Modular Termination Verification for Non-blocking Concurrency
(Extended Version)

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

[1] Imperial College London
{pmd09,pg,jhs110}@doc.ic.ac.uk
[2] Aarhus University
tyoung@cs.au.dk

Abstract. We present Total-TaDA, a program logic for verifying the total correctness of concurrent programs: that such programs both terminate and produce the correct result. With Total-TaDA, we can specify constraints on a thread's concurrent environment that are necessary to guarantee termination. This allows us to verify total correctness for non-blocking algorithms, e.g. a counter and a stack. Our specifications can express lock- and wait-freedom. More generally, they can express that one operation cannot impede the progress of another, a new non-blocking property we call *non-impedance*. Moreover, our approach is modular. We can verify the operations of a module independently, and build up modules on top of each other.

1 Introduction

The problem of understanding and proving the correctness of programs has been considered at least since Turing [20]. When proving a program, it is not just important to know that it will give the right answer, but also that the program terminates. This is especially challenging for concurrent programs. When multiple threads are changing some shared resource, knowing if each thread terminates can often depend on the behaviour of the other threads and even on the scheduler that decides which thread should run at a particular moment.

If we prove that a concurrent program only produces the right answer, we establish *partial correctness*. Many recent developments have been made in program logics for partial correctness of concurrent programs [5,21,18,16,11,17]. These logics emphasise a *modular* approach, which allows us to decouple the verification of a module's clients and its implementation. Each operation of the module is proven in isolation, and the reasoning is local to the thread. To achieve this, these logics abstract the interference between a thread and its environment.

These logics have been applied to reason about fine-grained concurrency, which is characterised by the use of low-level synchronisation operations (such as compare-and-swap). A well-known class of fine-grained concurrent programs is that of *non-blocking* algorithms. With non-blocking algorithms, suspension of a thread cannot halt the progress of other threads: the progress of a single thread cannot require another thread to be scheduled. Thus if the interference from the environment is suitably restricted, the operations are guaranteed to terminate.

If we prove that a program produces the correct results and also always completes in a finite time, we establish *total correctness*. Turing [20] and Floyd [6] introduced the use of well-founded relations, combined with partial-correctness arguments, to prove the termination of sequential programs. The same technique is general enough to prove concurrent programs too. However, previous applications of this technique in the concurrent setting, which we discuss in §7, do not support straight-forward reasoning about clients.

In this paper, we extend a particular concurrent program logic, TaDA [16], with well-founded termination reasoning. With the resulting logic, Total-TaDA, we can prove total correctness of fine-grained concurrent programs. The novelty of our approach is in using TaDA's abstraction mechanisms to specify constraints on the environment necessary to ensure termination. It retains the modularity of TaDA and abstracts the internal termination arguments. We demonstrate our approach on counter and stack algorithms.

We observe that Total-TaDA can be used to verify standard non-blocking properties of algorithms. However, our specifications capture more: we propose the concept of *non-impedance* that our specifications suggest. We say that one operation *impedes* another if the second can be prevented from terminating by repeated concurrent invocations of the first. This concept seems important to the design and use of non-blocking algorithms where we have some expectation about how clients use the algorithm, and what progress guarantees they expect.

TaDA. TaDA introduced a new form of specification, given by *atomic triples*, which supports local, modular reasoning and can express constraints on the concurrent environment. Simple atomic triples have the following form:

$$\vdash \mathbb{V} x \in X. \big\langle p(x) \big\rangle \ \mathbb{C} \ \big\langle q(x) \big\rangle$$

Intuitively, the specification states that the program $\mathbb{C}$ atomically updates $p(x)$ to $q(x)$ for an arbitrary $x \in X$. As we are in a concurrent setting, while $\mathbb{C}$ is executing, there might be interference from the environment before the atomic update. The pseudo-quantifier $\mathbb{V}$ restricts the interference: before the atomic update, the environment must maintain $p(x)$, but it is allowed to change the parameter as long as it stays within X; after the atomic update, the environment is not constrained. This specification thus provides a contract between the client of $\mathbb{C}$ and the implementation: the client can assume that the precondition holds for some $x \in X$ until it performs the update.

Using the atomic triple, an increment operation of a counter is specified as:[3]

$$\vdash \mathbb{V} n \in \mathbb{N}. \big\langle \mathsf{C}(s, \mathtt{x}, n) \big\rangle \ \mathtt{incr(x)} \ \big\langle \mathsf{C}(s, \mathtt{x}, n+1) \land \mathsf{ret} = n \big\rangle$$

[3] The parameter s of the abstract predicate was mistakenly abstracted in [16]. Technically, it is not possible to abstract it by existentially quantifying in the precondition of the atomic triple.

The internal structure of the counter is abstracted using the abstract predicate [14] $\mathsf{C}(s, x, n)$, which states that there is a counter at address x with value n and s abstracts implementation specific information about the counter. The specification says that the `incr` atomically increments the counter by 1. The environment is allowed to update the counter to any value of n as long as it is a natural number. The specification enforces obligations on both the client and the implementation: the client must guarantee that the counter is not destroyed and that its value is a natural number until the atomic update occurs; and the implementation must guarantee that it does not change the value of the counter until it performs the specified atomic action. Working at the abstraction of the counter means that each operation can be verified without knowing the rest of the operations of the module. Consequently, modules can be extended with new operations without having to re-verify the existing operations. Additionally, the implementation of `incr` can be replaced by another implementation that satisfies the same specification, without needing to re-verify the clients that make use of the counter. While atomic triples are expressive, they do not guarantee termination. In particular, an implementation could block, deadlock or live-lock and still be considered correct.

Non-blocking Algorithms. In general, guaranteeing the termination of concurrent programs is a difficult problem. In particular, termination could depend on the behaviour of the scheduler (whether or not it is *fair*) and of other threads that might be competing for resources. We focus on non-blocking programs. Non-blocking programs have the benefit that their termination is not dependent on the behaviour of the scheduler.

There are two common non-blocking properties: *wait-freedom* [8] and *lock-freedom* [13]. Wait-freedom requires that operations complete irrespective of the interference caused by other threads: termination cannot depend on the amount of interference caused by the environment. Lock-freedom is less restrictive. It requires that, when multiple threads are performing operations, then at least one of them must make progress. This means that a thread might never terminate if the amount of interference caused by the environment is unlimited.

TaDA is well suited to reasoning about interference between threads. In particular, we can write specifications that limit the amount of interference caused by the client, and so guarantee termination of lock-free algorithms. We will see how both wait-freedom and lock-freedom can be expressed in Total-TaDA.

Termination. Well-founded relations provide a general way to prove termination. In particular, Floyd [6] used well-founded relations to prove the termination of sequential programs. In fact, it is sufficient to use ordinal numbers [3] without losing expressivity. A 'Hoare-style' while rule, using ordinals and adapted from Floyd's work, has the form:

$$\frac{\forall \gamma \leq \alpha.\ \vdash_\tau \left\{p(\gamma) \land \mathbb{B}\right\}\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \land \beta < \gamma\right\}}{\vdash_\tau \left\{p(\alpha)\right\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \land \neg\mathbb{B} \land \beta \leq \alpha\right\}}$$

The loop invariant $p(\gamma)$ is parametrised by an ordinal γ (the *variant*) which is decreased by every execution of the loop body $\mathbb{C}$. Because ordinals cannot have infinite descending chains, the loop must terminate in a finite number of steps. This proof rule allows termination reasoning to be localised to the individual loops in the program. In this paper, we extend TaDA with termination based on ordinal numbers, using the `while` rule given above.

Total-TaDA. We obtain the program logic Total-TaDA by modifying TaDA to have a total-correctness semantics. The details are given in §3. With Total-TaDA, we can specify and verify non-blocking algorithms. Wait-free operations always terminate, independently of the operations performed by the environment. For lock-free operations however, we need to restrict the amount of interference the environment can cause in order to guarantee termination. Our key insight is that, as well as bounding the number of iterations of loops, ordinals can bound the interference on a module. This allows us to give total-correctness specifications for lock-free algorithms. In §2, we specify and verify lock-free implementations of a counter. The specification introduces ordinals to bound the number of times a client may update the counter. This makes it possible to guarantee that the lock-free increment operation will terminate, since either it will succeed or some other concurrent increment will succeed. As the number of increments is bounded, the operation must eventually succeed.

Total-TaDA retains the modularity of TaDA. In particular, we can verify the termination of clients of modules using the total-correctness specifications, without reference to the implementation. We show an example of this in §2.2. Since the client only depends on the specification, we can replace the implementation. In §2.3 we show that two different implementations of a counter satisfy the same total-correctness specification. With Total-TaDA we can verify the operations of a module independently, exploiting locality.

As a case study for Total-TaDA, we show how to specify and verify both functional correctness and termination of Treiber's stack in §4. In §5, we discuss the implications of a total-correctness semantics for the soundness proof of Total-TaDA. In §6, we show how lock-freedom and wait-freedom can be expressed with Total-TaDA specifications. We also introduce the concept of non-impedance in §6.3 and argue for its value in specifying non-blocking algorithms. We discuss related work in §7 and future directions in §8.

2 Motivating Examples: Counters

We introduce Total-TaDA by providing specifications of the operations of a counter module. We justify the specifications by using them to reason about two clients, one sequential and one concurrent. We show how two different implementations can be proved to satisfy the specification.

Our underlying programming language is a concurrent while language with functions, allocation and the atomic assignment $\mathtt{x} := E$, read $E := [E]$, write $[E] := E$ and compare-and-swap $\mathtt{x} := \mathtt{CAS}(E, E, E)$, where expressions E have no side effects. Consider a counter module with a constructor `makeCounter` and two operations: `incr` that increments the value of the counter by 1 and returns

```
function makeCounter() {
  x := alloc(1);
  [x] := 0;
  return x;
}

function read(x) {
  v := [x];
  return v;
}
```

```
function incr(x) {
  b := 0;
  while (b = 0) {
    v := [x];
    b := CAS(x, v, v + 1);
  }
  return v;
}
```

(a) Spin counter operations.

```
function incr(x) {
  n := 0; b := 0;
  while (b = 0) {
    if (n = 0) {
      v := [x];
      b := CAS(x, v, v + 1);
      n := random();
    } else {
      n := n − 1;
    }
  }
  return v;
}
```

(b) Backoff increment.

Fig. 1: Counter module implementations.

its previous value; and `read` that returns the value of the counter. We give an implementation in Fig. 1a, and an alternative implementation of `incr` in Fig. 1b.

2.1 Abstract Specification

The Total-TaDA specification for the `makeCounter()` operation is a Hoare triple with a total-correctness interpretation:

$$\forall \alpha.\ \vdash_\tau \{\mathsf{emp}\}\ \mathtt{x} := \mathtt{makeCounter}()\ \{\exists s.\, \mathsf{C}(s, \mathtt{x}, 0, \alpha)\}$$

The counter predicate is extended with an ordinal parameter, α, that provides a bound on the amount of interference the counter can sustain. When the value of the counter is updated, the ordinal α must decrease.

The operation allocates a new counter, with value 0, and allows the client to pick an initial ordinal α. If a finite bound on the number of updates is already determined, then that is an appropriate choice for the ordinal. However, it could be that the bound is determined by subsequent (non-deterministic) operations, in which case an infinite ordinal should be used. For example, consider the following client program:

```
x := makeCounter();
m := random();
while (m > 0) {
  incr(x); m := m − 1;
}
```

Here, the number of increments is bounded by the (finite) value returned by `random`, but it is not determined when the counter is constructed. Choosing $\alpha = \omega$ (the first infinite ordinal) is appropriate in this case: the first increment can decrease the ordinal from ω to $\mathtt{m} - 1$, while subsequent increments simply decrement the ordinal by 1.

The increment operation is specified as follows:

$$\forall \beta.\ \vdash_\tau \mathbb{W} n \in \mathbb{N}, \alpha. \quad \left\langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \land \alpha > \beta(n, \alpha) \right\rangle \\ \mathtt{incr}(\mathtt{x}) \\ \left\langle \mathsf{C}(s, \mathtt{x}, n+1, \beta(n, \alpha)) \land \mathsf{ret} = n \right\rangle$$

The specification resembles the partial-correctness specification given in the introduction, but with the addition of the ordinal α and the function β. The client chooses how to decrease the ordinal by providing a function β that determines the new ordinal in terms of the old ordinal and previous value of the counter. The condition $\alpha > \beta(n, \alpha)$ requires the client to guarantee that such a decrease is possible. (So, for example, the client could not use the specification in a situation where the concurrent environment might reduce the ordinal to zero.) The implementation may rely on the fact that a counter's ordinal cannot be increased to guarantee termination.

The read operation is specified as follows:

$$\vdash_\tau \mathbb{W} n \in \mathbb{N}, \alpha. \left\langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \right\rangle\ \mathtt{read}(\mathtt{x})\ \left\langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \land \mathsf{ret} = n \right\rangle$$

Unlike the increment, the read operation does not affect the ordinal. This means that the client is not bounded with respect to the number of reads it performs. Such a specification is possible for operations that do not impede the progress of other operations. In this case, `read` does not impede `incr` or `read`.

Finally, we give an axiom that allows the client to decrease the ordinal without requiring any physical operation.

$$\forall s, n, \alpha, \beta < \alpha.\, \mathsf{C}(s, \mathtt{x}, n, \alpha) \implies \mathsf{C}(s, \mathtt{x}, n, \beta)$$

This is possible because the ordinals do not have any concrete representation in memory. They are just a logical mechanism to limit the amount of interference over a resource.

The ordinal parameter is exposed in the specification of the counter to allow the implementation to guarantee that its loops terminate. In a wait-free implementation it would not be necessary to expose the ordinal parameter. For this counter, the read operation is wait-free, while the increment operation is lock-free, since termination depends on bounding the number of interfering increments.

2.2 Clients

Sequential Client. Consider a program that creates a counter and contains two nested loops. As in the previous example, the outer loop runs a finite but randomly determined number of times. The inner loop also runs a randomly determined number of times, and increments the counter on each iteration. Fig. 2 shows this client, together with its total-correctness proof.

The while rule is used for each of the loops: for the outer loop, the variant is `n`; for the inner loop, the variant is `m`. Since the number of iterations of each loop is determined before it is run, the variants need only be considered up to

$\{\mathsf{emp}\}$
$\mathtt{x} := \mathtt{makeCounter}();$
$\{\exists s.\, \mathsf{C}(s, \mathtt{x}, 0, \omega^2)\}$
$\mathtt{n} := \mathtt{random}();$
$\{\exists s.\, \mathsf{C}(s, \mathtt{x}, 0, \omega \cdot \mathtt{n})\}$
$\mathtt{while}\ (\mathtt{n} > 0)\ \{$
$\quad \forall \gamma.$
$\quad \{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot \mathtt{n}) \wedge \gamma = \mathtt{n} \wedge \mathtt{n} > 0\}$
$\quad \mathtt{m} := \mathtt{random}();$
$\quad \{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot (\mathtt{n} - 1) + \mathtt{m}) \wedge \gamma = \mathtt{n} \wedge \mathtt{n} > 0\}$
Frame: $\gamma = \mathtt{n} \wedge \mathtt{n} > 0$
$\quad\quad \mathtt{while}\ (\mathtt{m} > 0)\ \{$
$\quad\quad\quad \forall \delta.$
$\quad\quad\quad \{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot (\mathtt{n} - 1) + \mathtt{m}) \wedge \delta = \mathtt{m} \wedge \mathtt{m} > 0\}$
$\quad\quad\quad \mathtt{incr}(\mathtt{x});$
$\quad\quad\quad \{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot (\mathtt{n} - 1) + \mathtt{m} - 1) \wedge \delta = \mathtt{m} > 0\}$
$\quad\quad\quad \mathtt{m} := \mathtt{m} - 1;$
$\quad\quad\quad \{\exists s, \zeta, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot (\mathtt{n} - 1) + \mathtt{m}) \wedge \zeta = \mathtt{m} \wedge \zeta < \delta\}$
$\quad\quad \}$
$\quad \{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot (\mathtt{n} - 1)) \wedge \gamma = \mathtt{n} \wedge \mathtt{n} > 0\}$
$\quad \mathtt{n} := \mathtt{n} - 1;$
$\quad \{\exists s, \beta, v.\, \mathsf{C}(s, \mathtt{x}, v, \omega \cdot \mathtt{n}) \wedge \beta = \mathtt{n} \wedge \beta < \gamma\}$
$\}$
$\{\exists s, v.\, \mathsf{C}(s, \mathtt{x}, v, 0)\}$

Fig. 2: Proof of a sequential client of the counter.

finite ordinals (*i.e.* natural numbers). (We could modify the code to use a single loop that conditionally decrements $\mathtt{n}$ (and randomises $\mathtt{m}$) or decrements $\mathtt{m}$. This variation would require a transfinite ordinal for the variant.)

As well as enforcing loop termination, ordinals play a role as a parameter to the C predicate, which must be decreased on each increment. When we create the counter, we choose ω^2 as the initial ordinal. We have seen that ω allows us to decrement the counter a non-deterministic (but finite) number of times. We want to repeat this a non-deterministic (but finite) number of times, so $\omega \cdot \omega = \omega^2$ is the appropriate ordinal. Once the number $\mathtt{n}$ of iterations of the outer loop is determined, we decrease this to $\omega \cdot \mathtt{n}$ by using the axiom provided by the counter module. Similarly, when $\mathtt{m}$ is chosen, we decrease the ordinal from $\omega \cdot \mathtt{n} = \omega \cdot (\mathtt{n} - 1) + \omega$ to $\omega \cdot (\mathtt{n} - 1) + \mathtt{m}$.

Concurrent Client. Consider a program that creates two threads, each of which increments the counter a finite but unbounded number of times. We again prove this client using the abstract specification of the counter. The proof is given in Fig. 3. In this example, the counter is shared between the two threads, which may concurrently update it. To reason about sharing, we use a *shared region*.

