TaDA: A Logic for Time and Data Abstraction

Pedro da Rocha Pinto[1], Thomas Dinsdale-Young[2], and Philippa Gardner[1]

[1] Imperial College London
{pmd09,pg}@doc.ic.ac.uk
[2] Aarhus University
tyoung@cs.au.dk

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.

We present TaDA, a program logic that combines the benefits of abstract atomicity and abstract disjointness. Our key contribution is the introduction of *atomic triples*, which offer an expressive approach to specifying program modules. By building up examples, we show that TaDA supports elegant modular reasoning in a way that was not previously possible.

1 Introduction

The specification and verification of concurrent program modules is a difficult problem. When concurrent threads work with shared data, the resulting behaviour can be complex. Two abstractions provide useful simplifications: that operations effectively act at distinct times; and that operations effectively act on disjoint resources. Programmers work with sophisticated combinations of the time and data abstractions. In constrast, existing reasoning techniques tend to be limited to one or the other abstraction.

Atomicity is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of atomic operations is equivalent to some sequential interleaving of the operations. *Linearisability* [9] is a correctness condition, which specifies that the operations of a concurrent module appear to behave atomically. For example, a set module might use a sophisticated lock-free data structure to implement `insert`, `remove` and `contains` operations. Linearisability allows a client to use these as if they were simple atomic operations, abstracting the implementation details. Various proof techniques have been introduced and used to prove linearisability for concurrent modules such as queues [9] and lists with fine-grained synchronisation [21].

With linearisability, each operation is given a sequential specification, and the operations are asserted to behave atomically *with respect to each other*. Linearisability is therefore a whole-module property: if we extend the set module with an atomic `insertBoth` operation, we would have to redo the linearisability proof to check that this new operation respects the atomicity of the others, and vice versa. Moreover, all operations are required to be atomic, so we could not specify a non-atomic `insertBoth` behaving like two consecutive atomic inserts. It is also possible to add operations to a module that break the abstraction of atomicity for existing operations. For example, if the set module were to expose the low-level heap operations used in its implementation, a client could use them to observe intermediate states in the underlying data structure. Consequently, the fiction of atomicity is fragile.

The sequential specifications used for linearisability can be inadequate for expressing concurrent behaviours. In particular, we might wish to constrain which operations a client can perform concurrently. For instance, a module might provide alternative update operations that only appear atomic if all other concurrent operations are reads. Constraining the client in this way reduces the burden on the implementation, which can be more efficient. However, a sequential specification cannot express the distinction between the alternative and regular updates.

Disjointness is the abstraction that operations act on specific resources. When threads operate on disjoint resources, they do not interfere with each other, and so their overall effect is the combined effects of each. *Concurrent separation logics* [12,3,17,16] embody this principle, by providing modular reasoning about disjoint resource. *Concurrent abstract predicates* (CAP) [3], in particular, support reasoning about *abstract* disjoint resource, which can be used to specify program modules. In the case of a set module, for instance, values may be seen as resources, which may be independently in or out of the set. If concurrent threads use disjoint values, reasoning about them is simple. CAP also supports reasoning about *shared regions*, which can be used to implement abstract disjoint resources with shared resources. In this way, sophisticated concurrent implementations can be verified against simple specifications. Such reasoning has been applied to, for example, locks [3], sets [3] and concurrent indexes [14].

The CAP approach is, however, limited. With CAP, it is only possible to access shared regions using *primitive atomic* operations. Yet operations provided by concurrent modules are rarely primitive atomic. Consequently, the abstract resources provided by a module are not easily shared and the nesting of modules is difficult. For example, the CAP specification of a set module [3] constrains concurrent threads to operate on disjoint values. Two threads cannot remove the same value: since `remove` is not primitive atomic, it cannot operate on shared resources. It is possible to give a specification that has a finer resource granularity [14], which can support some form of shared concurrent removal. Such specifications are complex and ad hoc, as they do not support general sharing.

Linearisability and CAP have complementary virtues and weaknesses. Linearisability gives strong, whole-module specifications based on abstract atomicity; CAP gives weaker, independent specifications based on abstract disjointness.

