \right\rangle \mathbb{C}\ \exists y \in Q(x).\, \left\langle q_p(x,y) \middle| \mathbf{t}_a^{\lambda'}(y) * [\mathrm{G}]_a \right\rangle$$

Proof. Consider the case where $\mathbb{C}$ performs an action. Suppose that $\langle \mathbb{C}, s\rangle \xrightarrow{\alpha} \langle \mathbb{C}', s'\rangle$ where $\alpha \in \mathsf{AAction}$. By the premiss, there must be some $\overline{p'_p}$ with

$$\lambda; \mathcal{A} \vDash \left\langle p_p(s) * \exists x \in X.\, \mathbf{t}_a^{\lambda'}(x) * a \Mapsto \blacklozenge \right\rangle \alpha \left\langle \overline{p'_p}(s') \right\rangle \tag{1}$$

$$\lambda; \mathcal{A}; \Omega \vDash \left\{\overline{p'_p}\right\} \mathbb{C}' \left\{\exists x \in X, y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y)\right\}. \tag{2}$$

Fix $x \in X$. Fix $r \in \mathsf{View}_{\mathcal{A}'}$. Fix $\varphi \in p_p(s) * \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a * r$.

Let $p'_p = \lambda s.\, \left\{\varphi \in \mathsf{AWorld}_{\mathrm{dom}\,\mathcal{A}'} \;\middle|\; \varphi \bullet a \Mapsto \blacklozenge \in \overline{p'_p}(s)\right\}$.

Let $p''_p(x,y) = \lambda s.\, \left\{\varphi \in \mathsf{AWorld}_{\mathrm{dom}\,\mathcal{A}'} \;\middle|\; \varphi \bullet a \Mapsto (x,y) \in \overline{p'_p}(s)\right\}$.

Let $\overline{r} = r * [\mathrm{G}]_a * a \Mapsto -$. ($\overline{r}$ is stable with respect to $\mathcal{A}$ since the additional interference will be $a : x \in X \rightsquigarrow Q(x)$, and the subset of $\overline{r}$ that is compatible with $[\mathrm{G}]_a$ must be closed under this.) Let $\overline{\varphi} = \varphi \bullet a \Mapsto \blacklozenge$. By construction, $\lfloor \varphi \rfloor_\lambda = \lfloor \overline{\varphi} \rfloor_\lambda$. We have that $\overline{\varphi} \in (p_p(s) * \exists x \in X.\, \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge) * \overline{r}$.

By (1) there exists $\overline{\varphi'}$ with a) $\overline{\varphi}\ \mathrm{G}_{\lambda;\mathcal{A}}\ \overline{\varphi'}$, b) $h' \in \lfloor \overline{w'} \rfloor_\lambda$, and c) $\overline{\varphi'} \in \overline{p'_p}(s') * \overline{r}$.

From a) we can be sure that $d_{\overline{\varphi'}} \neq \lozenge$. Indeed, since $d_{\overline{\varphi}} = \blacklozenge$ and $\rho_{\overline{\varphi}} = x$, it must be that either $d_{\overline{\varphi'}} = \blacklozenge$ or $d_{\overline{\varphi}} = (x, y)$ for some $y \in Q(x)$.

Let φ' be such that $\varphi \in \varphi' * a \Mapsto -$. Now

$$\varphi' \in p'_p(s') * \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a \vee \exists y \in Q(x).\, p''_p(x, y, s') * \mathbf{t}_a^{\lambda'}(y)$$

since $\overline{\varphi'} \in \overline{p'_p}(s') * \overline{r}$ (by c). By a) and definitions, we get $\varphi\ \mathrm{G}_{\lambda;\mathcal{A}}\ \varphi'$. By construction $\lfloor \varphi' \rfloor_\lambda = \lfloor \overline{\varphi'} \rfloor_\lambda$ so $h' \in \lfloor \varphi' \rfloor_\lambda$ by b). Hence, we have established

$$\lambda; \mathcal{A} \vDash \left\langle p_p(s) * \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a \right\rangle \alpha \left\langle p'_p(s') * \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a \vee \exists y \in Q(x).\, p''_p(x, y, s') * \mathbf{t}_a^{\lambda'}(y) \right\rangle .$$