As in TaDA, a shared region encapsulates some resource that is available to multiple threads. Threads can access the resource when performing (abstractly) atomic operations, such as `incr`. The region presents an abstract state, and defines a protocol that determines how the region may be updated. Ghost resources, called *guards*, are associated with transitions in the protocol. The guards for a region form a partial commutative monoid with the operation $\bullet$, which is lifted by $*$ in assertions. In order for a thread to make a particular update, it must have ownership of a guard associated with the corresponding transition. All guards are allocated along with the region they are associated with.

For the concurrent client, we introduce a region with type name **CClient**. This region encapsulates the shared counter. Accordingly, the region type is parametrised by the address of the counter. The abstract state of the region records the current value of the counter.

There are two types of guard resources associated with **CClient** regions. The guard $\textsc{Inc}(m, \beta, \pi)$ provides capability to increment the counter. Conceptually, multiple threads may have $\textsc{Inc}$ guards, and a fractional permission $\pi \in (0, 1]$ (in the style of [2]) is used to keep track of these capabilities. The parameter m expresses the *local contribution* to the value of the counter — the actual value is the sum of the local contributions. The ordinal parameter β represents a local bound on the number of increments. Again, the actual bound is a sum of the local bounds. Standard ordinal addition is inconvenient since it is not commutative; we use the natural (or Hessenberg) sum [9], denoted $\oplus$, which is associative, commutative, and monotone in its arguments.

To allow the $\textsc{Inc}$ guard to be shared among threads, we impose the following equivalence on guards:

$$\textsc{Inc}(n + m, \alpha \oplus \beta, \pi_1 + \pi_2) = \textsc{Inc}(n, \alpha, \pi_1) \bullet \textsc{Inc}(m, \beta, \pi_2)$$

where $n \geq 0$, $m \geq 0$ and $1 \geq \pi_1 + \pi_2 > 0$. This equivalence expresses that $\textsc{Inc}$ guards can be split (or joined), preserving the total contribution to the value of the counter, ordinal bound and permission.

The second type of guard resource is $\textsc{Total}(n, \alpha)$, which tracks the actual value of the counter n and ordinal α. These values should match the totals for the $\textsc{Inc}$ guards, which we enforce by requiring the following implication to hold:

$$\textsc{Total}(n, \alpha) \bullet \textsc{Inc}(m, \beta, 1) \text{ defined} \implies n = m \land \alpha = \beta$$

We wish to allow the contributions recorded in $\textsc{Inc}$ guards to change, but to do so we must simultaneously update the $\textsc{Total}$ guard, as expressed by the following equivalence:

$$\textsc{Total}(n + m, \alpha \oplus \beta) \bullet \textsc{Inc}(m, \beta, \pi) = \textsc{Total}(n + m', \alpha \oplus \beta') \bullet \textsc{Inc}(m', \beta', \pi)$$

(We have constructed an instance of the authoritative monoid of Iris [11].)

The possible states of **CClient** regions are the natural numbers $\mathbb{N}$, representing the value of the shared counter, together with the distinguished state $\circ$, representing that the region is no longer required. The protocol for a region is specified by a guarded transition system, which describes how the abstract state may be updated in atomic steps, and which guard resources are required to do so. The transitions for **CClient** regions are as follows:

$$\textsc{Inc}(m, \gamma, \pi) : n \rightsquigarrow n + 1 \qquad\qquad \textsc{Inc}(m, \gamma, 1) : n \rightsquigarrow \circ$$

This specifies that any thread with an Inc guard may increment the value of the counter, and a thread owning the full Inc guard may dispose of the region.

It remains to define the interpretation of the region states:

$$I(\mathbf{CClient}_r(s, x, n)) \triangleq \exists \alpha.\, \mathsf{C}(s, x, n, \alpha) * [\textsc{Total}(n, \alpha)]_r$$
$$I(\mathbf{CClient}_r(s, x, \circ)) \triangleq \mathsf{True}$$

By interpreting the state $\circ$ as True, we allow a thread transitioning into that state to acquire the counter that previously belonged to the region. (This justifies the last step of the proof in Fig. 3.)

The proof rule that allows us to use the atomic specification of the `incr` operation to update the shared region is the $\mathsf{use\ atomic}$ rule, inherited from TaDA. A simplified version of the rule is as follows:

$$\frac{\forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \vdash_\tau \mathbb{V} x \in X. \big\langle I(\mathbf{t}_a(x)) * [\mathrm{G}]_a \big\rangle\ \mathbb{C}\ \big\langle I(\mathbf{t}_a(f(x))) * q \big\rangle}{\vdash_\tau \big\{ \exists x \in X.\, \mathbf{t}_a(x) * [\mathrm{G}]_a \big\}\ \mathbb{C}\ \big\{ \exists x \in X.\, \mathbf{t}_a(f(x)) * q \big\}}$$

In the conclusion of the rule, the abstract state of the region a (of type $\mathbf{t}$) is updated according to the function f. The first premiss requires that this update is allowed by the transition system for the region, given the guard resources available (G). The second premiss requires that the program $\mathbb{C}$ (abstractly) atomically performs the corresponding update on the concrete state of the region.

The $\{\}$-assertions in Total-TaDA are required to be stable. That is, the region states must account for the possible changes that the concurrent environment could make, under the assumption that it has guards that are compatible with those of the thread. This is why, for instance, in Fig. 3 the state of the **CClient** region is always existentially quantified.

2.3 Implementations

We prove the total correctness of the two distinct increment implementations against the abstract specification given in §2.1.

Spin Counter Increment. Consider `incr` shown in Fig. 1a. Note that the read, write and compare-and-swap operations are atomic. We want to prove the total correctness of `incr` against the atomic specification. The first step is to give a concrete interpretation of the abstract predicate $\mathsf{C}(s, x, n, \alpha)$. We introduce a new region type, **Counter**, with only one non-empty guard, G. The abstract states of the region are pairs of the form (n, α), where n is the value of the counter and α is a bound on the number of increments. All transitions are guarded by G with the transition:

$$\mathrm{G} : \forall n \in \mathbb{N}, m \in \mathbb{N}, \alpha > \beta.\, (n, \alpha) \rightsquigarrow (n + m, \beta)$$

The transition requires that updates to the state of the region must decrease the ordinal. This allows us to effectively bound interference, which is necessary to guarantee the termination of the loop in `incr`.

The interpretation of the `Counter` region states is defined as follows:

$$I(\mathbf{Counter}_r(x, n, \alpha)) \triangleq x \mapsto n$$

$$
\begin{array}{c}
\{\mathsf{emp}\} \\
\mathtt{x} := \mathtt{makeCounter}(); \\
\{\exists s.\, \mathsf{C}(s, \mathtt{x}, 0, \omega \oplus \omega)\} \\
\{\exists s, r.\, \mathbf{CClient}_r(s, \mathtt{x}, 0) * [\textsc{Inc}(0, \omega \oplus \omega, 1)]_r\} \\
\begin{array}{l||l}
\{\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) * [\textsc{Inc}(0, \omega, \frac{1}{2})]_r \land 0 \le v\} & \left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) \\ * [\textsc{Inc}(0, \omega, \frac{1}{2})]_r\end{array}\right\} \\
\mathtt{n} := \mathtt{random}();\ \mathtt{i} := 0; & \mathtt{m} := \mathtt{random}(); \\
\{\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \mathtt{n}, \frac{1}{2})]_r \land 0 \le v \land \mathtt{i} = 0\} & \mathtt{j} := 0; \\
\mathtt{while}\ (\mathtt{i} < \mathtt{n})\ \{ & \mathtt{while}\ (\mathtt{j} < \mathtt{m})\ \{ \\
\quad \forall \beta.\ \left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \beta, \frac{1}{2})]_r \land \mathtt{i} \le v \\ \land\, \mathtt{i} < \mathtt{n} \land \beta = \mathtt{n} - \mathtt{i}\end{array}\right\} & \quad \mathtt{incr}(\mathtt{x}); \\
\qquad \mathtt{incr}(\mathtt{x});\ \mathtt{i} := \mathtt{i} + 1; & \quad \mathtt{j} := \mathtt{j} + 1; \\
\qquad \left\{\begin{array}{l}\exists s, \delta, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \delta, \frac{1}{2})]_r \land \mathtt{i} \le v \\ \land\, \mathtt{i} \le \mathtt{n} \land \delta = \mathtt{n} - \mathtt{i} \land \delta < \beta\end{array}\right\} & \} \\
\} & \left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) \\ * [\textsc{Inc}(\mathtt{m}, 0, \frac{1}{2})]_r\end{array}\right\} \\
\{\exists s, v.\, \mathbf{CClient}_r(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{n}, 0, \frac{1}{2})]_r\} &
\end{array} \\
\{\exists s, r.\, \mathbf{CClient}_r(s, \mathtt{x}, \mathtt{n} + \mathtt{m}) * [\textsc{Inc}(\mathtt{n} + \mathtt{m}, 0, 1)]_r\} \\
\{\exists s.\, \mathsf{C}(s, \mathtt{x}, \mathtt{n} + \mathtt{m}, 0)\}
\end{array}
$$

Fig. 3: Proof of a concurrent client of the counter.

The expression $x \mapsto n$ asserts that there exists a heap cell with address x and value n. Note that α is not represented in the concrete heap, as it is not part of the program. We use it solely to ensure that the number of operations is finite.

We define the interpretation of the abstract predicate as follows:

$$\mathsf{C}(r, x, n, \alpha) \triangleq \mathbf{Counter}_r(x, n, \alpha) * [\mathrm{G}]_r$$

The abstract predicate $\mathsf{C}(r, x, n, \alpha)$ asserts that there is a **Counter** region with identifier r, address x, and with abstract state (n, α). Furthermore, it encapsulates exclusive ownership of the guard G, and so embodies exclusive permission to update the counter. (Note that the type of the first parameter of C, which is abstract to the client, is instantiated as RId.)

The specification for the increment is atomic and as such, we use the make atomic rule from TaDA. A slightly simplified version of the rule is as follows:

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

This rule establishes in its conclusion that $\mathbb{C}$ atomically updates region a from some state $x \in X$ to some state $y \in Q(x)$. The first premiss requires that the available guard G permits this update, according to the transition system. The second premiss essentially establishes that $\mathbb{C}$ will perform a single atomic update on region a, corresponding to the required update. The *atomicity context* $a : x \in X \rightsquigarrow Q(x)$ records the update we require. The program is given the atomic tracking resource $a \Mapsto \blacklozenge$ initially (in place of the guard G); this resource

$$
\begin{array}{l}
\forall \beta.\, \mathbb{W} n \in \mathbb{N}, \alpha. \\
\left\langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \land \alpha > \beta(n, \alpha) \right\rangle \\
\quad \left\langle \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * [\mathrm{G}]_r \land \alpha > \beta(n, \alpha) \right\rangle \\
\qquad r : (n, \alpha) \land n \in \mathbb{N} \land \alpha > \beta(n, \alpha) \rightsquigarrow (n+1, \beta(n, \alpha)) \vdash_\tau \\
\qquad \left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \land \alpha > \beta(n, \alpha) \right\} \\
\qquad \mathtt{b} := 0; \\
\qquad \left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \land \mathtt{b} = 0 \land \alpha > \beta(n, \alpha) \right\} \\
\qquad \mathtt{while}\ (\mathtt{b} = 0)\ \{ \\
\qquad\quad \forall \gamma. \\
\qquad\quad \left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \land \mathtt{b} = 0 \land \gamma \geq \alpha > \beta(n, \alpha) \right\} \\
\qquad\qquad \mathbb{W} n \in \mathbb{N}, \alpha. \\
\qquad\qquad \left\langle \mathtt{x} \mapsto n \land \gamma \geq \alpha > \beta(n, \alpha) \right\rangle \\
\qquad\qquad \mathtt{v} := [\mathtt{x}]; \\
\qquad\qquad \left\langle \mathtt{x} \mapsto n \land \mathtt{v} = n \land \gamma \geq \alpha > \beta(n, \alpha) \land (n > \mathtt{v} \Rightarrow \gamma > \alpha) \right\rangle \\
\qquad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \land \\ \gamma \geq \alpha > \beta(n, \alpha) \land n \geq \mathtt{v} \land (n > \mathtt{v} \Rightarrow \gamma > \alpha) \end{array} \right\} \\
\qquad\qquad \mathbb{W} n \in \mathbb{N}, \alpha. \\
\qquad\qquad \left\langle \mathtt{x} \mapsto n \land \gamma \geq \alpha > \beta(n, \alpha) \land n \geq \mathtt{v} \land (n > \mathtt{v} \Rightarrow \gamma > \alpha) \right\rangle \\
\qquad\qquad \mathtt{b} := \mathtt{CAS}(\mathtt{x}, \mathtt{v}, \mathtt{v} + 1); \\
\qquad\qquad \left\langle \begin{array}{l} \alpha > \beta(n, \alpha) \land \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \gamma > \alpha \land \mathtt{x} \mapsto n \\ \quad \underline{\mathbf{else}}\ \mathtt{x} \mapsto n + 1 \land \mathtt{v} = n \end{array} \right\rangle \\
\qquad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \gamma \geq \alpha > \beta(n, \alpha) \land \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \begin{pmatrix} \mathbf{Counter}_r(\mathtt{x}, n, \alpha) \\ * r \Mapsto \blacklozenge \land \gamma > \alpha \end{pmatrix} \\ \quad \underline{\mathbf{else}}\ r \Mapsto ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\} \\
\qquad \} \\
\qquad \left\{ \exists n \in \mathbb{N}, \alpha.\, r \Mapsto ((n, \alpha), (n + 1, \beta(n, \alpha))) \land \mathtt{v} = n \right\} \\
\qquad \mathtt{return}\ \mathtt{v}; \\
\qquad \left\{ \exists n \in \mathbb{N}, \alpha.\, r \Mapsto ((n, \alpha), (n + 1, \beta(n, \alpha))) \land \mathsf{ret} = n \right\} \\
\quad \left\langle \mathbf{Counter}_r(\mathtt{x}, n + 1, \beta(n, \alpha)) * [\mathrm{G}]_r \land \mathsf{ret} = n \right\rangle \\
\left\langle \mathsf{C}(s, \mathtt{x}, n, \beta(n, \alpha)) \land \mathsf{ret} = n \right\rangle
\end{array}
$$

abstract; substitute $s = r$

make atomic

open region

update region

Fig. 4: Proof of total correctness of increment.

permits a single update to the region in accordance with the atomicity context, while at the same time guaranteeing that the region's state will remain within X. When the single update occurs, the atomic tracking resource simultaneously changes to record the actual update performed: $a \Mapsto (x, y)$.

The $\mathsf{make\ atomic}$ rule of Total-TaDA is just the same as that of TaDA. The only difference is that termination is enforced. Whereas in TaDA it would be possible for an abstract atomic operation to loop forever without performing its atomic update, in Total-TaDA it is guaranteed to eventually perform the update.

A proof of the increment implementation is shown in Fig. 4. The atomicity context allows the environment to modify the abstract state of the counter. However, it makes no restriction on the number of times. The **Counter** transition system enforces that the ordinal α must decrease every time the value of the counter is increased. This means that the number of times the region's abstract state is updated is finite. Our loop invariant is parametrised with a variant γ that takes the value of α at the beginning of each loop iteration. When we first read the value of the counter n, we can assert: $n > \mathtt{v} \Rightarrow \gamma > \alpha$.

If the compare-and-swap operation fails, the value of the counter has changed. This can only happen in accordance with the region's transition system, and so the ordinal parameter α must have decreased. As such, the invariant still holds but for a lower ordinal, $\alpha < \gamma$. We are localising the termination argument for the loop, by relating the local variant with the ordinal parametrising the region.

If the compare-and-swap succeeds, then we record our update from $(\mathtt{v}, \alpha)$ to $(\mathtt{v}+1, \beta(\mathtt{v}, \alpha))$, where β is the function chosen by the client that determines how the ordinal is reduced. The $\mathsf{make\ atomic}$ rule allows us to export this update in the postcondition of the whole operation.

Backoff Increment. Consider a different implementation of the increment operation, given in Fig. 1b. Like the previous implementation, it loops attempting to perform the operation. However, if the compare-and-swap fails due to contention, it waits for a random number of iterations before retrying.

Despite the differences to the previous increment, the specification is the same. In fact, we can give the same interpretation for the abstract predicate $\mathsf{C}(x, n, \alpha)$, and the same guards and regions that were used for the previous implementation. (Since this is the case, a counter module could provide *both* of these operations: the proof system guarantees that they work correctly together.)

The main difference in the proof is that each iteration of the loop depends on not only the amount of interference on the counter, but also on the variable $\mathtt{n}$ that is randomised when the compare-and-swap fails. Any random number will be smaller than ω, and the maximum amount of times that the compare-and-swap can fail is α, the parameter of the C predicate. This is because α is a bound on the number of times the counter can be incremented. We therefore use $\omega \cdot \alpha + \mathtt{n}$ as the upper bound on the number of loop iterations.

Let γ be equal to $\omega \cdot \alpha + \mathtt{n}$ at the start of the loop iteration. At each loop iteration, we have two cases, when $\mathtt{n} = 0$ or otherwise. In the first case we try to perform the increment by doing a compare-and-swap. If the compare-and-swap succeeds, then the increment occurs and the loop will exit. If it fails, then the environment must have decreased α. This means that $\gamma \geq \omega \cdot \alpha + \omega$ for the new value of α. We then set $\mathtt{n}$ to be a new random number, which is less than ω, and end up with $\gamma > \omega \cdot \alpha + \mathtt{n}$. In the second case of the loop iteration, we simply decrement $\mathtt{n}$ by 1 and we know that $\gamma > \omega \cdot \alpha + \mathtt{n}$ for the new value of $\mathtt{n}$. The proof of the backoff increment is shown in Fig. 5.