Linearisability supports nested modules, but whole-module specifications make it difficult to extend modules; CAP supports the extension of modules, but the weak specifications make building up nested modules more difficult. Linearisability does not constrain the client, thus placing significant burden on the implementation; CAP constrains the client to use specific disjoint resource, enabling more flexibility in the implementation.

We propose a solution that combines the virtues of both approaches. Specifically, we introduce a new *atomic triple* judgement for specifying abstract atomicity in a program logic. The simplest form of atomic triple judgement is

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

where p and q are assertions in the style of separation logic 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 atomic 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 happened, the environment can do what it likes; it is not constrained to preserve q. Meanwhile, the program $\mathbb{C}$ may no longer have access to the resources in q.

The atomicity of $\mathbb{C}$ is *only* expressed 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.

Atomic triples are our key contribution, as they allow us to overcome limitations of the linearisability and CAP approaches. Atomic triples can be used to access shared resources concurrently, rather than relying on primitive atomic operations to do so. This makes it easier to build modules on top of each other. Atomic triples specify operations with respect to an abstraction, so they can be proved independently. This makes it possible to extend modules at a later date, and mix atomic and non-atomic operations as well as operations working at different levels of abstraction. Atomic triples can specify clear constraints on how a client can use them. For instance, they can enforce that the unlock operation on a lock should not be called by two threads at the same time (§2.1). Furthermore, atomic triples can specify the transfer of resources between a client and a module. For instance, they can specify an operation that non-atomically stores the result of an atomic read into a buffer provided by a client (§2.3).

Our other main contribution is TaDA, a program logic for Time and Data Abstraction, which extends CAP with rules for deriving and using atomic triples. Using TaDA, we first specify an atomic lock module (§2.1). From this specification, we then derive a resource-transferring CAP-style lock specification, which illustrates the weakening of the atomic specification to a specific use case. We

also prove that a spin lock implementation satisfies the atomic lock specification. We show how the logic supports vertical reasoning about modules, by verifying an implementation of multiple-compare-and-swap (MCAS) using the lock specification (§2.2), and an implementation of a concurrent double-ended queue (deque) using the MCAS specification (§4). We present the details of TaDA's proof rules in §3, and briefly describe their semantics and soundness in §5. We thus demonstrate that TaDA combines the benefits of abstract atomicity and abstract disjointness within a single program logic.

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()`.

Atomic Lock Specification. The lock operations are specified in terms of abstract predicates [13] 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 [3], but we shall see how the CAP specification can be derived using the atomic specification given here.

The specification for the `makeLock()` operation is a simple Hoare triple:

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

The operation 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:

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

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 Hoare triples, since they make no guarantee that intermediate steps preserve invariants on the resources. 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 $\mathtt{unlock}(\mathtt{x})$ 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 $\mathtt{lock}(\mathtt{x})$ 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:

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

This specification has two significant flaws. Firstly, it allows $\mathtt{lock}(\mathtt{x})$ 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:

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

In the left-hand triple, the lock is initially locked; the implementation may not terminate, nor change the state of the lock. In the right-hand triple, 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 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.

An equivalent specification makes use of a boolean logical variable:

$$\forall l \in \mathbb{B}. \vdash \left\langle (\mathsf{L}(\mathtt{x}) \wedge \neg l) \vee (\mathsf{U}(\mathtt{x}) \wedge l) \right\rangle \ \mathtt{lock}(\mathtt{x}) \ \left\langle \mathsf{L}(\mathtt{x}) \wedge l \right\rangle$$

The variable l records the state of the lock when the atomic operation takes effect. In particular, it cannot take effect unless the lock is already unlocked.

These specifications do not express the subtlety that the interference permitted before the atomic update is different for the environment and the operation. The environment should be allowed to change the value of l (i.e. acquire and release the lock) but the $\mathtt{lock}$ operation should not. The correct specification expresses this by binding the variable l in a new way:

$$\vdash \mathbb{W} l \in \mathbb{B}. \left\langle (\mathsf{L}(\mathtt{x}) \wedge \neg l) \vee (\mathsf{U}(\mathtt{x}) \wedge l) \right\rangle \ \mathtt{lock}(\mathtt{x}) \ \left\langle \mathsf{L}(\mathtt{x}) \wedge l \right\rangle$$

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).