We have that $p'_p * \exists x \in X.\, {}^{x}_{t}(a)\lambda' * a \Mapsto \blacklozenge \vDash \overline{p'_p}$ and is stable with respect to $\mathcal{A}$. From (2), by left consequence and the coinductive hypothesis, we have

$$\lambda; \mathcal{A}; \Omega \vDash \forall\!\!\!\forall x \in X.\, \langle p'_p | \mathbf{t}_a^{\lambda'}(x) * [\mathrm{G}]_a \rangle\ \mathbb{C}'\ \ \exists\!\!\!\exists y \in Y.\, \langle q_p(x, y) | \mathbf{t}_a^{\lambda'}(y) * [\mathrm{G}]_a \rangle$$

Finally, from (2) and Lemma 1, we have, for all $y \in Q(x)$

$$\lambda; \mathcal{A}'; \omega \vDash \left\{ p''_p(x, y) \right\}\ \mathbb{C}'\ \left\{ q_p(x, y) \right\} .$$

The remaining cases are simpler, or follow similar reasoning. □

Lemma 3 (Update Region Rule)**.** *Suppose that* $a \notin \mathcal{A}$ *and*

$$\lambda; \mathcal{A}; \Omega \vDash \begin{array}{c} \forall\!\!\!\forall x \in X.\, \left\langle p_p \middle| I(\mathbf{t}_a^{\lambda}(x)) * p(x) \right\rangle \\ \mathbb{C} \\ \exists\!\!\!\exists y \in Q(x), z \in Z.\, \left\langle q_p(x, y, z) \middle| \begin{array}{c} I(\mathbf{t}_a^{\lambda}(y)) * q_1(x, y, z) \vee \\ I(\mathbf{t}_a^{\lambda}(x)) * q_2(x, y, z) \end{array} \right\rangle \end{array} .$$

Then

$$\lambda + 1; \mathcal{A}'; \Omega \vDash \begin{array}{c} \forall\!\!\!\forall x \in X.\, \left\langle p_p \middle| \mathbf{t}_a^{\lambda}(x) * p(x) * a \Mapsto \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists\!\!\!\exists y \in Q(x), z \in Z.\, \left\langle q_p(x, y, z) \middle| \begin{array}{c} \mathbf{t}_a^{\lambda}(y) * q_1(x, y, z) * a \Mapsto (x, y) \vee \\ \mathbf{t}_a^{\lambda}(x) * q_2(x, y, z) * a \Mapsto \blacklozenge \end{array} \right\rangle \end{array} ,$$

where $\mathcal{A}' = (a : x \in X \rightsquigarrow Q(x), \mathcal{A})$.

Proof. Suppose that $\langle \mathbb{C}, s \rangle \xrightarrow{\alpha} \langle \mathbb{C}', s' \rangle$ with $\alpha \in \mathsf{AAction}$.

Fix $x \in X$. From our assumption, there are p'_p and $\overline{p''_p}$ with

$$\lambda; \mathcal{A} \vDash \begin{array}{c} \left\langle p_p(s) * I(\mathbf{t}_a^{\lambda}(x)) * p(x) \right\rangle \\ \alpha \\ \left\langle p'_p(s') * I(\mathbf{t}_a^{\lambda}(x)) * p(x) \vee \begin{array}{c} \exists y \in Q(x), z \in Z.\, \overline{p''_p}(x, y, z, s') * \\ \left(\begin{array}{c} I(\mathbf{t}_a^{\lambda}(y)) * q_1(x, y, z) \vee \\ I(\mathbf{t}_a^{\lambda}(x)) * q_2(x, y, z) \end{array} \right) \end{array} \right\rangle \end{array} \tag{3}$$

$$\lambda; \mathcal{A}; \Omega \vDash \begin{array}{c} \left\langle p'_p \middle| I(\mathbf{t}_a^{\lambda}(x)) * p(x) \right\rangle \\ \mathbb{C}' \\ \exists\!\!\!\exists y \in Q(x), z \in Z.\, \left\langle q_p(x, y, z) \middle| \begin{array}{c} I(\mathbf{t}_a^{\lambda}(y)) * q_1(x, y, z) \vee \\ I(\mathbf{t}_a^{\lambda}(x)) * q_2(x, y, z) \end{array} \right\rangle \end{array} \tag{4}$$

$$\forall y \in Q(x), z \in Z. \quad \lambda; \mathcal{A}; \Omega(x, y, z) \vDash \left\{ \overline{p''_p}(x, y, z) \right\}\ \mathbb{C}'\ \left\{ q_p(x, y, z) \right\} \tag{5}$$

We will show that these p'_p and $p''_p(x,y,z) = \overline{p''_p}(x,y,z) * a \Mapsto (x,y)$ work to establish our goal.