3 Logic

Total-TaDA is a Hoare logic which, for the first time, can be used to prove total correctness for fine-grained non-blocking concurrent programs. The logic is essentially the same as for TaDA, simply adapted to incorporate termination analysis using ordinals in a standard way.

$$
\begin{array}{l}
\forall\beta.\, \mathbb{V}n \in \mathbb{N}, \alpha. \\
\langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \beta(n, \alpha) \rangle \\
\quad \langle \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * [\mathrm{G}]_r \wedge \alpha > \beta(n, \alpha) \rangle \\
\quad\quad r : (n, \alpha) \wedge n \in \mathbb{N} \wedge \alpha > \beta(n, \alpha) \rightsquigarrow (n+1, \beta(n, \alpha)) \vdash_\tau \\
\quad\quad \{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \alpha > \beta(n, \alpha) \} \\
\quad\quad \mathtt{n} := 0; \mathtt{b} := 0; \\
\quad\quad \{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \mathtt{n} = 0 \wedge \mathtt{b} = 0 \wedge \alpha > \beta(n, \alpha) \} \\
\quad\quad \mathtt{while}\ (\mathtt{b} = 0)\ \{ \\
\quad\quad\quad \forall\gamma. \\
\quad\quad\quad \{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \mathtt{b} = 0 \wedge \gamma \geq \omega \cdot \alpha + \mathtt{n} \wedge \alpha > \beta(n, \alpha) \} \\
\quad\quad\quad \mathtt{if}\ (\mathtt{n} = 0)\ \{ \\
\quad\quad\quad\quad \{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \gamma \geq \omega \cdot \alpha \wedge \alpha > \beta(n, \alpha) \} \\
\quad\quad\quad\quad\quad \mathbb{V}n \in \mathbb{N}, \alpha. \\
\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto n \wedge \gamma \geq \omega \cdot \alpha \wedge \alpha > \beta(n, \alpha) \rangle \\
\quad\quad\quad\quad\quad \mathtt{v} := [\mathtt{x}]; \\
\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto n \wedge \mathtt{v} = n \wedge \gamma \geq \omega \cdot \alpha \wedge \alpha > \beta(n, \alpha) \wedge (n > \mathtt{v} \Rightarrow \gamma \geq \omega \cdot \alpha + \omega) \rangle \\
\quad\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \wedge \gamma \geq \omega \cdot \alpha \wedge \alpha > \beta(n, \alpha) \\ \wedge\, n \geq \mathtt{v} \wedge (n > \mathtt{v} \Rightarrow \gamma \geq \omega \cdot \alpha + \omega) \end{array} \right\} \\
\quad\quad\quad\quad\quad \mathbb{V}n \in \mathbb{N}, \alpha. \\
\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto n \wedge \gamma \geq \omega \cdot \alpha \wedge \alpha > \beta(n, \alpha) \wedge n \geq \mathtt{v} \wedge (n > \mathtt{v} \Rightarrow \gamma \geq \omega \cdot \alpha + \omega) \rangle \\
\quad\quad\quad\quad\quad \mathtt{b} := \mathtt{CAS}(\mathtt{x}, \mathtt{v}, \mathtt{v} + 1); \\
\quad\quad\quad\quad\quad \left\langle \begin{array}{l} \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \gamma \geq \omega \cdot \alpha + \omega \wedge \mathtt{x} \mapsto n \\ \quad \underline{\mathbf{else}}\ \mathtt{x} \mapsto n + 1 \wedge \mathtt{v} = n \end{array} \right\rangle \\
\quad\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \begin{pmatrix} \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \\ \wedge\, \gamma \geq \omega \cdot \alpha + \omega \end{pmatrix} \\ \quad \underline{\mathbf{else}}\ r \Rrightarrow ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\} \\
\quad\quad\quad\quad \mathtt{n} := \mathtt{random}(); \\
\quad\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \begin{pmatrix} \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \\ \wedge\, \gamma > \omega \cdot \alpha + \mathtt{n} \end{pmatrix} \\ \quad \underline{\mathbf{else}}\ r \Rrightarrow ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\} \\
\quad\quad\quad \}\ \mathtt{else}\ \{ \\
\quad\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \\ \wedge\, \mathtt{b} = 0 \wedge \gamma \geq \omega \cdot \alpha + \mathtt{n} \wedge \alpha > \beta(n, \alpha) \end{array} \right\} \\
\quad\quad\quad\quad \mathtt{n} := \mathtt{n} - 1; \\
\quad\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \\ \wedge\, \mathtt{b} = 0 \wedge \gamma > \omega \cdot \alpha + \mathtt{n} \wedge \alpha > \beta(n, \alpha) \end{array} \right\} \\
\quad\quad\quad \} \\
\quad\quad\quad \left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \begin{pmatrix} \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Rrightarrow \blacklozenge \\ \wedge\, \gamma > \omega \cdot \alpha + \mathtt{n} \end{pmatrix} \\ \quad \underline{\mathbf{else}}\ r \Rrightarrow ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\} \\
\quad\quad \} \\
\quad\quad \{ \exists n \in \mathbb{N}, \alpha, .\, r \Rrightarrow ((n, \alpha), (n + 1, \beta(n, \alpha))) \wedge \mathtt{v} = n \} \\
\quad\quad \mathtt{return}\ \mathtt{v}; \\
\quad\quad \{ \exists n \in \mathbb{N}, \alpha.\, r \Rrightarrow ((n, \alpha), (n + 1, \beta(n, \alpha))) \wedge \mathsf{ret} = n \} \\
\quad \langle \mathbf{Counter}_r(\mathtt{x}, n + 1, \beta(n, \alpha)) * [\mathrm{G}]_r \wedge \mathsf{ret} = n \rangle \\
\langle \mathsf{C}(s, \mathtt{x}, n, \beta(n, \alpha)) \wedge \mathsf{ret} = n \rangle
\end{array}
$$

open region

update region

make atomic

abstract; substitute $s = r$

Fig. 5: Proof of total correctness of backoff increment.

Total-TaDA assertions, ranged over by $p, q, \ldots$, are constructed from the standard assertions of separation logic [15], plus abstract predicates, region predicates and tokens, examples of which are given in §2. The Total-TaDA proof judgement has the form:

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

In our examples, the atomicity context $\mathcal{A}$ describes an update to a single region. In general, $\mathcal{A}$ may describe updates to multiple regions (although only one update to each).[4] The pre- and postconditions are split into a private part (the p_p and $q_p(x, y)$) and a public part (the $p(x)$ and $q(x, y)$). The idea is that the command may make multiple, non-atomic updates to the private part, but must only make a single atomic update to the public part. Before the atomic update, the environment is allowed to change the public part of the state, but only by changing the parameter x of p which must remain within X. After the atomic update, the specification makes no constraint on how the environment modifies the public state. All that is known is that, immediately after the atomic update, the public and private parts satisfy the postcondition for a common value of y. The private assertions in our judgements must be *stable*: that is, they must account for any updates other threads could have sufficient resources to perform.

The non-atomic Hoare triple $\vdash_\tau \{p\} \ \mathbb{C} \ \{q\}$ is syntactic sugar for the judgement $\vdash_\tau \langle p \mid \mathsf{true} \rangle \ \mathbb{C} \ \langle q \mid \mathsf{true} \rangle$. The atomic triple $\vdash_\tau \mathbb{V}x \in X. \langle p(x) \rangle \ \mathbb{C} \ \langle q(x) \rangle$ is syntactic sugar for the judgement $\vdash_\tau \mathbb{V}x \in X. \langle \mathsf{true} \mid p(x) \rangle \ \mathbb{C} \ \langle \mathsf{true} \mid q(x) \rangle$.

We give an overview of the key Total-TaDA proof rules that deal with termination and atomicity in Fig. 6. The while rule enforces that the number of times that the loop body can run is finite. The rule allows us to perform a while loop if we can guarantee that each loop iteration decreases the ordinal parametrising the invariant p. By the finite-chain property of ordinals, there cannot be an infinite number of iterations.

The parallel rule and the frame rule are analogous to those for separation logic. The parallel rule allows us to split resources among two threads as long as the resources of one thread are not touched by the other thread. The frame rule allows us to add the frame 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.

The next three rules allow us to access the contents 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 of the premiss. (The region is in the public part in the conclusion also, but may be moved by 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.

[4] We have omitted region levels, analogous to those in TaDA, in our judgements to simplify our presentation. They prevent a region from being opened twice within a single branch of the proof tree, which unsoundly duplicates resources.

while rule

$$\frac{\forall \gamma \leq \alpha.\, \mathcal{A} \vdash_\tau \left\{p(\gamma) \wedge \mathbb{B}\right\}\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \beta < \gamma\right\}}{\mathcal{A} \vdash_\tau \left\{p(\alpha)\right\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha\right\}}$$

parallel rule

$$\frac{\forall i \in \{1,2\}\,.\, \mathcal{A} \vdash_\tau \left\{p_i\right\}\ \mathbb{C}_i\ \left\{q_i\right\}}{\mathcal{A} \vdash_\tau \left\{p_1 * p_2\right\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \left\{q_1 * q_2\right\}}$$

frame rule

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

open region rule

$$\frac{\mathcal{A} \vdash_\tau \mathbb{V}x \in X.\left\langle p_p \,\middle|\, I(\mathbf{t}_a(x)) * p(x)\right\rangle\ \mathbb{C}\ \ \exists y \in Y.\left\langle q_p(x,y) \,\middle|\, I(\mathbf{t}_a(x)) * q(x,y)\right\rangle}{\mathcal{A} \vdash_\tau \mathbb{V}x \in X.\left\langle p_p \,\middle|\, \mathbf{t}_a(x) * p(x)\right\rangle\ \mathbb{C}\ \ \exists y \in Y.\left\langle q_p(x,y) \,\middle|\, \mathbf{t}_a(x) * q(x,y)\right\rangle}$$

use atomic rule

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

update region rule

$$\frac{\mathcal{A} \vdash_\tau \begin{array}{c} \mathbb{V}x \in X.\left\langle p_p \,\middle|\, I(\mathbf{t}_a(x)) * p(x)\right\rangle \\ \mathbb{C} \\ \exists y \in Y.\left\langle q_p(x,y) \,\middle|\, \exists z \in Q(x).\, I(\mathbf{t}_a(z)) * q_1(x,y,z) \vee I(\mathbf{t}_a(x)) * q_2(x,y)\right\rangle \end{array}}{a : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash_\tau \begin{array}{c} \mathbb{V}x \in X.\left\langle p_p \,\middle|\, \mathbf{t}_a(x) * p(x) * a \Mapsto \blacklozenge\right\rangle \\ \mathbb{C} \\ \exists y \in Y.\left\langle q_p(x,y) \,\middle|\, \begin{array}{c} \exists z \in Q(x).\, \mathbf{t}_a(z) * q_1(x,y,z) * a \Mapsto (x,z) \\ \vee\, \mathbf{t}_a(x) * q_2(x,y) * a \Mapsto \blacklozenge \end{array}\right\rangle \end{array}}$$

make atomic rule

$$\frac{\begin{array}{c} a \notin \mathcal{A} \quad \{(x,y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ a : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash_\tau \\ \left\{p_p * \exists x \in X.\, \mathbf{t}_a(x) * a \Mapsto \blacklozenge\right\}\ \mathbb{C}\ \left\{\exists x \in X, y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y)\right\} \end{array}}{\mathcal{A} \vdash_\tau \mathbb{V}x \in X.\left\langle p_p \,\middle|\, \mathbf{t}_a(x) * [\mathrm{G}]_a\right\rangle\ \mathbb{C}\ \ \exists y \in Q(x).\left\langle q_p(x,y) \,\middle|\, \mathbf{t}_a(y) * [\mathrm{G}]_a\right\rangle}$$

Fig. 6: A selection of proof rules of Total-TaDA.

The use atomic rule allows us to update the abstract state of a shared region. To do so, we need a guard that permits this update. 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.

$$\forall \alpha.\ \vdash_\tau \{\mathsf{emp}\}\ \mathtt{makeStack}()\ \{\exists s \in \mathbb{T}_1, t \in \mathbb{T}_2.\, \mathsf{Stack}(s, \mathsf{ret}, [], t, \alpha)\}$$

$$\begin{array}{c} \forall \beta.\ \vdash_\tau \mathbb{V} vs, t, \alpha. \langle \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \wedge \alpha > \beta(vs, \alpha)\rangle \\ \mathtt{push}(\mathtt{x}, \mathtt{v}) \\ \langle \exists t'.\, \mathsf{Stack}(s, \mathtt{x}, \mathtt{v} : vs, t', \beta(vs, \alpha))\rangle \end{array}$$

$$\begin{array}{c} \vdash_\tau \mathbb{V} vs, t, \alpha. \quad \langle \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha)\rangle \\ \mathtt{pop}(\mathtt{x}) \\ \left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \wedge \mathsf{ret} = 0 \\ \underline{\mathbf{else}}\ \exists vs', t'.\, \mathsf{Stack}(s, \mathtt{x}, vs', t', \alpha) \wedge vs = \mathsf{ret} : vs' \end{array} \right\rangle \end{array}$$

Fig. 7: Stack operation specifications.

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 $a \Rrightarrow \blacklozenge$ 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 in the premiss; they serve to record information about the atomic update that is performed for use further down the proof tree.

Finally, we revisit the make atomic rule, which elaborates on the version presented in §2.3. 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: Treiber's Stack

We now consider a version of Treiber's stack [19] to demonstrate how Total-TaDA can be applied to verify the total correctness of larger modules.

4.1 Specification

In Fig. 7, we give the specification of the lock-free stack operations. This is a Total-TaDA specification satisfiable by a reasonable non-blocking implementation. As with the counter, the predicate representing the stack is parametrised by an ordinal that bounds the number of operations on the stack, in order to guarantee termination. The $\mathsf{Stack}(s, x, vs, t, \alpha)$ predicate has five parameters: the address of the stack x; its contents vs; an ordinal α that decreases every time a `push` operation is performed; and two parameters, s and t that range over abstract types $\mathbb{T}_1$ and $\mathbb{T}_2$ respectively. These last two parameters encapsulate implementation-specific information about the configuration of the stack (s is invariant, while t may vary) and hence their types are abstract to the client.

The constructor returns an empty stack, parametrised by an arbitrary ordinal chosen by the client. The `push` operation atomically adds an element to the head of the stack. The `pop` operation atomically removes one element from the head of the stack, if one is available (*i.e.* the stack is non-empty); otherwise it will simply return 0. (As this stack is non-blocking, it would not be possible for the pop operation to wait for the stack to become non-empty.)

Note that the ordinal parametrising the stack is not required to decrease when popping the stack. This means that the stack operations cannot be starved by an unbounded number of `pop` invocations. This need not be the case in general for a lock-free stack, but it is true for Treiber's stack. We discuss the ramifications of this kind of specification further in §6.3.

4.2 Implementation

Fig. 8 gives an implementation of the stack operations based on Treiber's stack [19]. The stack is represented as a heap cell containing a pointer (the head pointer) to a singly-linked list of the values on the stack.

Values are pushed onto the stack by allocating a new node holding the value to be pushed and a pointer to the old head of the stack. A compare-and-swap operation updates the old head of the stack to point to the new node. If the operation fails, it will be because the head of the stack has changed, and so the operation is retried.

Values are popped from the stack by moving the head pointer one step along the list. Again, a compare-and-swap operation is used for this update, so if the head of the stack changes the operation can be retried. If the stack is empty (*i.e.* the head points to 0), then pop simply returns 0, without affecting the stack.

4.3 Correctness

To prove correctness of the implementation, we introduce predicates to represent the linked list:

$$\mathsf{list}(x, ns) \triangleq (x = 0 \wedge ns = []) \vee (\exists v, l.\, \mathsf{node}(x, v, l) * \mathsf{list}(l, ns') \wedge ns = (x, v) : ns')$$
$$\mathsf{node}(n, v, l) \triangleq n.\mathtt{value} \mapsto v * n.\mathtt{next} \mapsto l$$

It is important to the correctness of the algorithm that nodes that have been popped can never reappear as the head of the stack. To account for this, in our representation of the stack we track the set of previously popped nodes, and ensure that they are disjoint from the nodes in the stack. The $\mathsf{stack}(x, ns, ds)$

```
function makeStack() {
  x := alloc(1);
  [x] := 0;
  return x;
}
```

```
function push(x, v) {
  y := alloc(2);
  [y.value] := v;
  do {
    z := [x];
    [y.next] := z;
    b := CAS(x, z, y);
  } while (b = 0);
}
```

```
function pop(x) {
  do {
    y := [x];
    if (y = 0) { return 0; }
    z := [y.next];
    b := CAS(x, y, z);
  } while (b = 0);
  v := [y.value];
  return v;
}
```

Fig. 8: Treiber's stack operations.

predicate therefore consists of a list starting at address x, with contents ns, and a disjoint set of nodes ds (the discarded nodes):

$$\mathsf{stack}(x, ns, ds) \triangleq \mathsf{list}(x, ns) * \bigcircledast_{(n,v)\in ds} \mathsf{node}(n, v, \_)$$

We define a region type **TStack** to hold the shared data-structure. The type is parametrised by the address of the stack, and its abstract state consists of a list of nodes in the stack ns, a set of popped nodes ds, and an ordinal α. The **TStack** region type has the following interpretation:

$$I(\mathbf{TStack}_r(x, ns, ds, \alpha)) \triangleq \exists y.\, x \mapsto y * \mathsf{stack}(y, ns, ds)$$

We use a single guard G to give threads permissions to push and pop the stack. The transition system is given as follows:

$$\mathrm{G} : \forall n, v, ns, ds, \alpha, \beta < \alpha.\, (ns, ds, \alpha) \rightsquigarrow ((n, v) : ns, ds, \beta)$$
$$\mathrm{G} : \forall n, v, ns, ds, \alpha.\, ((n, v) : ns, ds, \alpha) \rightsquigarrow (ns, (n, v) \uplus ds, \alpha)$$

The first action allows us to add an element to the head of the stack. The second action allows us to remove the top element of the stack, adding it to the set of discarded nodes. There is no explicit transition for the pop on the empty stack, since this operation does not change the abstract state.