CAP Lock Specification. The atomic specification of the lock captures its essence as a synchronisation primitive. In practice, a lock is often used to protect some resource. We demonstrate how a CAP-style lock specification [3], which views the lock as a mechanism for protecting a resource invariant, can be derived from the atomic specification. This illustrates a typical use of a TaDA specification: first prove a strong abstract-atomic specification, then specialise to whatever is required by the client.

The CAP specification is parametrised by an abstract predicate Inv, representing the resource invariant to be protected by the lock. The client can choose how to instantiate this predicate.[3] The specification provides two abstract predicates itself: $\mathsf{isLock}(x)$, 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):

$$\vdash \left\{\mathsf{Locked}(\mathtt{x}) * \mathsf{Inv}\right\}\ \mathtt{unlock(x)}\ \left\{\mathsf{emp}\right\}$$

$$\vdash \left\{\mathsf{isLock}(\mathtt{x})\right\}\ \mathtt{lock(x)}\ \left\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x}) * \mathsf{Inv}\right\}$$

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

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

To implement this specification, we must provide an interpretation for the abstract predicates isLock and Locked. 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 $\mathsf{L}(x)$, $\mathsf{U}(x)$ and Inv, plus some additional guard resource (described below). A shared region is associated with a protocol, which determines how its contents change over time. Following iCAP [16], 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.

[3] The restriction is that the predicate must be *stable* — i.e. invariant under interference from the environment.

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, we need only a very simple guard separation algebra: there is a single, indivisible guard named K (for 'key'), as well as the empty guard $\mathbf{0}$. 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 $x \in \{\mathbf{0}, \mathrm{K}\}$, and $\mathrm{K} \bullet \mathrm{K}$ is 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:

$$\mathbf{0} \ : \ 0 \rightsquigarrow 1 \qquad\qquad \mathrm{K} \ : \ 1 \rightsquigarrow 0$$

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:

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

With this interpretation, the guard K and invariant Inv are in the region when it is in the unlocked state. This means that, when a thread acquires the lock, it takes ownership of the guard and the lock invariant by removing them from the region. Having the guard K allows the thread to subsequently release the lock, returning the guard and invariant to the region.

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:

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

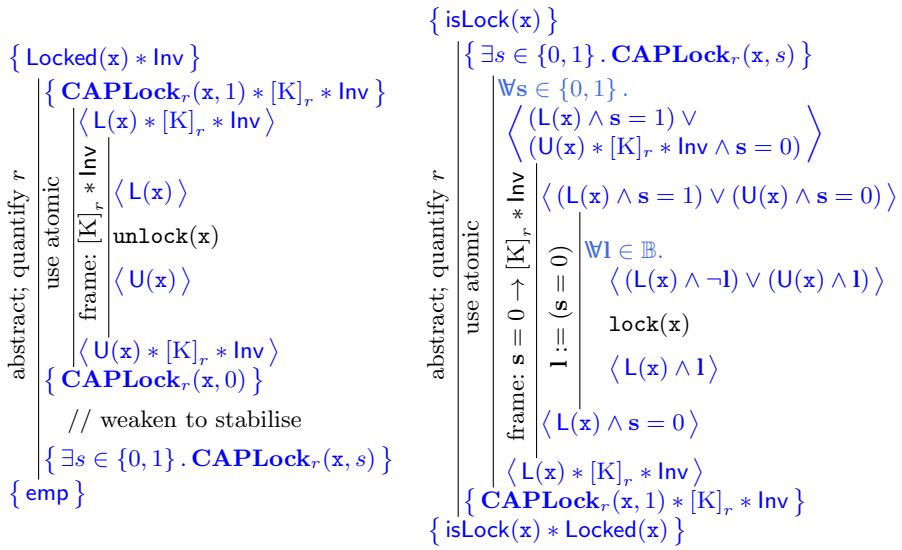

Fig. 1. Derivation of CAP lock specifications.

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 $\forall$ quantifier in the premiss of the "use atomic" to account for the fact that, in the precondition, the lock could be in either state. The proof uses the frame rule, with a frame that is conditional on the state of the lock. It also uses the *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.