Fix $r \in \mathsf{View}_{\mathcal{A}'}$, $\varphi \in p_p(s) * \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge * r$, $h \in \lfloor \varphi \rfloor_{\lambda+1}$, $h' \in [\![\alpha]\!](h)$. Let $\overline{r} \in \mathsf{View}_{\mathcal{A}}$ be such that

$$\overline{r} = \mathrm{removedone}_a \left(r * \bigoplus_{\substack{a' \in \mathsf{RId} \\ a' \neq a \\ r_\varphi(a') = (\lambda, -, -)}} I(r_\varphi(a'), a', \rho_\varphi(a')) \right).$$

That is, we open all regions at level λ (except a) with their states as given by φ and remove the atomicity tracking for a.

There will be some $\overline{\varphi} \in p_p * I(\mathbf{t}_a^\lambda(x)) * p(x) * \overline{r}$ with $r_\varphi = r_{\overline{\varphi}}$ and $\rho_\varphi = \rho_{\overline{\varphi}}$, and $\lfloor \overline{\varphi} \rfloor_\lambda = \lfloor \varphi \rfloor_{\lambda+1}$, and so $h \in \lfloor \overline{\varphi} \rfloor_\lambda$. By (3), there is some $\overline{\varphi'}$ with $\overline{\varphi} \; \mathrm{G}_{\lambda;\mathcal{A}} \; \overline{\varphi'}$, $h' \in \lfloor \varphi' \rfloor_\lambda$ and

$$\overline{\varphi'} \in \left(p'_p(s') * I(\mathbf{t}_a^\lambda(x)) * p(x) \vee \begin{array}{c} \exists y \in Q(x), z \in Z.\, \overline{p''_p}(x,y,z,s') * \\ \left(\begin{array}{c} I(\mathbf{t}_a^\lambda(y)) * q_1(x,y,z) \vee \\ I(\mathbf{t}_a^\lambda(x)) * q_2(x,y,z) \end{array} \right) \end{array} \right) * \overline{r}$$

We have the following cases for $\overline{\varphi'}$:

- $\overline{\varphi'} \in p'_p(s'') * I(\mathbf{t}_a^\lambda(x)) * p(x) * \overline{r}$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

  $$\overline{\overline{\varphi'}} \in I(\mathbf{t}_a^\lambda(x)) * \bigoplus_{\substack{a' \in \mathsf{RId} \\ a' \neq a \\ r_\varphi(a') = (\lambda, -, -)}} I(r_\varphi(a'), a', \rho_\varphi(a'))$$

  and $\varphi'' \in p'_p(s') * p(x) * r$. Let

  $$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}, d_{\varphi''}[a \mapsto \blacklozenge]).$$

  Hence, by the guarantee, $\varphi' \in p'_p(s'') * \mathbf{t}_a^\lambda(x) * p(x) * r$, and by construction $\lfloor \varphi' \rfloor_{\lambda+1} = \lfloor \varphi'' \rfloor_\lambda$. Also $\varphi \; \mathrm{G}_{\lambda+1;\mathcal{A}'} \; \varphi'$.

- $\overline{\varphi'} \in \overline{p''_p}(x,y,z,s') * I(\mathbf{t}_a^\lambda(y)) * q_1(x,y,z) * \overline{r}$ for some $y \in Q(x)$ and $z \in Z$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

  $$\overline{\overline{\varphi'}} \in I(\mathbf{t}_a^\lambda(y)) * \bigoplus_{\substack{a' \in \mathsf{RId} \\ a' \neq a \\ r_\varphi(a') = (\lambda, -, -)}} I(r_\varphi(a'), a', \rho_\varphi(a'))$$

  and $\varphi'' \in \overline{p''_p}(x,y,z,s') * q_1(x,y,z) * r$. Let

  $$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}[a \mapsto y], d_{\varphi''}[a \mapsto (x,y)]).$$

  Hence, by the guarantee, $\varphi' \in p''_p(x,y,z,s') * \mathbf{t}_a^\lambda(y) * q_1(x,y,z) * r$, and by construction $\lfloor \varphi' \rfloor_{\lambda+1} = \lfloor \varphi'' \rfloor_\lambda$. Also $\varphi \; \mathrm{G}_{\lambda+1;\mathcal{A}'} \; \varphi'$.