Note that for every transition $(ns, ds, \alpha) \rightsquigarrow (ns', ds', \alpha')$, we have $2\cdot\alpha+|ns| > 2\cdot\alpha' + |ns'|$. Pushing decreases the ordinal, but extends the length of the stack by 1; popping maintains the ordinal, but decreases the length of the stack. This property allows us to use $2\cdot\alpha + |ns|$ as a variant in the compare-and-swap loops, since it is guaranteed to decrease under any interference.

The abstract predicate $\mathsf{Stack}(s, x, vs, t, \alpha)$ combines the region and the guard:

$$\mathsf{Stack}(r, x, vs, (ns, ds), \alpha) \triangleq \mathbf{TStack}_r(x, ns, ds, \alpha) * [\mathrm{G}]_r \land vs = \mathsf{snds}(ns)$$

The function snds returns the list of elements of the second elements of the list of pairs ns. Consequently, vs is the list of values on the stack, rather than pairs of address and value.

abstract; substitute $s = r$, $t = (ns, ds)$

make atomic

update region

$$
\begin{array}{l}
\forall\!\!\!\forall vs, t, \alpha.\\
\left\langle \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \right\rangle\\
\quad \left\langle \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * [\mathrm{G}]_r \land vs = \mathsf{snds}(ns) \right\rangle\\
\qquad r : (ns, ds, \alpha)\\
\qquad\quad \rightsquigarrow \underline{\mathbf{if}}\ ns = []\ \underline{\mathbf{then}}\ (ns, ds, \alpha)\ \underline{\mathbf{else}}\ (ns', (n, v) \uplus ds, \alpha) \land ns = (n, v) : ns' \ \vdash_\tau\\
\qquad \left\{ \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \right\}\\
\qquad \mathtt{do}\ \{\\
\qquad\quad \forall \gamma.\\
\qquad\quad \left\{ \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land \gamma \geq 2 \cdot \alpha + |ns| \right\}\\
\qquad\quad\quad \forall\!\!\!\forall ns, ds, \alpha.\ \left\langle \exists w.\, \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) \land \gamma \geq 2 \cdot \alpha + |ns| \right\rangle\\
\qquad\quad\quad\quad \mathtt{y} := [\mathtt{x}];\\
\qquad\quad\quad\quad \left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{y} * \mathsf{stack}(\mathtt{y}, ns, ds) \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ \underline{\mathbf{if}}\ \mathtt{y} = 0\ \underline{\mathbf{then}}\ ns = []\ \underline{\mathbf{else}}\ \exists v.\, (\mathtt{y}, v) = \mathsf{head}(ns) \end{array} \right\rangle\\
\qquad\quad \left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \underline{\mathbf{if}}\ \mathtt{y} = 0\ \underline{\mathbf{then}}\ r \Rrightarrow (([], ds, \alpha), ([], ds, \alpha)) \\ \quad \underline{\mathbf{else}}\ \exists v. \left(\begin{array}{l} \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land (\mathtt{y}, v) \in ns +\!\!+\, ds \land \\ \gamma \geq 2 \cdot \alpha + |ns| \land \mathsf{head}(ns) \neq (\mathtt{y}, v) \Rightarrow \gamma > 2 \cdot \alpha + |ns| \end{array} \right) \end{array} \right\}\\
\qquad\quad \mathtt{if}\ (\mathtt{y} = 0)\ \{\\
\qquad\quad\quad \mathtt{return}\ 0; \quad \left\{ \exists ds, \alpha.\, r \Rrightarrow (([], ds, \alpha), ([], ds, \alpha)) \land \mathsf{ret} = 0 \right\}\\
\qquad\quad \}\\
\qquad\quad \left\{ \begin{array}{l} \exists ns, ds, v, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land (\mathtt{y}, v) \in ns +\!\!+\, ds \land \\ \gamma \geq 2 \cdot \alpha + |ns| \land \mathsf{head}(ns) \neq (\mathtt{y}, v) \Rightarrow \gamma > 2 \cdot \alpha + |ns| \end{array} \right\}\\
\qquad\quad \mathtt{z} := [\mathtt{y.next}];\\
\qquad\quad \left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ \left(\begin{array}{l} (\exists v, v', ns'.\, ns = [(\mathtt{y}, v), (\mathtt{z}, v')] +\!\!+\, ns') \lor (\exists v.\, ns = [(\mathtt{y}, v)] \land \mathtt{z} = 0) \\ \lor (\exists v.\, (\mathtt{y}, v) \in ns +\!\!+\, ds \land \mathsf{head}(ns) \neq (\mathtt{y}, v) \land \gamma > 2 \cdot \alpha + |ns|) \end{array} \right) \end{array} \right\}\\
\qquad\quad\quad \forall\!\!\!\forall ns, ds, \alpha.\\
\qquad\quad\quad \left\langle \begin{array}{l} \exists w.\, \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ \left(\begin{array}{l} (\exists v, v', ns'.\, ns = [(\mathtt{y}, v), (\mathtt{z}, v')] +\!\!+\, ns') \\ \lor (\exists v.\, ns = [(\mathtt{y}, v)] \land \mathtt{z} = 0) \\ \lor (\exists v.\, (\mathtt{y}, v) \in ns +\!\!+\, ds \land \mathsf{head}(ns) \neq (\mathtt{y}, v) \land \gamma > 2 \cdot \alpha + |ns|) \end{array} \right) \end{array} \right\rangle\\
\qquad\quad\quad \mathtt{b} := \mathtt{CAS}(\mathtt{x}, \mathtt{y}, \mathtt{z});\\
\qquad\quad\quad \left\langle \begin{array}{l} \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \exists w.\, \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) \land \gamma > 2 \cdot \alpha + |ns| \\ \quad \underline{\mathbf{else}}\ \exists v, ns'.\, \mathtt{x} \mapsto \mathtt{z} * \mathsf{stack}(\mathtt{z}, ns', (\mathtt{y}, v) \uplus ds) \land ns = (\mathtt{y}, v) : ns' \end{array} \right\rangle\\
\qquad\quad \left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \gamma \geq 2 \cdot \alpha + |ns| \land \\ \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land \gamma > 2 \cdot \alpha + |ns| \\ \quad \underline{\mathbf{else}} \left(\begin{array}{l} \exists v, ns', ds', \alpha'.\, (\mathtt{y}, v) \in ds' \land \mathbf{TStack}_r(\mathtt{x}, ns', ds', \alpha') \\ \quad * r \Rrightarrow (((\mathtt{y}, v) : ns, ds, \alpha), (ns, (\mathtt{y}, v) \uplus ds), \alpha) \end{array} \right) \end{array} \right\}\\
\qquad \}\ \mathtt{while}\ (\mathtt{b} = 0);\\
\qquad \left\{ \begin{array}{l} \exists v, ns, ds, \alpha, ns', ds', \alpha'.\, (\mathtt{y}, v) \in ds' \land \mathbf{TStack}_r(\mathtt{x}, ns', ds', \alpha') \\ * r \Rrightarrow (((\mathtt{y}, v) : ns, ds, \alpha), (ns, (\mathtt{y}, v) \uplus ds, \alpha)) \end{array} \right\}\\
\qquad \mathtt{v} := [\mathtt{y.value}]; \quad \left\{ \exists ns, ds, \alpha.\, r \Rrightarrow (((\mathtt{y}, \mathtt{v}) : ns, ds, \alpha), (ns, (\mathtt{y}, \mathtt{v}) \uplus ds, \alpha)) \right\}\\
\qquad \mathtt{return}\ \mathtt{v}; \quad \left\{ \exists y, ns, ds, \alpha.\, r \Rrightarrow (((y, \mathsf{ret}) : ns, ds, \alpha), (ns, (y, \mathsf{ret}) \uplus ds), \alpha) \right\}\\
\quad \left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * [\mathrm{G}]_r \land vs = \mathsf{snds}(ns) \land \mathsf{ret} = 0 \\ \quad \underline{\mathbf{else}} \left(\begin{array}{l} \exists ns', vs', y.\, \mathbf{TStack}_r(\mathtt{x}, ns', (y, \mathsf{ret}) \uplus ds, \alpha) * [\mathrm{G}]_r \\ \quad \land vs' = \mathsf{snds}(ns') \land ns = (y, \mathsf{ret}) : ns' \end{array} \right) \end{array} \right\rangle\\
\left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \land \mathsf{ret} = 0 \\ \quad \underline{\mathbf{else}}\ \exists vs', t'.\, \mathsf{Stack}(s, \mathtt{x}, vs', t', \alpha) \land vs = \mathsf{ret} : vs' \end{array} \right\rangle
\end{array}
$$

Fig. 9: Proof of total correctness of Treiber's stack pop operation.

The proof for `pop` is given in Fig. 9. When the stack is non-empty, if the compare-and-swap fails then another thread must have succeeded in updating the stack, and so reduced the ordinal or the length of the stack; by basing the loop variant on the ordinal and stack length, we can guarantee that the operation will eventually succeed. The proof for `push` is given in appendix B.

5 Soundness

The proof of soundness of Total-TaDA is similar to that for TaDA [16] and based on the Views Framework [4]. We use the same model for assertions as that for TaDA. We also use a similar semantic judgement, $\vDash$, which ensures that the concrete behaviours of programs simulate the abstract behaviours represented by the specifications. The key distinction is that, whereas in TaDA the judgement is defined coinductively (as a greatest fixed point), in Total-TaDA the judgement is defined inductively (as a least fixed point). This means that TaDA admits executions that never terminate, while Total-TaDA requires executions to always terminate: that is, reach a base-case of the inductive definition.

The soundness proof consists of lemmas that justify each of the proof rules for the semantic judgement. Most of the Total-TaDA rules have similar proofs to the corresponding TaDA rules, but proceed by induction instead of coinduction. Of course, the while rule is different, since termination does not follow trivially. We sketch the proof for while. All details are in appendix A.

Lemma 1 (While Rule). *Let α be an ordinal. If, for all $\gamma \leq \alpha$,*

$$\mathcal{A} \vDash_\tau \left\{p(\gamma) \wedge \mathbb{B}\right\}\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \beta < \gamma\right\}, \text{ then} \tag{1}$$

$$\mathcal{A} \vDash_\tau \left\{p(\alpha)\right\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha\right\}. \tag{2}$$

Proof. The proof is by transfinite induction on α. As the inductive hypothesis (IH), assume that the lemma holds for all $\delta < \alpha$. The program `while` $(\mathbb{B})$ $\mathbb{C}$ has two possible reductions, which do not affect the state, depending on the truth value of the loop test. Consequently, to show (2), it is sufficient to establish:

$$\mathcal{A} \vDash_\tau \left\{p(\alpha) \wedge \mathbb{B}\right\}\ \mathbb{C};\texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha\right\} \tag{3}$$

$$\mathcal{A} \vDash_\tau \left\{p(\alpha) \wedge \neg\mathbb{B}\right\}\ \texttt{skip}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha\right\} \tag{4}$$

(4) holds trivially. To establish (3), (1) gives $\mathcal{A} \vDash_\tau \left\{p(\alpha) \wedge \mathbb{B}\right\}\ \mathbb{C}\ \left\{\exists \delta.\, p(\delta) \wedge \delta < \alpha\right\}$. For all $\delta < \alpha$, IH gives $\mathcal{A} \vDash_\tau \left\{p(\delta)\right\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \delta\right\}$, and hence $\mathcal{A} \vDash_\tau \left\{\exists \delta.\, p(\delta) \wedge \delta < \alpha\right\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \left\{\exists \beta.\, p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha\right\}$. Now (3) follows using the analogous sequential composition lemma in appendix A. ☐

6 Non-blocking Properties

Non-blocking properties are used to characterise concurrent algorithms that guarantee progress. A *lock-free* algorithm guarantees global progress: an individual thread might fail to make progress, but only because some other thread does make progress. A *wait-free* algorithm guarantees local progress: every thread makes progress when it is scheduled. We consider how non-blocking properties can be formalised using Total-TaDA.

6.1 Lock-freedom

We have described lock-freedom in terms of an informal notion of "progress". In order to properly characterise modules as lock-free, we need a more formal definition. We can characterise global progress for a module as follows: at any time, eventually either a pending operation will be completed or another operation will be begun. If we assume that the number of threads is bounded, then as long as there are pending module operations, some operation will eventually complete. (If the number of threads is unbounded, then there is no guarantee that any operation will complete, even if it is scheduled arbitrarily often, since additional operations can always begin.)

Based on this observation, Gotsman *et al.* [7] reduced lock-freedom to the termination of a simple class of programs, the bounded most-general clients (BMGCs) of a module. Hoffmann *et al.* [10] generalised the result to apply to algorithms where the identity or number of threads is significant. An (m, n)-bounded general client consists of m threads which each invoke n module operations in sequence. If all such bounded general clients (for every n and m)[5] terminate, then the module is lock free.

Definition 1. *Consider a module $\mathcal{M}$ with initialiser* `init` *and a set of operations O. Define the following sets of programs:*

$$T_n = \{\texttt{op}_1; \ldots ; \texttt{op}_n \mid \texttt{op}_i \in O\} \qquad C_{m,n} = \{\texttt{init}; (t_1 \| \ldots \| t_m) \mid t_i \in T_n\} .$$

Theorem 1 (Hoffmann *et al.* [10]). *Given a module $\mathcal{M}$, if, for all m and n, every program $c \in C_{m,n}$ terminates, then $\mathcal{M}$ is lock free.*

Using this theorem, we define a specification pattern for Total-TaDA that guarantees lock-freedom and follows easily from the typical specifications we establish for lock-free modules.

Theorem 2. *Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract parameters and an ordinal parameter), suppose that the following specifications are provable:*

$$\forall \alpha. \vdash_\tau \left\{\mathsf{true}\right\} \texttt{init} \left\{\exists s, u. \mathsf{M}(s, u, \alpha)\right\}$$
$$\forall \texttt{op} \in O. \forall \beta. \vdash_\tau \mathbb{V}\alpha, u. \left\langle \mathsf{M}(s, u, \alpha) \wedge \alpha > \beta(\alpha) \right\rangle \texttt{ op } \left\langle \exists u'. \mathsf{M}(s, u', \beta(\alpha)) \right\rangle.$$

Then $\mathcal{M}$ is lock-free.

Proof. By Theorem 1, it is sufficient to show that, for arbitrary m, n and $c \in C_{m,n}$, the program c terminates. Fix the number of threads m.

We define a region type $\mathbf{M}$ whose abstract states consist of vectors $\bar{x} \in \mathbb{N}^m$. (We denote by x_i, for $1 \leq i \leq m$, the i-th component of vector $\bar{x}$. We denote by $\sum \bar{x}$ the sum $\sum_{i=1}^{i=m} x_i$.) Region states are interpreted as follows: $I(\mathbf{M}_a(s, \bar{x})) \triangleq \exists u. \mathsf{M}\,(s, u, \sum \bar{x})$. The guard algebra for $\mathbf{M}$ consists of m distinct

[5] The bounded *most-general* client may be seen as the program which non-deterministically chooses among all bounded general clients.

guards $\mathrm{G}_1, \ldots, \mathrm{G}_m$. The state transition system for $\mathbf{M}$ allows a thread holding guard G_i to decrease the i-th component of the abstract state:

$$\mathrm{G}_i : (\forall j \neq i.\, x_j = y_j) \wedge x_i > y_i \wedge \bar{x} \rightsquigarrow \bar{y}.$$

For $1 \leq i \leq m$, arbitrary n, and $\mathsf{op} \in O$, using the $\mathsf{use\ atomic}$ rule, we have

$$\frac{\mathbb{V}k, u. \big\langle \mathsf{M}(s, u, k+n+1) \big\rangle \; \mathsf{op} \; \big\langle \exists u'.\, \mathsf{M}(s, u', k+n) \big\rangle}{\big\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a \wedge x_i = n+1\big\} \; \mathsf{op} \; \big\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a \wedge x_i = n\big\}}$$

Applying this specification repeatedly (by induction), we have for arbitrary $t \in T_n$

$$\vdash_\tau \big\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a \wedge x_i = n\big\} \; t \; \big\{\mathsf{true}\big\}$$

Let $c = \mathtt{init}; (t_1 \| \ldots \| t_m) \in C_{m,n}$ be arbitrary. We derive $\vdash_\tau \{\mathsf{true}\}\; c\; \{\mathsf{true}\}$ easily by choosing $n \cdot m$ as the initial ordinal and creating an $\mathbf{M}$-region with initial state $(n, \ldots, n)$. Consequently, c terminates, as required. □

It is straightforward to apply Theorem 2 to the modules we have considered.

6.2 Wait-freedom

Whereas lock-freedom only requires that *some* thread makes progress, wait-freedom requires that *every* thread makes progress (provided that it is not permanently descheduled). In terms of operations, this requires that each operation of a module should complete within a finite number of steps. Since Total-TaDA specifications guarantee that operations terminate, it is simple to describe a specification that implies that a module is wait-free.

Theorem 3. *Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract parameters), suppose that the following specifications are provable:*

$$\vdash_\tau \big\{\mathsf{true}\big\} \; \mathtt{init} \; \big\{\exists s, t.\, \mathsf{M}(s, u)\big\} \qquad \forall \mathsf{op} \in O.\ \vdash_\tau \mathbb{V}u. \big\langle \mathsf{M}(s, u) \big\rangle \; \mathsf{op} \; \big\langle \exists u'.\, \mathsf{M}(s, u') \big\rangle.$$

Then $\mathcal{M}$ is wait-free.

Proof. The specifications imply that M is an invariant which is established by the initialiser and preserved at all times by the module operations. Furthermore, all of the module operations terminate, assuming the environment maintains M invariant. Consequently, all of the module operations terminate in the context of an environment calling module operations: the module is wait-free. □

Lock-freedom can only be applied to a module as a whole, since it relates to global progress. Wait-freedom, by contrast, relates to local progress — that the operations of *each* thread terminate — and so it is meaningful to consider an individual operation to be wait-free in a context where other operations may be lock-free or even blocking. By combining (partial-correctness) TaDA and Total-TaDA specifications (indicated by $\vdash$ and $\vdash_\tau$ respectively), we can give a specification pattern that guarantees wait-freedom for a specific module operation.