Note that neither of the bad specifications for `lock(x)` could be used in this derivation: the first because there would be no way to express that the frame $[\mathrm{K}]_r * \mathsf{Inv}$ is conditional on the state of the lock; and the second because we could not combine both cases in a single derivation.

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

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

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

Fig. 2. Lock operations.

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 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:

$$\mathrm{G} \ : \ 0 \rightsquigarrow 1 \qquad\qquad \mathrm{G} \ : \ 1 \rightsquigarrow 0$$

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

$$I(\mathbf{Lock}_a(x,1)) \ \triangleq \ x \mapsto 1 \qquad\qquad I(\mathbf{Lock}_a(x,0)) \ \triangleq \ x \mapsto 0$$

We now define the interpretation of the predicates 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.

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:

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

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.

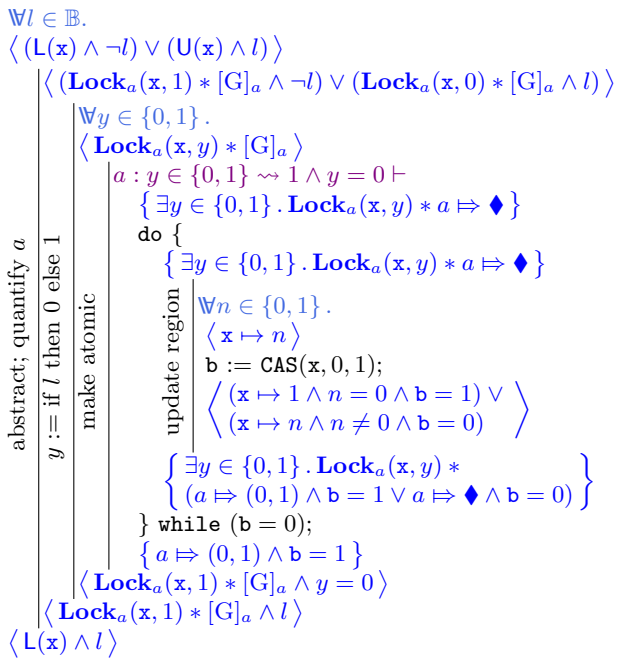

Fig. 3. Proof of the `lock(x)` operation.

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 (the $a \Mapsto \blacklozenge$ indicates it has not) and, if so, the state of the shared region immediately before and after (the $a \Mapsto (x, y)$). 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 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.

The proof of the `lock(x)` implementation is given in Fig. 3. 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 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 proofs for `makeLock` and `unlock` are simpler, and may be found in the technical report [15].

Remark 1. 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 $\vdash \langle \mathsf{L}(\mathtt{x}) \vee \mathsf{U}(\mathtt{x}) \rangle\ \mathtt{unlock(x)}\ \langle \mathsf{U}(\mathtt{x}) \rangle$, for example. Since the specification constrains the client, it allows flexibility in the implementation.

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 with 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 the others 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. 4.

Implementation. We give a straightforward coarse-grained implementation of the MCAS specification. The operation `makeMCL` creates a lock which protects updates to pointers under the control of the library. 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.

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

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

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

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

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

$$\vdash \mathbb{V}v.\ \left\langle \mathsf{MCP}(\mathtt{l},\mathtt{x},v)\right\rangle\ \mathtt{b} := \mathtt{cas}(\mathtt{l},\mathtt{x},\mathtt{v1},\mathtt{v2})\ \left\langle \begin{array}{c}\underline{\mathbf{if}}\ v = \mathtt{v1}\ \underline{\mathbf{then}}\ \mathtt{b} = 1 \wedge \mathsf{MCP}(\mathtt{l},\mathtt{x},\mathtt{v2})\\ \underline{\mathbf{else}}\ \mathtt{b} = 0 \wedge \mathsf{MCP}(\mathtt{l},\mathtt{x},v)\end{array}\right\rangle$$

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

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

$$\mathsf{MCL}(l) \iff \mathsf{MCL}(l) * \mathsf{MCL}(l)$$

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

Fig. 4. The abstract specification for the MCAS module.