- $\overline{\varphi'} \in \overline{p''_p}(x, y, z, s') * I(\mathbf{t}_a^\lambda(x)) * q_2(x, y, z) * \overline{r}$ for some $y \in Q(x)$ and $z \in Z$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

$$\overline{\overline{\varphi'}} \in I(\mathbf{t}_a^\lambda(x)) * \mathop{\circledast}_{\substack{a' \in \mathsf{RId} \\ a' \neq a \\ r_\varphi(a') = (\lambda, -, -)}} I(r_\varphi(a'), a', \rho_\varphi(a'))$$

and $\varphi'' \in \overline{p''_p}(x, y, z, s') * q_2(x, y, z) * r$. Let

$$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}, d_{\varphi''}[a \mapsto \blacklozenge]).$$

Hence, by the guarantee, $\varphi' \in p''_p(x, y, z, s') * \mathbf{t}_a^\lambda(x) * q_2(x, y, z) * r$, and by construction $\lfloor \varphi' \rfloor_{\lambda+1} = \lfloor \varphi'' \rfloor_\lambda$. Also $\varphi \; \mathrm{G}_{\lambda+1;\mathcal{A}'} \; \varphi'$.

In each case we have φ' which satisfies $\varphi \; \mathrm{G}_{\lambda+1;\mathcal{A}} \; \varphi'$, $h' \in \lfloor \varphi' \rfloor_{\lambda+1}$ and

$$\varphi' \in p'_p(s') * \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \vee \begin{pmatrix} \exists y \in Q(x), z \in Z.\, p''_p(x, y, z, s') * \\ \mathbf{t}_a^\lambda(y) * q_1(x, y, z) * a \Mapsto (x, y) \vee \\ \mathbf{t}_a^\lambda(x) * q_2(x, y, z) * a \Mapsto \blacklozenge \end{pmatrix} * r.$$

So we have established that

$$\lambda{+}1; \mathcal{A}' \vDash \begin{array}{c} \left\langle p_p(s) * \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \right\rangle \\ \alpha \\ \left\langle p'_p(s') * \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \vee \begin{pmatrix} \exists y \in Q(x), z \in Z.\, p''_p(x, y, z, s') * \\ \mathbf{t}_a^\lambda(y) * q_1(x, y, z) * a \Mapsto (x, y) \vee \\ \mathbf{t}_a^\lambda(x) * q_2(x, y, z) * a \Mapsto \blacklozenge \end{pmatrix} \right\rangle \end{array}.$$

By (4), it follows from the coinductive hypothesis that

$$\lambda + 1; \mathcal{A}'; \Omega \vDash \begin{array}{c} \mathbb{V}x \in X.\, \left\langle p'_p \middle| \mathbf{t}_a^\lambda(x) * p(x) * a \Mapsto \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists\!\!\exists y \in Q(x), z \in Z.\, \left\langle q_p(x, y, z) \middle| \begin{matrix} \mathbf{t}_a^\lambda(y) * q_1(x, y, z) * a \Mapsto (x, y) \vee \\ \mathbf{t}_a^\lambda(x) * q_2(x, y, z) * a \Mapsto \blacklozenge \end{matrix} \right\rangle \end{array}$$

Fix $y \in Q(x)$ and $z \in Z$. Because $p''_p(x, y, z) = \overline{p''_p}(x, y, z) * a \Mapsto (x, y)$, we can extend the atomicity context and have

$$\lambda + 1; \mathcal{A}'; \Omega(x, y, z) \vDash \left\{ p''_p(x, y, z) \right\} \; \mathbb{C}' \; \left\{ q_p(x, y, z) \right\}$$

□

6 Conclusions

We have introduced a program logic, TaDA, which includes a novel judgement for specifying abstract atomicity. We have shown how modules can be proved to implement such specifications, and how clients can make use of them, through a series of examples culminating in a concurrent double-ended queue. By combining abstract atomicity with abstract disjointness, we have seen how TaDA can achieve the best of both worlds. With the `readTo` operation, we saw that TaDA can even handle resource transfer between a client and an abstractly atomic module operation, which is not possible with traditional linearisability. We believe that TaDA's expressive new approach supports simple and intuitive proofs. Our intention with TaDA is to build a program logic that will scale to industrial concurrency libraries, such as `java.util.concurrent`. With its simplicity and expressivity, TaDA represents a step in this direction.

6.1 Future Work

Helping. In some concurrent modules, one thread's abstract atomic action may actually be effected by another thread — a phenomenon termed *helping.* As presented, TaDA does not support helping, since each abstract atomic operation of a thread can be traced down to a concrete atomic action of that thread at which it takes effect. By transforming the atomic tracking component into a transferrable resource, it should be possible to support helping. However, this will require a different semantic model.