Theorem 4. *Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract parameters), suppose that the following specifications are provable:*

$$\vdash \{\mathsf{true}\}\ \mathtt{init}\ \{\exists s, u.\, \mathsf{M}(s,u)\} \qquad \vdash_\tau \mathbb{V}u.\langle \mathsf{M}(s,u)\rangle\ \mathtt{op}\ \langle \exists u'.\, \mathsf{M}(s,u')\rangle$$
$$\forall \mathtt{op}' \in O.\ \vdash \mathbb{V}u.\langle \mathsf{M}(s,u)\rangle\ \mathtt{op}'\ \langle \exists u'.\, \mathsf{M}(s,u')\rangle$$

Then `op` *is wait-free.*

Proof. As before, M is a module invariant; `op` is guaranteed to terminate with this invariant, therefore it is wait-free. ☐

The specifications required by Theorem 4 do not follow from those given for our examples. However, where applicable, the proofs can easily be adapted. For instance, to show that the `read` operation of the counter is wait-free, we would remove the ordinals from the region definition, and abstract the value of the counter. This breaks the termination proof for the increment operations, but we can adapt it to a partial-correctness proof in TaDA. The termination proof for `read` does not depend on the ordinal parameter of the region, and so we can still establish total correctness, as required.

6.3 Non-impedance

Recall the counter specification from §2.1. If we abstract the value and address of the counter (which are irrelevant to termination), the specification becomes:

$$\forall \alpha.\ \vdash_\tau \{\mathsf{emp}\}\ \mathtt{x} := \mathtt{makeCounter}()\ \{\exists s \in \mathbb{T}_1, u \in \mathbb{T}_2.\, \mathsf{C}(s,u,\alpha)\}$$
$$\vdash_\tau \mathbb{V}u,\alpha.\langle \mathsf{C}(s,u,\alpha)\rangle\ \mathtt{read}(\mathtt{x})\ \langle \mathsf{C}(s,u,\alpha)\rangle$$
$$\forall \beta.\ \vdash_\tau \mathbb{V}u,\alpha.\langle \mathsf{C}(s,u,\alpha) \land \alpha > \beta(\alpha)\rangle\ \mathtt{incr}(\mathtt{x})\ \langle \exists u'.\, \mathsf{C}(s,u',\beta(\alpha))\rangle$$

Since the `read` operation does not change the ordinal, it implies that both the `read` and `incr` operations will terminate in a concurrent environment that performs an unbounded number of `read`s. This suggests an alternative approach to characterising lock-free modules in terms of which operations *impede* each other — that is, which operations may prevent the termination of an operation if infinitely many of them are invoked during a (fair) execution of the operation. Our specification implies that `read` does not impede either `read` or `incr`. This is expressed by edges 1 and 2 in the following non-impedance graph:

$$\begin{tikzcd} \mathtt{incr} \arrow[r, bend left, "3"] & \mathtt{read} \arrow[l, bend left, "2"] \arrow[loop right, "1"] \end{tikzcd}$$

Note that the above specifications for the counter do not by themselves imply that `incr` does not impede `read` (edge 3). This can be demonstrated by considering an alternative implementation of `read`, that satisfies the specification but is not wait-free:

```
do {
  v := [x]; w := [x];
} while (v ≠ w);
return v;
```

Recall that we can prove that `read` is wait-free by giving a different specification as in Theorem 4. An operation is wait-free exactly when every operation does not impede it. For `read`, this is expressed by edges 1 and 3 in the above graph.

The stack specification in Fig. 7, much like the counter specification, implies that `pop` does not impede either `push` or `pop`:

$$\begin{tikzcd} \texttt{push} & \texttt{pop} \arrow[l] \arrow[loop right] \end{tikzcd}$$

The `pop` operation, however, may be impeded by `push`.

The non-impedance relationships implied by the stack specification are important for clients. For instance, consider a producer-consumer scenario in which the stack is used to communicate data from producers to consumers. When no data is available, consumers may simply loop attempting to pop the stack. If the `pop` operation could impede `push`, then producers might be starved by consumers. In this situation, we could not guarantee that the system would make progress. This suggests that non-impedance, which is captured by Total-TaDA specifications, can be an important property of non-blocking algorithms.

7 Related Work

Hoffmann *et al.* [10] introduced a concurrent separation logic for verifying total correctness. By adapting the most-general-client approach of Gotsman *et al.* [7], they establish that modules are lock-free. (They do not, however, establish functional correctness.) This method involves a thread passing "tokens" to other threads whose lock-free operations are impeded by modifications to the shared state. Subsequent approaches [1,12] also use some form of tokens that are used up in loops or function calls. These approaches require special proof rules for the tokens. When these approaches restrict to dealing with finite numbers of tokens, support for unbounded non-determinism (as in the backoff increment example of Fig. 5) is limited. In Total-TaDA such token passing is not necessary. Instead, we require the client to provide a general (ordinal) limit on the amount of impeding interference. Consequently, we can guarantee the termination of loops with standard proof rules.

Liang *et al.* [12] have developed a proof theory for termination-preserving refinement, applying it to verify linearisability and lock-freedom. Their approach constrains impedance by requiring that impeding actions correspond to progress at the abstract level. In Total-TaDA, such constraints are made by requiring that impeding actions decrease an ordinal associated with a shared region. Their approach does not freely combine lock-free and wait-free specifications whereas, with Total-TaDA, we can reason about lock- and wait-freedom in combination, and more subtle conditions such as non-impedance. For example, we can show when a read operation of a lock-free data-structure is wait-free. Their specifications establish termination-preserving refinement (given a context, if the abstract program is guaranteed to terminate, then so is the concrete), whereas Total-TaDA specifications establish termination (in a context, the program will terminate).

Name	Appendix
Treiber's Stack (continued)	B
Linked List Map	C
Hash Map	D
Hash Set	E

Table 1: Additional examples.

Boström and Müller [1] have introduced an approach that can verify termination and progress properties of concurrent programs. The approach supports blocking concurrency and non-terminating programs, which Total-TaDA does not. However, the approach does not aim at racy concurrent programs and cannot deal with any of the examples shown in the paper. Furthermore, the relationship between termination and lock- and wait-freedom is not considered.

Of the above approaches, none covers total functional correctness for fine-grained concurrent programs. With Total-TaDA we can reason about clients that use modules, without their implementation details. Moreover, with Total-TaDA it is easy to verify module operations independently, with respect to a common abstraction, rather than considering a whole module at once. Finally, our approach to specification is unique in supporting lock- and wait-freedom simultaneously, as well as expressing more subtle conditions such as non-impedance.

8 Conclusions and Future Work

We have introduced Total-TaDA, a program logic that provides local, modular reasoning for proving the termination and functional correctness of non-blocking concurrent programs. With our abstract specifications, clients can reason about total correctness without needing to know about the underlying implementation. Different implementations, satisfying the same specification, can have different termination arguments, but these arguments are not exposed to the clients. By using ordinals to bound interference, our specifications can express traditional non-blocking properties. Moreover, they capture a new notion of *non-impedance*: that one operation does not set back the progress of another.

We have claimed that our approach supports modular reasoning, and substantiated this claim by reasoning about implementations and clients of modules. In the appendices, we provide further examples, listed in Table 1. In particular, we specify a non-blocking map and verify two implementations, based on lists and hash tables, with the second making use of the first through the abstract specification. We also implement a set specification on top of the map.

Blocking. Many concurrent modules make use of *blocking*, for example by using semaphores or monitors. Properties such as starvation-freedom can be expressed in terms of termination, but require the assumption of a fair scheduler. Some aspects of our approach are likely to apply here. However, it is also necessary to constrain future behaviours, for instance, to specify that a lock that has been acquired will be released in a finite time. This might be achieved with a program logic that can reason explicitly about continuations.

Non-termination. Some programs, such as operating systems, are designed not to terminate. Such programs should still continually perform useful work. It would be interesting to extend Total-TaDA to specify and verify progress properties of non-terminating systems. Progress can be seen as localised termination, so the same reasoning techniques should apply. However, a different approach to specification will be necessary to express and verify these properties.

Acknowledgements. We thank Bart Jacobs, Hongjin Liang, Peter Müller and the anonymous referees for useful feedback. This research was supported by EPSRC Programme Grants EP/H008373/1 and EP/K008528/1, by the "Modu-Res" Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU) and the "Automated Verification for Concurrent Programs" Individual Postdoc Grant from The Danish Council for Independent Research for Technology and Production Sciences (FTP).

References

1. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. In: 29th European Conference on Object-Oriented Programming. vol. 37, pp. 639–663. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
2. Boyland, J.: Checking interference with fractional permissions. In: Static Analysis, Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer Berlin Heidelberg (2003)
3. Cantor, G.: Beiträge zur begründung der transfiniten mengenlehre. Mathematische Annalen 49(2), 207–246 (1897), `http://dx.doi.org/10.1007/BF01444205`
4. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300 (2013)
5. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
6. Floyd, R.W.: Assigning Meanings to Programs. Proceedings of the American Mathematical Society Symposia on Applied Mathematics 19, 19–31 (1967)
7. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving That Non-blocking Algorithms Don't Block. In: POPL. pp. 16–28 (2009)
8. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
9. Hessenberg, G.: Grundbegriffe der Mengenlehre. Abhandlungen der Fries'schen Schule / Neue Folge, Vandenhoeck & Ruprecht (1906)
10. Hoffmann, J., Marmar, M., Shao, Z.: Quantitative reasoning for proving lock-freedom. In: Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on. pp. 124–133. IEEE (2013)
11. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. pp. 637–650 (2015)
12. Liang, H., Feng, X., Shao, Z.: Compositional verification of termination-preserving refinement of concurrent programs. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). p. 65. ACM (2014)

13. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. SIGOPS Oper. Syst. Rev. pp. 108– (1992)
14. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL. pp. 247–258 (2005)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
16. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: ECOOP 2014–Object-Oriented Programming, pp. 207–231. Springer Berlin Heidelberg (2014)
17. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Steps in modular specifications for concurrent modules (invited tutorial paper). Electronic Notes in Theoretical Computer Science 319, 3–18 (2015)
18. Svendsen, K., Birkedal, L.: Impredicative Concurrent Abstract Predicates. In: ESOP. pp. 149–168 (2014)
19. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center (April 1986)
20. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines. pp. 67–69 (1949), http://www.turingarchive.org/browse.php/B/8
21. 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)

A Semantics

A.1 Operational Semantics

The operational semantics of our language are given in Fig. 10 and Fig. 11.

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

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

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

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

$$\frac{\tau \xrightarrow{a} \tau'}{\langle s, \texttt{x} := \tau \rangle \xrightarrow{a} \langle s, \texttt{x} := \tau' \rangle}$$

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

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

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

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

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

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

$$\frac{}{\langle s, \texttt{fork } f(\overrightarrow{\mathbb{E}}) \rangle \xrightarrow{\mathsf{spawn}(f, \mathcal{E}[\![\overrightarrow{\mathbb{E}}]\!]_s)} \langle s, \texttt{skip} \rangle}$$

Fig. 10: Small-step operational semantics for threads, $\xrightarrow{a}_\gamma$. The parameter γ is fixed, and not shown.

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

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

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

Fig. 11: Small-step operational semantics for thread pools, $\xrightarrow{a}_{\gamma}$.

A.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 [4]) 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.

Heaps. We assume a set Val of program values, which includes a set $\mathsf{Loc} \subseteq \mathsf{Val}$ of program locations. A heap $h \in \mathsf{Heap} \stackrel{\text{def}}{=} \mathsf{Loc} \rightharpoonup_{fin} \mathsf{Val}$ 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 $b \in \mathsf{APBag} \stackrel{\text{def}}{=} \mathcal{M}_{fin}(\mathsf{APName} \times \mathsf{Val}^*)$ 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.

Region Assignments. We assume a (countably infinite) set of region identifiers, RId. A region assignment $r \in \mathsf{RAss} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{fin} \mathsf{Level} \times \mathsf{RTName} \times \mathsf{Val}^*$ 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:

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

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

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

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: $\rho_1 \leq \rho_2 \stackrel{\text{def}}{\iff} \forall a \in \mathrm{dom}(\rho_1).\, \rho_1(a) = \rho_2(a)$.

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 $\mathcal{R} \subseteq_{\text{fin}} \mathsf{RId}$, 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 $\mathcal{A} \in \mathsf{AContext} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{\mathit{fin}} \mathsf{AState} \rightharpoonup \mathcal{P}(\mathsf{AState})$ is a (finite) partial mapping from region identifiers to partial, non-deterministic abstract state transformers. 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:

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

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

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 $g \# g'$.) 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 $\blacklozenge$ 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 [4]) 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

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

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.[6]

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: $p{\downarrow_\lambda} \overset{\text{def}}{=} \bigcup_{\varphi \in p} \varphi{\downarrow_\lambda}$.

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

$$(r, h, b, \gamma, \rho, d){\downharpoonright_1} \overset{\text{def}}{=} \left\{ (r, h, \emptyset, \gamma, \rho, d) \cdot (w, \emptyset) \;\middle|\; w \in \circledast_{a \in b} \iota(a) \right\}$$

[6] Here, we have avoided having region interpretations directly referring to region interpretations. Impredicative CAP [18] 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.

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

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

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

Note 1. 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:

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

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:

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

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 2. 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!

Semantic Judgements In the Views Framework [4], 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 2 (Primitive Atomic Satisfaction Judgement). *The primitive atomic satisfaction judgement* $\lambda; \mathcal{A} \vDash \langle p_p \mid p \rangle\ a\ \langle p'_p \mid - \rangle + \langle q_p \mid q \rangle$*, where* $\lambda \in \mathsf{Level}$*,* $\mathcal{A} \in \mathsf{AContext}$*,* $a \in \mathsf{AAction}$ *and* $p_p, p, p'_p, q, q_p \in \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$*, is defined as:*

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

Definition 3 (Primitive Atomic Satisfaction Judgement).

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

Definition 4 (Semantic Judgement). *The semantic judgement*

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

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 least-general judgement that holds when the following conditions hold:

– *For all* $s, s' \in \mathsf{Store}$, $\mathbb{C}' \in \mathsf{Command}$, $a \in \mathsf{AAction}$ *with* $\langle \mathbb{C}, s\rangle \xrightarrow{a} \langle \mathbb{C}', s'\rangle$, *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*

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

– *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* $s_f(\mathit{vars}(\gamma(f))) = \overrightarrow{v}$,

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

– *If* $\mathbb{C} = \mathtt{skip}$ *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* $\mathbb{C} = \mathtt{return}\ \mathbb{E}; \mathbb{C}'$ *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.

A.3 Soundness

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

Lemma 2 (While Rule). *Let α be an ordinal. If, for all $\gamma \leq \alpha$,*

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\gamma) \land \mathbb{B}\right\} \ \mathbb{C} \ \left\{\exists \beta.\, p_p(\beta) \land \beta < \gamma\right\} \tag{5}$$

then

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\alpha)\right\} \ \texttt{while}\ (\mathbb{B})\ \mathbb{C} \ \left\{\exists \beta.\, p_p(\beta) \land \neg\mathbb{B} \land \beta \leq \alpha\right\}. \tag{6}$$

Proof. The proof is by transfinite induction on the ordinal α. As the inductive hypothesis, we shall assume that the lemma holds for all $\delta < \alpha$. Since $\texttt{while}\ (\mathbb{B})\ \mathbb{C}$ has two possible reductions, both with transition id, to show (6), it is sufficient to establish:

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\alpha) \land \mathbb{B}\right\} \ \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C} \ \left\{\exists \beta.\, p_p(\beta) \land \neg\mathbb{B} \land \beta \leq \alpha\right\} \tag{7}$$

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\alpha) \land \neg\mathbb{B}\right\} \ \texttt{skip} \ \left\{\exists \beta.\, p_p(\beta) \land \neg\mathbb{B} \land \beta \leq \alpha\right\} \tag{8}$$

(This is since the first condition of Definition 4 is the only one that may apply. For the reduction $\langle s, \texttt{while}\ (\mathbb{B})\ \mathbb{C}\rangle \xrightarrow{\mathsf{id}} \langle s, \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C}\rangle$ (which requires $\mathbb{B}(s)$), take $p'_p = p_p(\alpha) \land \mathbb{B}$ and $p''_p = \mathsf{false}$. The first and third sub-conditions become trivial, while the second reduces to (7). For the reduction $\langle s, \texttt{while}\ (\mathbb{B})\ \mathbb{C}\rangle \xrightarrow{\mathsf{id}} \langle s, \texttt{skip}\rangle$ (which requires $\neg\mathbb{B}(s)$), take $p'_p = p_p(\alpha) \land \neg\mathbb{B}$ and $p''_p = \mathsf{false}$. Similarly, the first and third sub-conditions are trivial and the second reduces to (8).)

To establish (7), we have from (5)

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\alpha) \land \mathbb{B}\right\} \ \mathbb{C} \ \left\{\exists \delta.\, p_p(\delta) \land \delta < \alpha\right\}.$$

By the inductive hypothesis, we have, for all $\delta < \alpha$

$$\lambda; \mathcal{A} \vDash_\tau \left\{p_p(\delta)\right\} \ \texttt{while}\ (\mathbb{B})\ \mathbb{C} \ \left\{\exists \beta.\, p_p(\beta) \land \neg\mathbb{B} \land \beta \leq \delta\right\},$$

and hence

$$\lambda; \mathcal{A} \vDash_\tau \left\{\exists \delta.\, p_p(\delta) \land \delta < \alpha\right\} \ \texttt{while}\ (\mathbb{B})\ \mathbb{C} \ \left\{\exists \beta.\, p_p(\beta) \land \neg\mathbb{B} \land \beta \leq \alpha\right\}.$$

Now (5) follows from the above by the sequencing lemma.

It is trivial to establish (8) by choosing α as the witness for β.