This guard is used by all operations of the library that access the heap cell x. The following implication 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 Own guards for heap cells that are not currently under the protection of the module into $\textsc{Owned}(X)$, where X is the set of all cells that *are* protected. We have the following equivalence:

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

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 Owned guard:

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

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 $\textsc{Own}(x)$, $x \mapsto v$ 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:

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

Internally, the region may be in one of two states, indicated by the disjunction. Either the lock l is unlocked, and the heap cells corresponding to the abstract state of the region are actually in the region, as well as the $\textsc{Unlocked}$ guard. Or the lock l is locked, and some portion h_1 of the abstract heap is in the 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 $\textsc{Owned}(\mathrm{dom}(h))$ guard belongs to the region, encapsulating the $\textsc{Own}$ guards for heap addresses that are not protected.

We now give an interpretation to the predicates as follows:

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

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, which it owns, and possibly other heap cells in the region.

We can now prove that the specification is satisfied by the implementation. For brevity, we only show the `dcas` command in Fig. 5. The other commands have similar proofs.

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

Fig. 5. 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:

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

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 partial 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:

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

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 can 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:

$$\vdash \mathbb{V}v, w.\, \langle \mathtt{y} \mapsto w \mid \mathsf{MCP}(\mathtt{l},\mathtt{x},v) \rangle\ \mathtt{readTo}(\mathtt{l},\mathtt{x},\mathtt{y})\ \langle \mathtt{y} \mapsto v \mid \mathsf{MCP}(\mathtt{l},\mathtt{x},v) \rangle$$

One way of understanding such specifications is in terms of ownership transfer between a client and a module, as in [8]: 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, $\vdash \{p\}\ \mathbb{C}\ \{q\}$ is equivalent to $\vdash \langle p \mid \mathsf{true} \rangle\ \mathbb{C}\ \langle q \mid \mathsf{true} \rangle$.

3 Logic

We give an overview of the key TaDA proof rules that deal with atomicity in Fig. 6. Here, we do not formally define the syntax and semantics of our assertions, although we describe how they are modelled in §5. These details are given in the technical report [15].

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.

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

Frame rule

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

Substitution rule

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

Atomicity weakening rule

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

Open region rule

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

Use atomic rule

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

Update region rule

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

Make atomic rule

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

Fig. 6. Selected proof rules of TaDA.

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. 5, 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 perform 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. 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 $\exists\!\!|$-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. A deque has operations that allow elements to be inserted and removed from both ends of a list.

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.

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:

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

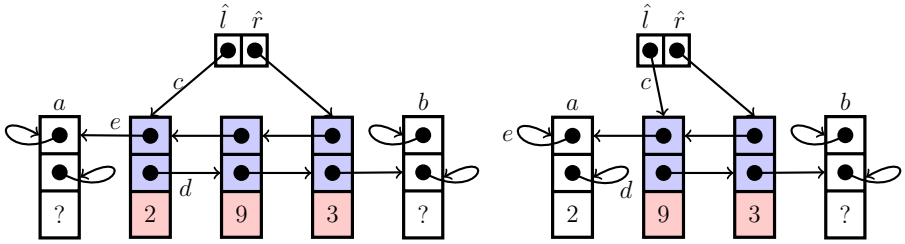

Fig. 7. Examples of a deque before and after performing `popLeft`, which uses `3cas` to updated pointers c, d and e.

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

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

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

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* [5]. An example of the shape of the data structure is shown in Fig. 7. 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* ($\hat{l}$ 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 focus on the `popLeft` implementation. This implementation first reads the left hat value to a local variable. It then reads the left-link of the node referenced by that variable. If both values are the same, it means that the node is dead and 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. 7. 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 set of pairs of nodes addresses and values that were part of the deque, but are now dead. We maintain the set of dead nodes to guarantee that after a node is removed from the deque, its value can still be read. In order to change the abstract state of the deque, we require the guard G. The labelled transition system is as follows:

$$\begin{aligned}
&\mathrm{G} : \forall n, v, ns, ds.\, (ns, ds) \rightsquigarrow ((n, v) : ns, ds)\\
&\mathrm{G} : \forall n, v, ns, ds.\, (ns, ds) \rightsquigarrow (ns : (n, v), ds)\\
&\mathrm{G} : \forall n, v, ns, ds.\, ((n, v) : ns, ds) \rightsquigarrow (ns, ds \uplus \{(n, v)\})\\
&\mathrm{G} : \forall n, v, ns, ds.\, (ns : (n, v), ds) \rightsquigarrow (ns, ds \uplus \{(n, v)\})
\end{aligned}$$

In order to provide an interpretation for the abstract state, we first define a number of auxiliary predicates. We use field notation: $E.\mathtt{field}$ is 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 cells:

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

Here l and r are the left- and right-link addresses. The L parameter is the address of the MCAS lock. 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).