Higher-order. iCAP [16] makes use of impredicative protocols for shared regions — protocols that can reference arbitary protocols. This gives it the expressive power to handle higher-order programs and reentrancy. It would be interesting to combine TaDA with iCAP, which may be possible by proving the rules of TaDA (or similar) sound in the metatheory of iCAP. Iterators on concurrent collections, which can have subtle specifications, could benefit from the expressive power of such a logic.

Weak Memory. Burkhardt *et al.* [1] have extended the concept of linearisability to the total store order (TSO) memory model. Sieczkowski *et al.* [15] have extended iCAP to work with the TSO memory model, also. TaDA already has some potential to specify weak behaviours. For instance, the following three specifications for a read operation are increasingly weak:

$$\begin{gathered}
\vdash \mathbb{V}v.\, \big\langle \mathtt{x} \mapsto v \big\rangle\ \mathtt{y} := [\mathtt{x}]\ \big\langle \mathtt{x} \mapsto v \land \mathtt{y} = v \big\rangle \\
\vdash \big\langle \mathtt{x} \mapsto v \big\rangle\ \mathtt{y} := [\mathtt{x}]\ \big\langle \mathtt{x} \mapsto v \land \mathtt{y} = v \big\rangle \\
\vdash \big\{ \mathtt{x} \mapsto v \big\}\ \mathtt{y} := [\mathtt{x}]\ \big\{ \mathtt{x} \mapsto v \land \mathtt{y} = v \big\}
\end{gathered}$$

The first of these specifications gives the usual atomic semantics; the second prohibits concurrent updates; the third prohibits any concurrent access. An interesting research direction would be to investigate extensions of TaDA that can specify and verify programs that make use of weak memory models such as TSO.

References

[1] Burckhardt, S., Gotsman, A., Musuvathi, M., and Yang, H. Concurrent library correctness on the tso memory model. In *ESOP* (2012), pp. 87–107.

[2] da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., and Wheelhouse, M. A simple abstraction for complex concurrent indexes. In *OOPSLA* (2011), pp. 845–864.

[3] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., and Yang, H. Views: compositional reasoning for concurrent programs. In *POPL* (2013), pp. 287–300.

[4] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. Concurrent abstract predicates. In *ECOOP* (2010), pp. 504–528.

[5] Dodds, M., Feng, X., Parkinson, M., and Vafeiadis, V. Deny-guarantee reasoning. In *ESOP* (2009), pp. 363–377.

[6] Doherty, S., Detlefs, D. L., Groves, L., Flood, C. H., Luchangco, V., Martin, P. A., Moir, M., Shavit, N., and Steele, Jr., G. L. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA* (2004), pp. 216–224.

[7] Dreyer, D., Neis, G., and Birkedal, L. The impact of higher-order state and control effects on local relational reasoning. In *ICFP* (2010), pp. 143–156.

[8] Filipović, I., O'Hearn, P., Rinetzky, N., and Yang, H. Abstraction for concurrent objects. In *ESOP* (2009), pp. 252–266.

[9] Gotsman, A., and Yang, H. Linearizability with ownership transfer. In *CONCUR* (2012), pp. 256–271.

[10] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[11] Jacobs, B., and Piessens, F. Expressive modular fine-grained concurrency specification. In *POPL* (2011), pp. 271–282.

[12] Ley-Wild, R., and Nanevski, A. Subjective auxiliary state for coarse-grained concurrency. In *POPL* (2013), pp. 561–574.

[13] O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci. 375*, 1-3 (Apr. 2007), 271–307.

[14] Parkinson, M., and Bierman, G. Separation logic and abstraction. In *POPL* (2005), pp. 247–258.

[15] Sieczkowski, F., Svendsen, K., and Birkedal, L. A separation logic for fictional sequential consistency. Submitted for publication, 2013.

[16] Svendsen, K., and Birkedal, L. Impredicative concurrent abstract predicates. Submitted for publication, 2013.

[17] Svendsen, K., Birkedal, L., and Parkinson, M. Modular reasoning about separation of concurrent data structures. In *ESOP* (2013), pp. 169–188.

[18] Turon, A., Dreyer, D., and Birkedal, L. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP* (2013), pp. 377–390.

[19] Turon, A. J., Thamsborg, J., Ahmed, A., Birkedal, L., and Dreyer, D. Logical relations for fine-grained concurrency. In *POPL* (2013), pp. 343–356.

[20] Vafeiadis, V. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, Computer Laboratory, 2008.

[21] Vafeiadis, V., Herlihy, M., Hoare, T., and Shapiro, M. Proving correctness of highly-concurrent linearisable objects. In *PPoPP* (2006), pp. 129–136.