Lemma 3 (Recursion Rule). *If*

$$\lambda; \mathcal{A} \vdash_\tau \left\{p(\mathtt{x}, \alpha)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathtt{y})\right\}$$

then

$$\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{p(\mathtt{x}, \alpha)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathtt{y})\right\}$$

Proof. By transfinite induction, we prove $\forall \beta.\ P(\beta)$, where:

$$\begin{aligned} P(\beta) \equiv \alpha \geq \beta \Rightarrow \lambda; \mathcal{A} \vdash_\tau & \left\{p(\mathtt{x}, \beta)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathsf{ret})\right\} \\ \wedge\, \lambda; \mathcal{A} \vdash_\tau & \left\{p(\mathit{vars}(f), \beta)\right\} \; \mathit{code}(f) \; \left\{q(\mathit{vars}(f), \mathsf{ret})\right\} \end{aligned}$$

Assume by transfinite induction $\forall \gamma < \beta.\ P(\gamma)$. Assume $\alpha \geq \beta$.

Clearly, $\forall \gamma < \beta.\ P(\gamma)$ implies $\forall \gamma < \beta.\ \lambda; \mathcal{A} \vdash_\tau \{p(x, \gamma)\} \; y := f(x) \; \{q(x, y)\}$ and by our premiss, we have:

$$\begin{gathered} (\forall \gamma < \beta.\ \lambda; \mathcal{A} \vdash_\tau \{p(x, \gamma)\} \; y := f(x) \; \{q(x, y)\}), \\ \lambda; \mathcal{A} \vdash_\tau \left\{p(\mathit{vars}(f), \beta)\right\} \; \mathit{code}(f) \; \left\{q(\mathit{vars}(f), \mathsf{ret})\right\} \end{gathered}$$

So, $\lambda; \mathcal{A} \vdash_\tau \left\{p(\mathit{vars}(f), \beta)\right\} \; \mathit{code}(f) \; \left\{q(\mathit{vars}(f), \mathsf{ret})\right\}$ holds, as the above implies such a proof under the assumption that proofs exists for $\lambda; \mathcal{A} \vdash_\tau \{p(x, \gamma)\} \; y := f(x) \; \{q(x, y)\}$ for all $\gamma < \beta$. Clearly, this implies

$$\lambda; \mathcal{A} \vdash_\tau \left\{p(\mathtt{x}, \beta)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathtt{y})\right\}.$$

From this we have $P(\beta)$. Therefore, by transfinite induction, $\forall \gamma.\ P(\gamma)$.

To prove $\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{p(\mathtt{x}, \alpha)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathtt{y})\right\}$, as we can execute one step of the operational semantics, $\langle \mathtt{y} := f(\mathtt{x}), s\rangle \xrightarrow{\mathsf{id}} \langle \mathtt{y} := \langle \mathit{code}(\gamma(f)), s'\rangle, s\rangle$, with $s'(\mathit{vars}(\gamma(f))) = \mathcal{E}[\![\overrightarrow{\mathbb{E}}]\!]_s$, that is not a fork, we need to prove:

$$\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{p(\mathtt{x}, \alpha)\right\} \; \mathtt{y} := \langle \mathit{code}(\gamma(f)), s'\rangle \; \left\{q(\mathtt{x}, \mathtt{y})\right\}$$

From the above transfinite induction, $P(\alpha)$ gives us

$$\lambda; \mathcal{A} \vdash_\tau \left\{p(\mathit{vars}(f), \alpha)\right\} \; \mathit{code}(f) \; \left\{q(\mathit{vars}(f), \mathsf{ret})\right\}.$$

By the inductive assumption of the structural induction over the structure of syntactic triples, we have:

$$\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{p(\mathit{vars}(f), \alpha)\right\} \; \mathit{code}(f) \; \left\{q(\mathit{vars}(f), \mathsf{ret})\right\}$$

This clearly gives us enough to prove our goal, therefore, for the ordinal α, the following holds:

$$\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{p(\mathtt{x}, \alpha)\right\} \; \mathtt{y} := f(\mathtt{x}) \; \left\{q(\mathtt{x}, \mathtt{y})\right\}$$

Lemma 4. *If, for $p \in \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $q, \omega \in \coprod_{x \in X} Q(x) \to \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $x \in X$, $y \in Q(x)$*

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

then

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

Lemma 5 (Make Atomic Rule). *Suppose that*

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

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

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

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

$$\lambda; \mathcal{A} \vDash \left\langle p_p(s) * \exists x \in X.\, \mathbf{t}_a^{\lambda'}(x) * a \Mapsto \blacklozenge \right\rangle a \left\langle \overline{p_p'}(s') \right\rangle \tag{9}$$
$$\lambda; \mathcal{A}; \Omega \vDash_\tau \left\{\overline{p_p'}\right\} \mathbb{C}' \left\{\exists x \in X, y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y)\right\}. \tag{10}$$

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 (9) 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

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

We have that $p'_p * \exists x \in X. \, {}^{x}_{t}(a)\lambda' * a \Mapsto \blacklozenge \vDash_\tau \overline{p'_p}$ and is stable with respect to $\mathcal{A}$. From (10), by left consequence and the inductive hypothesis, we have

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

Finally, from (10) and Lemma 4, we have, for all $y \in Q(x)$

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

The remaining cases are simpler, or follow similar reasoning.

Lemma 6 (Update Region Rule). *Suppose that $a \notin \mathcal{A}$ and*

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

Then

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

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

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

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

$$\lambda; \mathcal{A} \vDash \begin{array}{c} \left\langle p_p(s) * I(\mathbf{t}_a^\lambda(x)) * p(x) \right\rangle \\ a \\ \left\langle p'_p(s') * I(\mathbf{t}_a^\lambda(x)) * p(x) \vee \begin{array}{c} \exists y \in Q(x), z \in Z. \, \overline{p''_p}(x, y, z, s') * \\ \left(\begin{array}{c} I(\mathbf{t}_a^\lambda(y)) * q_1(x, y, z) \vee \\ I(\mathbf{t}_a^\lambda(x)) * q_2(x, y, z) \end{array} \right) \end{array} \right\rangle \end{array} \quad (11)$$

$$\lambda; \mathcal{A}; \mathit{\Omega} \vDash_\tau \begin{array}{c} \left\langle p'_p \middle| I(\mathbf{t}_a^\lambda(x)) * p(x) \right\rangle \\ \mathbb{C}' \\ \mathbb{E}y \in Q(x), z \in Z. \left\langle q_p(x, y, z) \middle| \begin{array}{c} I(\mathbf{t}_a^\lambda(y)) * q_1(x, y, z) \vee \\ I(\mathbf{t}_a^\lambda(x)) * q_2(x, y, z) \end{array} \right\rangle \end{array} \quad (12)$$

$$\forall y \in Q(x), z \in Z. \quad \lambda; \mathcal{A}; \mathit{\Omega}(x, y, z) \vDash_\tau \left\{ \overline{p''_p}(x, y, z) \right\} \ \mathbb{C}' \ \left\{ q_p(x, y, z) \right\} \quad (13)$$

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

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

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 (11), there is some $\overline{\varphi'}$ with $\overline{\varphi} \ \mathrm{G}_{\lambda;\mathcal{A}} \ \overline{\varphi'}$, $h' \in \lfloor \varphi' \rfloor_\lambda$ and

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

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

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

  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

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

  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 $\varphi \ \mathrm{G}_{\lambda+1;\mathcal{A}'} \ \varphi'$.

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

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

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, $\varphi' \in p''_p(x, y, z, s') * \mathbf{t}^\lambda_a(x) * q_2(x, y, z) * r$, 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

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

So we have established that

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

By (12), it follows from the inductive hypothesis that

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

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

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

B Treiber's Stack

The proof of the `push` operation is in Fig. 12.

$$\forall \beta.\, \mathbb{V} vs, t, \alpha.$$
$$\left\langle \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \land \alpha > \beta(\alpha, vs) \right\rangle$$

abstract; substitute $s = r$, $t = (ns, ds)$

$$\left\langle \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * [\mathrm{G}]_r \land vs = \mathsf{snds}(ns) \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \right\rangle$$

make atomic

$$r : (ns, ds, \alpha) \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \rightsquigarrow ((n, \mathtt{v}) : ns, ds, \beta(\alpha, \mathsf{snds}(ns))) \vdash_\tau$$
$$\left\{ \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \right\}$$
$$\mathtt{y} := \mathtt{alloc}(2);$$
$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{node}(\mathtt{y}, \_, \_) \\ \land\, \alpha > \beta(\alpha, \mathsf{snds}(ns)) \end{array} \right\}$$
$$[\mathtt{y}] := \mathtt{v};$$
$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{node}(\mathtt{y}, \mathtt{v}, \_) \\ \land\, \alpha > \beta(\alpha, \mathsf{snds}(ns)) \end{array} \right\}$$
$$\mathtt{do}\ \{$$
$$\forall \gamma.$$
$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\ \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{node}(\mathtt{y}, \mathtt{v}, \_) \\ \land\, \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma \geq 2 \cdot \alpha + |ns| \end{array} \right\}$$

open region

$$\mathbb{V} ns, ds, \alpha.$$
$$\left\langle \exists y.\, \mathtt{x} \mapsto y * \mathsf{stack}(y, ns, ds) \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma \geq 2 \cdot \alpha + |ns| \right\rangle$$
$$\mathtt{z} := [\mathtt{x}];$$
$$\left\langle \begin{array}{l} \exists y.\, \mathtt{x} \mapsto y * \mathsf{stack}(y, ns, ds) \land \mathtt{z} = y \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \\ \land\, \gamma \geq 2 \cdot \alpha + |ns| \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{node}(\mathtt{y}, \mathtt{v}, \_) \\ \land\, \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ ((ns = [] \land \mathtt{z} = 0) \lor \mathsf{head}(ns) = (\mathtt{z}, \_) \lor \gamma > 2 \cdot \alpha + |ns|) \end{array} \right\}$$
$$[\mathtt{y.next}] := \mathtt{z};$$
$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \mathsf{node}(\mathtt{y}, \mathtt{v}, \mathtt{z}) \land \\ \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ ((ns = [] \land \mathtt{z} = 0) \lor \mathsf{head}(ns) = (\mathtt{z}, \_) \lor \gamma > 2 \cdot \alpha + |ns|) \end{array} \right\}$$

update region

$$\mathbb{V} ns, ds, \alpha.$$
$$\left\langle \begin{array}{l} \exists w.\, \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) * \mathsf{node}(\mathtt{y}, \mathtt{v}, \mathtt{z}) \land \\ \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma \geq 2 \cdot \alpha + |ns| \land \\ ((ns = [] \land \mathtt{z} = 0) \lor \mathsf{head}(ns) = (\mathtt{z}, \_) \lor \gamma > 2 \cdot \alpha + |ns|) \end{array} \right\rangle$$
$$\mathtt{b} := \mathtt{CAS}(\mathtt{x}, \mathtt{z}, \mathtt{y});$$
$$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \left(\begin{array}{c} \exists w.\, \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) * \mathsf{node}(\mathtt{y}, \mathtt{v}, \_) \\ \land\, \gamma > 2 \cdot \alpha + |ns| \end{array} \right) \\ \quad \underline{\mathbf{else}}\ \mathtt{x} \mapsto \mathtt{y} * \mathsf{stack}(\mathtt{y}, ((\mathtt{y}, \mathtt{v}) : ns, ds)) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}} \left(\begin{array}{c} \exists ns, ds, \alpha.\, \mathbf{TStack}_r(\mathtt{x}, ns, ds, \alpha) * r \Rrightarrow \blacklozenge * \\ \mathsf{node}(\mathtt{y}, \mathtt{v}, \_) \land \alpha > \beta(\alpha, \mathsf{snds}(ns)) \land \gamma > 2 \cdot \alpha + |ns| \end{array} \right) \\ \quad \underline{\mathbf{else}}\ \exists ns, ds, \alpha.\, r \Rrightarrow ((ns, ds, \alpha), ((\mathtt{y}, \mathtt{v}) : ns, ds, \beta(\alpha, \mathsf{snds}(ns)))) \end{array} \right\}$$
$$\}\ \mathtt{while}\ (\mathtt{b} = 0);$$
$$\left\{ \exists ns, ds, \alpha.\, r \Rrightarrow ((ns, ds, \alpha), ((\mathtt{y}, \mathtt{v}) : ns, ds, \beta(\alpha, \mathsf{snds}(ns)))) \right\}$$
$$\left\langle \exists n.\, \mathbf{TStack}_r(\mathtt{x}, (n, \mathtt{v}) : ns, ds, \beta(\alpha, vs)) * [\mathrm{G}]_r \land \mathtt{v} : vs = \mathsf{snds}((n, \mathtt{v}) : ns) \right\rangle$$
$$\left\langle \exists t'.\, \mathsf{Stack}(s, \mathtt{x}, \mathtt{v} : vs, t', \beta(\alpha, vs)) \right\rangle$$

Fig. 12: Proof of total correctness of Treiber's stack `push` operation.

C Linked List Map

C.1 Code

We introduce an implementation of a Map module that uses a linked list. Each node in the linked list represents a key-value pair that is in the map and is implemented with adjacent memory cells, the key, the value and a pointer to the next node (0 if there is no next element). Whenever an key-value pair with a key that was never previously in the map, a node is added at the end of the linked list (using the `LLMapAdd` function) storing the key value pair. If the key was previously in the map, a cell with this key must exist, as, when the delete operation of the map, `LLMapRemove`, finds a node in the linked list storing a key-value pair with a key it is trying to remove from the map, it simply updates the value of the node to 0, making it "invisible" to the map, rather than deleting it, so in this case, the add function of the map simply write the new value into the old cell with value 0. This imposses the restriction that a key cannot map to the value 0.

```
function makeLLMap() {
    x := alloc(1);
    [x] := 0;
    return x;
}
```

Fig. 13: `makeLLMap`

```
function LLMapAdd(x, k, v) {
    nn := alloc(3);
    [nn.key] := k;
    [nn.value] := v;
    [nn.next] := 0;
    l := 0;
    while (l = 0) {
        prev := x;
        n := [x];
        while (n ≠ 0) {
            nk := [n.key];
            if (nk = k) {
                [n.value] := v;
                return 0;
            }
            prev := n.next;
            n := [n.next];
        }
        l := CAS(prev, 0, nn);
    }
    return 0;
}
```

Fig. 14: LLMapAdd function.

```
function LLMapRemove(x, k) {
    n := [x];
    while (n ≠ 0) {
        nk := [n.key];
        if (nk = k) {
            b := 0;
            while (b = 0) {
                v := [n.value];
                if (v ≠ 0) {
                    b := CAS(n.value, v, 0);
                } else {
                    return 0;
                }
            }
            return 1;
        }
        n := [n.next];
    }
    return 0;
}
```

Fig. 15: LLMapRemove function.

```
function LLMapLookup(x, k) {
    n := [x];
    while (n ≠ 0) {
        nk := [n.key];
        if (nk = k) {
            v := [n.value];
            return v;
        }
        n := [n.next];
    }
    return 0;
}
```

Fig. 16: LLMapLookup function

$$\begin{aligned}
&\mathtt{n.key} = \mathtt{n}\\
&\mathtt{n.value} = \mathtt{n} + 1\\
&\mathtt{n.next} = \mathtt{n} + 2
\end{aligned}$$

Fig. 17: Object offsets

C.2 Specifications

We introduce the abstract specification for indexes. Note that the remove operation decreases the ordinal parametrising the abstract predicate as it impedes itself due to the CAS loop used to write the value 0 to a cell when removing it from that map to check that the previous value of the node is not 0 so that the function can return a value that informs the client wether or not the deletion found a key-value mapping to delete.

Otherwise, the specification defines a standard map module, the add function, `LLMapAdd`, adds a new key-value pair to the mapping, removing any previous mappings associated with the same key, the remove function, `LLMapRemove`, removes the mapping associated with a given key and the lookup function, `LLMapLookup`, returns the value associated with a given key within the map, or 0 if the key is not defined in the map.

$$\forall \alpha.\ \vdash_\tau \left\{\mathsf{emp}\right\}\ \mathtt{makeLLMap}()\ \left\{\exists s, t.\ \mathsf{LLMap}(s, \mathsf{ret}, \emptyset, t, \alpha)\right\}$$

$$\forall \beta.\ \vdash_\tau \mathbb{V} S, t, \alpha. \left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \alpha > \beta(S, \alpha) \wedge \mathtt{v} \neq 0 \right\rangle\ \mathtt{LLMapAdd}(\mathtt{x}, \mathtt{k}, \mathtt{v})\ \left\langle \exists t'.\ \mathsf{LLMap}(s, \mathtt{x}, (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, t', \beta(S, \alpha)) \right\rangle$$

$$\forall \beta.\ \vdash_\tau \mathbb{V} S, t, \alpha. \left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \alpha > \beta(S, \alpha) \right\rangle\ \mathtt{LLMapRemove}(\mathtt{x}, \mathtt{k})\ \left\langle \begin{array}{c} \underline{\textbf{if}}\ (\mathtt{k}, \_) \in S \\ \underline{\textbf{then}}\ \exists t'.\ \mathsf{LLMap}(s, \mathtt{x}, S \setminus \{(\mathtt{k}, \_)\}, t', \beta(S, \alpha)) \wedge \mathsf{ret} = 1 \\ \underline{\textbf{else}}\ \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \mathsf{ret} = 0 \end{array} \right\rangle$$

$$\vdash_\tau \mathbb{V} S, t, \alpha. \left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \right\rangle\ \mathtt{LLMapLookup}(\mathtt{x}, \mathtt{k})\ \left\langle \begin{array}{c} \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \\ (((\mathtt{k}, \mathsf{ret}) \in S) \vee (\mathsf{ret} = 0 \wedge (\mathtt{k}, \_) \notin S)) \end{array} \right\rangle$$

Fig. 18: Index operation specifications.