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

We define a predicate to include the dead nodes (ds) as well as the doubly-linked list:

$$\begin{aligned}
&\mathsf{dls}(L, l, r, ns, ds) \equiv\\
&\qquad \exists a, b.\, (a, -), (b, -) \in ds \wedge \mathsf{dlseg}(L, l, r, a, b, ns) * \circledast_{(n,v)\in ds} \mathsf{dead}(L, n, v)
\end{aligned}$$

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.

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

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:

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

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. 8. The remaining proofs are given in the technical report [15].

5 Semantics

We briefly describe the model for TaDA and the intuition behind the soundness proof. Details can be found in the technical report [15].

Assertions are modelled as sets of *worlds*. A world includes (partial) information about the concrete heap state, as well as the instrumentation used by the proof system. This instrumentation consists of the type and state of each shared region, abstract predicate resources, and guard resources for each region. Depending on the atomicity context, it may also include atomicity tracking resources. Composition between worlds (which is lifted to sets to interpret $*$ in assertions) requires that they agree on the type and state of all regions, and that their resources (including heap resources) must be disjoint. Worlds are subject to interference, which is represented by a relation. This interference relation expresses the conditions under which the environment may modify the shared regions, which is dependent on guards and atomicity tracking resources. Assertions must be stable — closed under the interference relation — and are consequently *views* in the sense of the Views Framework [2], which we use as the basis for our soundness proof.

The judgements of TaDA are interpreted with a semantic judgement:

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

The meaning of this judgement is expressed in terms of the steps that $\mathbb{C}$ may take in the operational semantics. Each step may either leave $p(\mathbf{x})$ intact, or update it to $q(\mathbf{x}, \mathbf{y})$ for some value of $\mathbf{y}$. Simultaneously, it may update its private state p_p arbitrarily, so long as any changes to shared regions are permitted by guards that it owns, or atomic tracking resources. Once the update from $p(\mathbf{x})$ to $q(\mathbf{x}, \mathbf{y})$ occurs, the thread gives up access to $q(\mathbf{x}, \mathbf{y})$. From then, it can only update the private state, and must ensure that $q_p(\mathbf{x}, \mathbf{y})$ holds when it terminates.

The key result for establishing soundness is the following:

Theorem 1. *If* $\lambda; \mathcal{A} \vdash \mathbb{\forall}\mathbf{x} \in X.\, \langle p_p | p(\mathbf{x}) \rangle \; \mathbb{C} \; \mathbb{\exists}\mathbf{y} \in Y.\, \langle q_p(\mathbf{x}, \mathbf{y}) | q(\mathbf{x}, \mathbf{y}) \rangle$ *is provable in the logic, then* $\lambda; \mathcal{A} \vDash \mathbb{\forall}\mathbf{x} \in X.\, \langle p_p | p(\mathbf{x}) \rangle \; \mathbb{C} \; \mathbb{\exists}\mathbf{y} \in Y.\, \langle q_p(\mathbf{x}, \mathbf{y}) | q(\mathbf{x}, \mathbf{y}) \rangle$ *holds semantically.*