C.3 Definitions

We introduce the predicates, functions, regions and guards required for the proof. The function *map* gives the key-values pairs of a map given the abstract state of the linked list, a list of triples of node address, keys and values and the *dom* function gives the set of all of the keys that a given map has ever been defined over (including those where a 0 has been written to the node representing the mapping to remove it) from this same list.

$$map(ls) = \begin{cases} \{(k, v)\} \uplus map(ls') & ls = (x, k, v) : ls' \wedge v \neq 0 \\ map(ls') & ls = (x, k, 0) : ls' \\ \emptyset & ls = [] \end{cases}$$

$$dom(ls) = \begin{cases} \{k\} \uplus dom(ls') & ls = (x, k, v) : ls' \\ \emptyset & ls = [] \end{cases}$$

$$\mathsf{LinkedListMap}(x, ls) \triangleq \begin{cases} \exists y.\, x \mapsto k, v, y * \mathsf{LinkedListMap}(y, ls') & ls = (x, k, v) : ls' \\ \mathsf{emp} & ls = [] \land x = 0 \end{cases}$$

$$I(\mathbf{LinkedListMap}_a(x, ls, \alpha)) \triangleq \exists y.\, x \mapsto y * \mathsf{LinkedListMap}(y, ls)$$

$$\mathsf{LLMap}(a, x, S, ls, \alpha) \triangleq \mathbf{LinkedListMap}_a(x, ls, \alpha) * [\mathrm{G}]_a \land map(ls) = S$$

$$\mathrm{G} : \forall x, k, v, hs, ts, \alpha, \beta < \alpha.\ (hs \mathbin{++} (x, k, \_) : ts, \alpha) \rightsquigarrow (hs \mathbin{++} (x, k, v) : ts, \beta)$$
$$\mathrm{G} : \forall x, k, v, ls, \alpha, \beta < \alpha.\ (ls, \alpha) \land k \notin dom(ls) \rightsquigarrow (ls \mathbin{++} [(x, k, v)], \beta)$$
$$\mathrm{G} : \forall ls, \alpha, \beta < \alpha.\ (ls, \alpha) \rightsquigarrow (ls, \beta)$$

This transition system, allows the value of a cell to be arbitrarily changed, cells to be appended to the end of the list, both involve simulatenously decreasing the ordinal parametrising the region, and the ordinal parametrising the region to be arbitrarily decreased.

C.4 Proofs

Below are the proofs of total correctness of the operations on the Linked List map. There are two loop invariants used in these proofs. The first is simply the ordinal parametrizing the region, in the case of loops that have to continue executing due to concurrent writes from the environment, as can be seen in the outer loop of the `LLMapAdd` function and in the inner loop of the `LLMapRemove` function and the second is $\alpha \cdot 2 + |ts|$ for loops that traverse the list, where ts is the part of the list the loop is yet to traverse, as whenever the list is lengthened, $|ts|$ increases by one, but α must decrease by at least the same amount so the total value of the expression must decrease, and each time the loop progresses in its traversal, $|ts|$ decreases, so the total value of the expression once again decreases.

abstract; substitute $s = a$, $t = ls$

make atomic

update region

$$\forall \beta.$$
$$\mathbb{V} S, t, \alpha.$$
$$\left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \alpha > \beta(S, \alpha) \wedge \mathtt{v} \neq 0 \right\rangle$$
$$\left\langle \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * [\mathrm{G}]_a \wedge map(ls) = S \wedge \alpha > \beta(S, \alpha) \wedge \mathtt{v} \neq 0 \right\rangle$$
$$a : (ls, \alpha) \rightsquigarrow (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha)) \vdash_\tau$$
$$\left\{ \begin{array}{l} \exists ls, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \end{array} \right\}$$
```
nn := alloc(3);
[nn.key] := k;
[nn.value] := v;
[nn.next] := 0;
```
$$\left\{ \begin{array}{l} \exists ls, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \end{array} \right\}$$
```
l := 0;
```
$$\left\{ \begin{array}{l} \exists ls, \alpha.\ (\underline{\mathbf{if}}\ \mathtt{l} = 0\ \underline{\mathbf{then}} \left(\begin{array}{c} \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \end{array} \right) \\ \quad \underline{\mathbf{else}}\ a \Mapsto ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha)))) \end{array} \right\}$$
```
while (l = 0) {
```
$$\forall \gamma.$$
$$\left\{ \begin{array}{l} \exists ls, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \end{array} \right\}$$
```
prev := x;
```
$$\left\{ \begin{array}{l} \exists ls, \alpha. \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \mathtt{prev} = \mathtt{x} \end{array} \right\}$$
$$\mathbb{V} ls, \alpha.$$
$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \\ ((ls = (y, k, v) : ls') \vee (y = 0 \wedge ls = [])) \end{array} \right\rangle$$
```
n := [x];
```
$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \\ ((ls = (y, k, v) : ls') \vee (y = 0 \wedge ls = [])) \wedge \mathtt{n} = y \end{array} \right\rangle$$
$$\left\{ \begin{array}{l} \exists k, k', v, v', ls, hs, ts, \alpha. \\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \mathtt{k} \notin dom(hs) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \\ \left(\begin{array}{c} (ls = hs +\!\!+ (\mathtt{prev}, k', v') : (\mathtt{n}, k, v) : ts \wedge k' \neq \mathtt{k}) \vee \\ (ls = hs +\!\!+ (\mathtt{prev} - 2, k', v') : ts \wedge ts \neq [] \Rightarrow \alpha < \gamma \wedge \mathtt{n} = 0 \wedge k' \neq \mathtt{k}) \vee \\ (ls = (\mathtt{n}, k, v) : ts \wedge hs = [] \wedge \mathtt{prev} = \mathtt{x}) \vee \\ (ls = ts \wedge hs = [] \wedge ts \neq [] \Rightarrow \alpha < \gamma \wedge \mathtt{prev} = \mathtt{x} \wedge \mathtt{n} = 0) \end{array} \right) \end{array} \right\}$$
```
// see figure 20
```
$$\left\{ \begin{array}{l} \exists k, k', v, v', ls, ts, hs, \alpha. \\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \\ \left(\begin{array}{l} (ls = hs +\!\!+ (\mathtt{prev}, k', v') : ts \wedge ts \neq [] \Rightarrow \alpha < \gamma) \vee \\ (ls = ts \wedge hs = [] \wedge ts \neq [] \Rightarrow \alpha < \gamma \wedge \mathtt{prev} = \mathtt{x}) \end{array} \right) \end{array} \right\}$$
```
l := CAS(prev, 0, nn);
```
$$\left\{ \begin{array}{l} \exists ls, \alpha.\ \underline{\mathbf{if}}\ \mathtt{l} = 0\ \underline{\mathbf{then}} \left(\begin{array}{c} \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Mapsto \blacklozenge * \\ \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \end{array} \right) \wedge \\ \qquad\qquad \underline{\mathbf{else}}\ a \Mapsto ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha))) \\ \alpha < \gamma \end{array} \right\}$$
```
}
```
$$\left\{ \exists ls, \alpha.\ a \Mapsto ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha))) \right\}$$
```
return 0;
```
$$\left\{ \exists ls, \alpha.\ a \Mapsto ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha))) \right\}$$
$$\left\langle \mathbf{LinkedListMap}_a(\mathtt{x}, ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha)) * [\mathrm{G}]_a \wedge map(ls) = S \right\rangle$$
$$\left\langle \exists t'.\ \mathsf{LLMap}(s, \mathtt{x}, (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, t', \beta(S, \alpha)) \right\rangle$$

Fig. 19: Proof of total correctness of `LLMapAdd` function, details of inner while loop in figure 20.

```
while (n ≠ 0) {
```

$$\forall\delta.$$

$$\left\{\begin{array}{l}\exists k, k', v, v', ls, hs, ts, \alpha.\\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \mathtt{k} \notin dom(hs) \wedge\\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \alpha \cdot 2 + |ts| \leq \delta \wedge\\ \left(\begin{array}{c}(ls = hs +\!\!+ (\mathtt{prev}, k', v') : (\mathtt{n}, k, v) : ts \wedge k' \neq \mathtt{k}) \vee\\ (ls = (\mathtt{n}, k, v) : ts \wedge hs = [] \wedge \mathtt{prev} = \mathtt{x})\end{array}\right)\end{array}\right\}$$

open region

$$\mathbb{V} ls, \alpha$$

$$\langle \exists y, k, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge ls = hs +\!\!+ (\mathtt{n}, k, v) : ts \rangle$$

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

$$\langle \exists y, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{nk}, v) : ts \rangle$$

$$\left\{\begin{array}{l}\exists k', v, v', ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge\\ \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \alpha \cdot 2 + |ts| \leq \delta \wedge\\ \left(\begin{array}{c}(ls = hs +\!\!+ (\mathtt{prev} - 2, k', v') : (\mathtt{n}, \mathtt{nk}, v) : ts \wedge k' \neq \mathtt{k}) \vee\\ (ls = (\mathtt{n}, \mathtt{nk}, v) : ts \wedge hs = [] \wedge \mathtt{prev} = \mathtt{x})\end{array}\right)\end{array}\right\}$$

```
if (nk = k) {
```

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in dom(ls) \wedge\\ ls = (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(map(ls), \alpha)\end{array}\right\}$$

update region

$$\mathbb{V} ls, \alpha$$

$$\left\langle\begin{array}{l}\exists y.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \mathtt{k} \in dom(ls) \wedge\\ ls = (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(map(ls), \alpha)\end{array}\right\rangle$$

```
[n.value] := v;
```

$$\left\langle\begin{array}{l}\exists y.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \mathtt{k} \in dom(ls) \wedge\\ ls = (\mathtt{n}, \mathtt{k}, \mathtt{v}) : ts \wedge \alpha > \beta(map(ls), \alpha)\end{array}\right\rangle$$

$$\{\exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha)))\}$$

```
return 0;
```

$$\{\exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto \mathtt{v}], \beta(map(ls), \alpha)))\}$$

```
}
```

$$\left\{\begin{array}{l}\exists k', v, v', ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge\\ \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \leq \gamma \wedge \alpha \cdot 2 + |ts| \leq \delta \wedge \mathtt{nk} \neq \mathtt{k} \wedge\\ \left(\begin{array}{c}(ls = hs +\!\!+ (\mathtt{prev} - 2, k', v') : (\mathtt{n}, k, v) : ts \wedge k' \neq \mathtt{k}) \vee\\ (ls = (\mathtt{n}, \mathtt{nk}, v) : ts \wedge hs = [] \wedge \mathtt{prev} = \mathtt{x})\end{array}\right)\end{array}\right\}$$

```
prev := n.next;
```

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge\\ \mathtt{nk} \neq \mathtt{k} \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{nk}, v) : ts \wedge \mathtt{k} \notin dom(hs) \wedge\\ \alpha \leq \gamma \wedge \alpha \cdot 2 + |ts| \leq \delta \wedge \mathtt{nk} \neq \mathtt{k} \wedge \mathtt{n} = \mathtt{prev} - 2\end{array}\right\}$$

```
n := [n.next];
```

$$\left\{\begin{array}{l}\exists k, k', v, v', ls, hs, ts, \alpha.\\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge * \mathsf{LinkedListMap}(\mathtt{nn}, [(\mathtt{nn}, \mathtt{k}, \mathtt{v})]) \wedge \mathtt{k} \notin dom(hs) \wedge\\ \alpha > \beta(map(ls), \alpha) \wedge \mathtt{v} \neq 0 \wedge \alpha \cdot 2 + |ts| < \delta \wedge\\ \left(\begin{array}{c}(ls = hs +\!\!+ (\mathtt{prev}, k', v') : (\mathtt{n}, k, v) : ts \wedge k' \neq \mathtt{k}) \vee\\ (ls = hs +\!\!+ (\mathtt{prev} - 2, k', v') : ts \wedge ts \neq [] \Rightarrow \alpha < \gamma \wedge \mathtt{n} = 0 \wedge k' \neq \mathtt{k}) \vee\\ (ls = (\mathtt{n}, k, v) : ts \wedge hs = [] \wedge \mathtt{prev} = \mathtt{x}) \vee\\ (ls = ts \wedge hs = [] \wedge ts \neq [] \Rightarrow \alpha < \gamma \wedge \mathtt{prev} = \mathtt{x} \wedge \mathtt{n} = 0)\end{array}\right)\end{array}\right\}$$

```
}
```

Fig. 20: Details of `LLMapAdd` inner loop proof.

abstract; substitute $s = a,\ t = ls$

make atomic

$$\forall \beta.$$
$$\mathbb{W} S, t, \alpha.$$
$$\left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \alpha > \beta(S, \alpha) \right\rangle$$
$$\left\langle \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * [\mathrm{G}]_a \wedge map(ls) = S \wedge \alpha > \beta(S, \alpha) \right\rangle$$
$$a : (ls, \alpha) \rightsquigarrow \underline{\mathbf{if}}\ (\mathtt{k}, \_) \in map(ls)\ \underline{\mathbf{then}}\ (ls[\mathtt{k} \mapsto 0], \beta(map(ls), \alpha))\ \underline{\mathbf{else}}\ (ls, \alpha) \vdash_\tau$$
$$\left\{ \exists ls, \alpha.\, \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \alpha > \beta(map(ls), \alpha) \right\}$$

update region

$$\mathbb{W} ls, \alpha.$$
$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ x \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \\ ((ls = (y, k, v) : ls') \vee (y = 0 \wedge ls = [])) \end{array} \right\rangle$$
$$\mathtt{n} := [\mathtt{x}];$$
$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ x \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \\ ((ls = (y, k, v) : ls') \vee (y = 0 \wedge ls = [])) \wedge \mathtt{n} = y \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ls, ts, \alpha. \\ \left(\left(\exists k, v, hs.\ \begin{array}{c} \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, k, v) : ts \wedge \mathtt{k} \notin dom(hs) \wedge \mathtt{n} \neq 0 \end{array} \right) \vee \atop (a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \wedge ts = [] \wedge (\mathtt{k}, \_) \notin map(ls) \wedge \mathtt{n} = 0) \right) \wedge \alpha > \beta(map(ls), \alpha) \end{array} \right\}$$

```
while (n ≠ 0) {
```

$$\forall \gamma.$$
$$\left\{ \begin{array}{l} \exists k, v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, k, v) : ts \wedge \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \alpha \cdot 2 + |ts| \leq \gamma \end{array} \right\}$$

open region

$$\mathbb{W} ls, \alpha$$
$$\left\langle \begin{array}{l} \exists y, k, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, k, v) : ts \wedge \mathtt{k} \notin dom(hs) \end{array} \right\rangle$$
$$\mathtt{nk} := [\mathtt{n.key}];$$
$$\left\langle \begin{array}{l} \exists y, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, \mathtt{nk}, v) : ts \wedge \mathtt{k} \notin dom(hs) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, \mathtt{nk}, v) : ts \wedge \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \alpha \cdot 2 + |ts| \leq \gamma \end{array} \right\}$$

// see figure 22

$$\left\{ \begin{array}{l} \exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge ls = hs \mathbin{++} (\mathtt{n}, \mathtt{nk}, v) : ts \wedge \\ \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \alpha \cdot 2 + |ts| \leq \gamma \wedge \mathtt{nk} \neq \mathtt{k} \end{array} \right\}$$
$$\mathtt{n} := [\mathtt{n.next}];$$
$$\left\{ \begin{array}{l} \exists ls, ts, \alpha. \\ \left(\left(\exists k, v, hs.\ \begin{array}{l} \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ ls = hs \mathbin{++} (\mathtt{n}, k, v) : ts \wedge \mathtt{k} \notin dom(hs) \wedge \alpha > \beta(map(ls), \alpha) \wedge \mathtt{n} \neq 0 \end{array} \right) \vee \atop (a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \wedge ts = [] \wedge (\mathtt{k}, \_) \notin map(ls) \wedge \mathtt{n} = 0) \right) \wedge \\ \alpha \cdot 2 + |ts| < \gamma \end{array} \right\}$$

```
}
```

$$\left\{ \exists ls, \alpha.\, a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \wedge \mathtt{n} = 0 \wedge (\mathtt{k}, \_) \notin map(ls) \right\}$$

```
return 0;
```

$$\left\{ \exists ls, ls', \alpha, \alpha'.\ \left(\begin{array}{c} \underline{\mathbf{if}}\ (\mathtt{k}, \_) \in map(ls) \\ \underline{\mathbf{then}}\ ls' = ls[\mathtt{k} \mapsto 0] \wedge \alpha' = \beta(map(ls), \alpha) \wedge \mathsf{ret} = 1 \\ \underline{\mathbf{else}}\ ls' = ls \wedge \alpha' = \alpha \wedge \mathsf{ret} = 0 \end{array} \right) * a \Rrightarrow ((ls, \alpha), (ls', \alpha')) \right\}$$

$$\exists ls', \alpha'.$$
$$\left\langle \begin{array}{l} \left(\begin{array}{l} \underline{\mathbf{if}}\ (\mathtt{k}, \_) \in map(ls) \\ \underline{\mathbf{then}}\ ls' = ls[\mathtt{k} \mapsto 0] \wedge \alpha' = \beta(map(ls), \alpha) \wedge \mathsf{ret} = 1 \\ \underline{\mathbf{else}}\ ls' = ls \wedge \alpha' = \alpha \wedge \mathsf{ret} = 0 \end{array} \right) * \\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls', \alpha) * [\mathrm{G}]_a \wedge map(ls) = S \end{array} \right\rangle$$

$$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ (\mathtt{k}, \_) \in S\ \underline{\mathbf{then}}\ \exists t'.\, \mathsf{LLMap}(s, \mathtt{x}, S \setminus \{(\mathtt{k}, \_)\}, t', \beta(S, \alpha)) \wedge \mathsf{ret} = 1 \\ \underline{\mathbf{else}}\ \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \wedge \mathsf{ret} = 0 \end{array} \right\rangle$$

Fig. 21: Proof of total correctness of `LLMapRemove` function, details of inner while loop in figure 22.