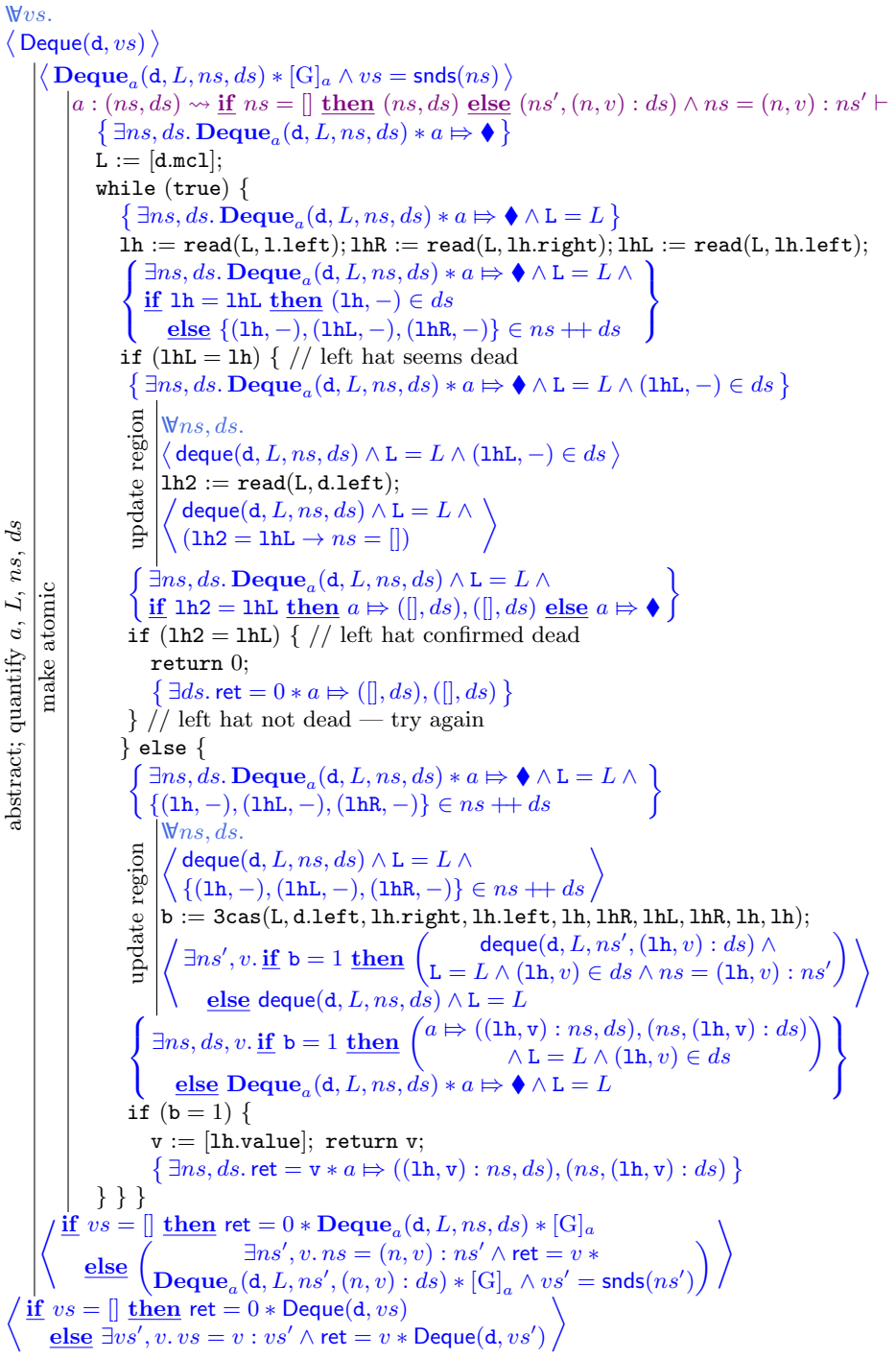

Fig. 8. Proof of the `popLeft` implementation.

The proof of soundness demonstrates that the semantic judgement obeys all of the syntactic proof rules. For the novel proof rules, such as "make atomic", the proof essentially establishes a simulation. Each step of the judgement in the conclusion of the rule is shown to correspond to a step in the judgement of the premiss. The technical report [15] gives the details.

6 Related Work

TaDA inherits from a family of logics deriving from concurrent separation logic [12]: RGSep [20], Deny-Guarantee [4], CAP [3], Higher-Order CAP (HOCAP) [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.* [6], 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, as in the MCAS example (§2.2). Such locality is in the spirit of local life stories [19], and can be seen as an instance of Ley-Wild and Nanevski's "subjective auxiliary state" [11].

While HOCAP and iCAP do not support abstract atomic specifications, they support an approach to atomicity introduced by Jacobs and Piessens [10] 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 has been extensive work understanding and generalising linearisability, especially in light of work on separation logic. Vafeiadis [20] has combined the ownership given by his RGSep reasoning with linearisability. Gotsman and Yang [8] have generalised linearisability to include ownership transfer of memory between a client and a module, which is also supported by our approach. Filipovic *et al.* [7] have demonstrated that linearisability can be viewed as a particular proof technique for contextual refinement. Turon *et al.* [18] have introduced CaReSL, a logic that combines contextual refinement and Hoare-style reasoning to prove higher-order concurrent programs. Like linearisability, contextual refinement requires a whole-module approach.

7 Conclusions

We have introduced a program logic, TaDA, which includes novel *atomic triples* for specifying abstract atomicity, as well as separation-style Hoare triples for

specifying abstract disjointness. We have specified and verified several example modules: an atomic lock module, which cannot be fully specified using linearisability; an atomic MCAS module implemented using our lock module, a classic linearisability example which cannot be done using concurrency abstract predicates; and a double-ended queue module implemented using MCAS. With the combination of abstract atomicity and abstract disjointness that TaDA provides, we can specify and verify modules with atomic and non-atomic operations, posisbly at different levels of abstraction. Moreover, we can easily extend modules with new operations, and build new modules on top of existing ones.

7.1 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 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. TaDA already has some potential to specify weak behaviours. For instance, the following three specifications for a read operation are increasingly weak:

$$\vdash \mathbb{V}v.\, \langle \mathtt{x} \mapsto v \rangle\ \mathtt{y} := [\mathtt{x}]\ \langle \mathtt{x} \mapsto v \wedge \mathtt{y} = v \rangle$$
$$\vdash \langle \mathtt{x} \mapsto v \rangle\ \mathtt{y} := [\mathtt{x}]\ \langle \mathtt{x} \mapsto v \wedge \mathtt{y} = v \rangle$$
$$\vdash \{ \mathtt{x} \mapsto v \}\ \mathtt{y} := [\mathtt{x}]\ \{ \mathtt{x} \mapsto v \wedge \mathtt{y} = v \}$$

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.

Acknowledgements. We thank Lars Birkedal, Daiva Naudžiūnienė, Matthew Parkinson, Julian Sutherland, Kasper Svendsen, Aaron Turon, Adam Wright, and the anonymous referees for discussions and useful feedback. This research was supported by an EPSRC Programme Grants EP/H008373/1 (all authors)

and EP/K008528/1 (Dinsdale-Young, Gardner), and the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (Dinsdale-Young).

References

1. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: ESOP. pp. 87–107 (2012)
2. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300 (2013)
3. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
4. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-Guarantee Reasoning. In: ESOP. pp. 363–377 (2009)
5. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele, Jr., G.L.: DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In: SPAA. pp. 216–224 (2004)
6. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. In: ICFP. pp. 143–156 (2010)
7. Filipović, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for Concurrent Objects. In: ESOP. pp. 252–266 (2009)
8. Gotsman, A., Yang, H.: Linearizability with ownership transfer. In: CONCUR. pp. 256–271 (2012)
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
10. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: POPL. pp. 271–282 (2011)
11. Ley-Wild, R., Nanevski, A.: Subjective auxiliary state for coarse-grained concurrency. In: POPL. pp. 561–574 (2013)
12. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (Apr 2007)
13. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL. pp. 247–258 (2005)
14. da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., Wheelhouse, M.: A simple abstraction for complex concurrent indexes. In: OOPSLA. pp. 845–864 (2011)
15. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. Tech. rep., Imperial College London (2014)
16. Svendsen, K., Birkedal, L.: Impredicative Concurrent Abstract Predicates. In: ESOP. pp. 149–168 (2014)
17. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: ESOP. pp. 169–188 (2013)
18. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In: ICFP. pp. 377–390 (2013)
19. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL. pp. 343–356 (2013)
20. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, Computer Laboratory (2008)
21. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving Correctness of Highly-concurrent Linearisable Objects. In: PPoPP. pp. 129–136 (2006)