$\mathtt{if}\ (\mathtt{nk} = \mathtt{k})\ \{$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ \mathtt{k} \in \mathit{dom}(ls) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(\mathit{map}(ls), \alpha)\end{array}\right\}$$

$\mathtt{b} := 0;$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \left(\begin{array}{c}(\mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge \mathtt{b} = 0) \vee \\ (a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto 0], \alpha)) \wedge (\mathtt{k}, \_) \in \mathit{map}(ls) \wedge \mathtt{b} = 1)\end{array}\right) \wedge \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(\mathit{map}(ls), \alpha)\end{array}\right\}$$

$\mathtt{while}\ (\mathtt{b} = 0)\ \{$

$\forall \delta.$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \\ \mathtt{k} \in \mathit{dom}(ls) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(\mathit{map}(ls), \alpha) \wedge \alpha > \beta(\mathit{map}(ls), \alpha) \wedge \alpha \leq \delta\end{array}\right\}$$

update region

$\mathbb{V} ls, \alpha.$

$$\left\langle \exists y, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha \leq \delta \right\rangle$$

$\mathtt{v} := [\mathtt{n.value}];$

$$\left\langle\begin{array}{l}\exists y.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \wedge \\ \left(\begin{array}{c}(\exists v, hs, ts.\ \mathtt{k} \in \mathit{dom}(ls) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \mathtt{v} \neq v \Rightarrow \alpha < \delta \wedge \alpha \leq \delta \wedge \mathtt{v} \neq 0) \vee \\ ((\mathtt{k}, \_) \notin \mathit{map}(ls) \wedge \mathtt{v} = 0)\end{array}\right)\end{array}\right\rangle$$

$$\left\{\exists ls, \alpha.\ \left(\begin{array}{c}\left(\exists v, hs, ts.\ \begin{array}{c}\mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(\mathit{map}(ls), \alpha) \wedge \\ \mathtt{v} \neq v \Rightarrow \alpha < \delta \wedge \alpha \leq \delta \wedge \mathtt{v} \neq 0\end{array}\right) \vee \\ (a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \wedge (\mathtt{k}, \_) \notin \mathit{map}(ls) \wedge \mathtt{v} = 0)\end{array}\right)\right\}$$

$\mathtt{if}\ (\mathtt{v} \neq 0)\ \{$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha > \beta(\mathit{map}(ls), \alpha) \wedge \mathtt{v} \neq v \Rightarrow \alpha < \gamma \wedge \alpha \leq \delta\end{array}\right\}$$

$\mathtt{b} := \mathtt{CAS}(\mathtt{n.value}, \mathtt{v}, 0);$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \left(\begin{array}{c}(\mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge \mathtt{b} = 0) \vee \\ (a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto 0], \beta(\mathit{map}(ls), \alpha))) \wedge (\mathtt{k}, \_) \in \mathit{map}(ls) \wedge \mathtt{b} = 1)\end{array}\right) \wedge \\ \alpha > \beta(\mathit{map}(ls), \alpha) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha < \delta\end{array}\right\}$$

$\}\ \mathtt{else}\ \{$

$$\left\{\exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \wedge (\mathtt{k}, \_) \notin \mathit{map}(ls)\right\}$$

$\mathtt{return}\ 0;$

$$\left\{\begin{array}{l}\exists ls, ls', \alpha, \alpha'.\ \left(\begin{array}{r}\underline{\mathbf{if}}\ (\mathtt{k}, \_) \in \mathit{map}(ls)\ \underline{\mathbf{then}}\ ls' = ls[\mathtt{k} \mapsto 0] \wedge \alpha' = \beta(\mathit{map}(ls), \alpha) \wedge \mathsf{ret} = 1 \\ \underline{\mathbf{else}}\ ls' = ls \wedge \alpha' = \alpha \wedge \mathsf{ret} = 0\end{array}\right) * \\ a \Rrightarrow ((ls, \alpha), (ls', \alpha'))\end{array}\right\}$$

$\}$

$$\left\{\begin{array}{l}\exists v, ls, hs, ts, \alpha.\ \left(\begin{array}{c}(\mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{k} \in \mathit{dom}(ls) \wedge \mathtt{b} = 0) \vee \\ (a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto 0], \beta(\mathit{map}(ls), \alpha))) \wedge (\mathtt{k}, \_) \in \mathit{map}(ls) \wedge \mathtt{b} = 1)\end{array}\right) \wedge \\ \alpha > \beta(\mathit{map}(ls), \alpha) \wedge ls = hs +\!\!+ (\mathtt{n}, \mathtt{k}, v) : ts \wedge \alpha < \delta\end{array}\right\}$$

$\}$

$$\left\{\exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls[\mathtt{k} \mapsto 0], \beta(\mathit{map}(ls), \alpha))) \wedge (\mathtt{k}, \_) \in \mathit{map}(ls)\right\}$$

$\mathtt{return}\ 1;$

$$\left\{\begin{array}{l}\exists ls, ls', \alpha, \alpha'.\ \left(\begin{array}{l}\underline{\mathbf{if}}\ (\mathtt{k}, \_) \in \mathit{map}(ls) \\ \underline{\mathbf{then}}\ ls' = ls[\mathtt{k} \mapsto 0] \wedge \alpha' = \beta(\mathit{map}(ls), \alpha) \wedge \mathsf{ret} = 1 \\ \underline{\mathbf{else}}\ ls' = ls \wedge \alpha' = \alpha \wedge \mathsf{ret} = 0\end{array}\right) * \\ a \Rrightarrow ((ls, \alpha), (ls', \alpha'))\end{array}\right\}$$

$\}$

Fig. 22: Details of `LLMapRemove` function inner loop proof.

abstract; substitute $s = a,\ t = ls$

make atomic

update region

open region

$$\mathbb{V} S, t, \alpha.$$

$$\left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \right\rangle$$

$$\left\langle \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * [\mathrm{G}]_a \land map(ls) = S \right\rangle$$

$$a : (ls, \alpha) \rightsquigarrow (ls, \alpha) \vdash_\tau$$

$$\left\{ \exists ls, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \right\}$$

$$\mathbb{V} ls, \alpha.$$

$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ x \mapsto y * \mathsf{LinkedListMap}(y, ls) \land \\ ((ls = (y, k, v) : ls') \lor (y = 0 \land ls = [])) \end{array} \right\rangle$$

$\mathtt{n} := [\mathtt{x}];$

$$\left\langle \begin{array}{l} \exists y, k, v, ls'.\ x \mapsto y * \mathsf{LinkedListMap}(y, ls) \land \\ ((ls = (y, k, v) : ls') \lor (y = 0 \land ls = [])) \land \mathtt{n} = y \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ls, ts, \alpha. \\ \left(\begin{array}{l} \left(\begin{array}{l} \exists hs, k, v.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, k, v) : ts \land \mathtt{k} \notin dom(hs) \land \mathtt{n} \neq 0 \end{array} \right) \lor \\ (a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land (\mathtt{k}, \_) \notin map(ls) \land ts = [] \land \mathtt{n} = 0) \end{array} \right) \end{array} \right\}$$

$\mathtt{while}\ (\mathtt{n} \neq 0)\ \{$

$$\forall \beta.$$

$$\left\{ \begin{array}{l} \exists k, v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, k, v) : ts \land \mathtt{k} \notin dom(hs) \land \alpha \cdot 2 + |ts| \leq \beta \end{array} \right\}$$

$$\mathbb{V} ls, \alpha$$

$$\left\langle \begin{array}{l} \exists y, k, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, k, v) : ts \land \mathtt{k} \notin dom(hs) \end{array} \right\rangle$$

$\mathtt{nk} := [\mathtt{n.key}];$

$$\left\langle \begin{array}{l} \exists y, v, hs, ts.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{nk}, v) : ts \land \mathtt{k} \notin dom(hs) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{nk}, v) : ts \land \mathtt{k} \notin dom(hs) \land \alpha \cdot 2 + |ts| \leq \beta \end{array} \right\}$$

$\mathtt{if}\ (\mathtt{nk} = \mathtt{k})\ \{$

$$\left\{ \exists ls, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \mathtt{k} \in dom(ls) \right\}$$

$$\mathbb{V} ls, \alpha$$

$$\left\langle \exists y.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \land \mathtt{k} \in dom(ls) \right\rangle$$

$\mathtt{v} := [\mathtt{n.value}];$

$$\left\langle \begin{array}{l} \exists y.\ \mathtt{x} \mapsto y * \mathsf{LinkedListMap}(y, ls) \land \\ (((\mathtt{k}, \mathtt{v}) \in map(ls)) \lor (\mathtt{v} = 0 \land (\mathtt{k}, \_) \notin map(ls))) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land \\ (((\mathtt{k}, \mathtt{v}) \in map(ls)) \lor (\mathtt{v} = 0 \land (\mathtt{k}, \_) \notin map(ls))) \end{array} \right\}$$

$\mathtt{return}\ \mathtt{v};$

$$\left\{ \begin{array}{l} \exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land \\ (((\mathtt{k}, \mathsf{ret}) \in map(ls)) \lor (\mathsf{ret} = 0 \land (\mathtt{k}, \_) \notin map(ls))) \end{array} \right\}$$

$\}$

$$\left\{ \begin{array}{l} \exists v, ls, hs, ts, \alpha.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, \mathtt{nk}, v) : ts \land \mathtt{k} \notin dom(hs) \land \alpha \cdot 2 + |ts| \leq \beta \land \mathtt{nk} \neq \mathtt{k} \end{array} \right\}$$

$\mathtt{n} := [\mathtt{n.next}];$

$$\left\{ \begin{array}{l} \exists ls, ts, \alpha. \\ \left(\begin{array}{l} \left(\begin{array}{l} \exists k, v, hs.\ \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * a \Rrightarrow \blacklozenge \land \\ ls = hs +\!\!+ (\mathtt{n}, k, v) : ts \land \mathtt{k} \notin dom(hs) \land \mathtt{n} \neq 0 \end{array} \right) \lor \\ (a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land (\mathtt{k}, \_) \notin map(ls) \land ts = [] \land \mathtt{n} = 0) \end{array} \right) \land \alpha \cdot 2 + |ts| < \beta \end{array} \right\}$$

$\}$

$$\left\{ \exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land (\mathtt{k}, \_) \notin map(ls) \right\}$$

$\mathtt{return}\ 0;$

$$\left\{ \begin{array}{l} \exists ls, \alpha.\ a \Rrightarrow ((ls, \alpha), (ls, \alpha)) \land \\ (((\mathtt{k}, \mathsf{ret}) \in map(ls)) \lor (\mathsf{ret} = 0 \land (\mathtt{k}, \_) \notin map(ls))) \end{array} \right\}$$

$$\left\langle \begin{array}{l} \mathbf{LinkedListMap}_a(\mathtt{x}, ls, \alpha) * [\mathrm{G}]_a \land map(ls) = S \land \\ (((\mathtt{k}, \mathsf{ret}) \in S) \lor (\mathsf{ret} = 0 \land (\mathtt{k}, \_) \notin S)) \end{array} \right\rangle$$

$$\left\langle \mathsf{LLMap}(s, \mathtt{x}, S, t, \alpha) \land (((\mathtt{k}, \mathsf{ret}) \in S) \lor (\mathsf{ret} = 0 \land (\mathtt{k}, \_) \notin S)) \right\rangle$$

Fig. 23: Proof of total correctness of `LLMapLookup` function.

D Hash Map

D.1 Code

We introduce a Hash Map module that uses the linked list module above as the list stored in each element of the array that this hashtable uses. It is made up of an array whos first element is a number denoting the size of the array, n, and then n cells containing the addresses of linked lists. The hash function used is the remainder function. Whenever a key has remainder i modulo n, any operations related to this key will be performed on the list in the i^{th} element of the array.

```
function makeHMap(n) {
    a := alloc(n + 1);
    [a] := n;
    i := 1;
    while (i ≤ n) {
        l := makeLLMap();
        [a + i] := l;
        i := i + 1;
    }
    return a;
}
```

Fig. 24: makeHMap

```
function HMapAdd(x, k, v) {
    n := [x];
    i := k % n;
    l := [x + i + 1];
    LLMapAdd(l, k, v);
    return 0;
}
```

Fig. 25: HMapAdd

```
function HMapRemove(x, k) {
    n := [x];
    i := k % n;
    l := [x + i + 1];
    b := LLMapRemove(l, k);
    return b;
}
```

Fig. 26: HMapRemove

```
function HMapLookup(x, k) {
    n := [x];
    i := k % n;
    l := [x + i + 1];
    v := LLMapLookup(l, k);
    return v;
}
```

Fig. 27: HMapLookup

D.2 Specifications

This Map style specification is almost identical to that of the linked list map as it simply "refers" operations to the right linked list.

$$\forall \alpha.\vdash_\tau \{\mathsf{emp}\}\ \mathtt{makeHMap}()\ \{\exists s.\,\mathsf{HMap}(s,\mathsf{ret},\emptyset,\alpha)\}$$

$$\forall \beta.\vdash_\tau \mathbb{V}S,\alpha.\langle \mathsf{HMap}(s,\mathtt{x},S,\alpha)\wedge\alpha>\beta(S,\alpha)\wedge\mathtt{v}\neq 0\rangle\ \mathtt{HMapAdd}(\mathtt{x},\mathtt{k},\mathtt{v})\ \langle \mathsf{HMap}(s,\mathtt{x},(S\setminus\{(\mathtt{k},\_)\})\uplus\{(\mathtt{k},\mathtt{v})\},\beta(S,\alpha))\rangle$$

$$\forall \beta.\vdash_\tau \mathbb{V}S,\alpha.\left\langle \mathsf{HMap}(s,\mathtt{x},S,\alpha)\wedge\alpha>\beta(S,\alpha)\right\rangle\ \mathtt{HMapRemove}(\mathtt{x},\mathtt{k})\ \left\langle \begin{array}{c}\underline{\mathbf{if}}\ (\mathtt{k},\_)\in S\\ \underline{\mathbf{then}}\ \mathsf{HMap}(s,\mathtt{x},S\setminus\{(\mathtt{k},\_)\},\beta(S,\alpha))\wedge\mathsf{ret}=1\\ \underline{\mathbf{else}}\ \mathsf{HMap}(s,\mathtt{x},S,\alpha)\wedge\mathsf{ret}=0\end{array}\right\rangle$$

$$\vdash_\tau \mathbb{V}S,\alpha.\left\langle \mathsf{HMap}(s,\mathtt{x},S,\alpha)\right\rangle\ \mathtt{HMapLookup}(\mathtt{x},\mathtt{k})\ \left\langle \begin{array}{c}\mathsf{HMap}(s,\mathtt{x},S,\alpha)\wedge\\ (((\mathtt{k},\mathsf{ret})\in S)\vee(\mathsf{ret}=0\wedge(\mathtt{k},\_)\notin S))\end{array}\right\rangle$$

Fig. 28: Index operation specifications.

D.3 Definitions

We defined the regions, abstract predicates and guards required for the proof.

$$I(\mathbf{HashMap}_a(s',x,n,arr,S,\alpha))\triangleq$$

$$x\mapsto n*\bigcircledast_{i=0}^{n-1}(x{+}i{+}1\mapsto arr(i)*\exists t.\,\mathsf{LLMap}(s'(i),arr(i),\{(k,v)|(k,v)\in S,k\%n=i\},t,\alpha))$$

$$\mathsf{HMap}((a, n, arr, s'), x, S, \alpha) \triangleq \mathbf{HashMap}_a(s', x, n, arr, S, \alpha) * [\mathrm{H}]_a$$

The first two actions are for the different kinds of adds to the map (depending on wether or not they key being added was previously in the map), the third encodes deletion and the last simply allows the ordinal parametrizing the region to be decreased.

$$\begin{aligned}
&\mathrm{H} : \forall S, k, v, \alpha, \beta < \alpha.\ (S \uplus \{(k, _)\}, \alpha) \rightsquigarrow (S \uplus \{(k, v)\}, \beta)\\
&\mathrm{H} : \forall S, k, v, \alpha, \beta < \alpha.\ (S, \alpha) \wedge (k, _) \notin S \rightsquigarrow (S \uplus \{(k, v)\}, \beta)\\
&\mathrm{H} : \forall S, k, v, \alpha, \beta < \alpha.\ (S \uplus \{(k, _)\}, \alpha) \rightsquigarrow (S, \beta)\\
&\mathrm{H} : \forall S, \alpha, \beta < \alpha.\ (S, \alpha) \rightsquigarrow (S, \beta)
\end{aligned}$$

D.4 Proofs

In this subsection, we give the proofs of total correctness of the operations of the Hash Map module. The functions are all quite small and follow the same pattern, so the proofs are very similar.

abstract; substitute $s = (a, n, arr, s')$

make atomic

open region

open region

update region

$$\forall \beta.$$
$$\mathbb{V} S, \alpha.$$
$$\left\langle \mathsf{HMap}(s, \mathtt{x}, S, \alpha) \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \right\rangle$$
$$\left\langle \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * [\mathrm{H}]_a \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \right\rangle$$
$$a : (S, \alpha) \rightsquigarrow ((S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, \beta(S, \alpha)) \vdash_\tau$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \right\}$$
$$\mathbb{V} S, \alpha$$
$$\left\langle x \mapsto n * \circledast_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \right\rangle$$
$$\mathtt{n} := [\mathtt{x}];$$
$$\left\langle x \mapsto n * \circledast_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha) \land \mathtt{n} = n \right\rangle$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \right\}$$
$$\mathtt{i} := \mathtt{k} \,\%\, \mathtt{n};$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \right\}$$
$$\mathbb{V} S, \alpha$$
$$\left\langle x \mapsto n * \circledast_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \right\rangle$$
$$\mathtt{l} := [\mathtt{x} + \mathtt{i} + 1];$$
$$\left\langle \begin{array}{l} x \mapsto n * \circledast_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \land \\ \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \end{array} \right\rangle$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \right\}$$
$$\mathbb{V} S, t, \alpha$$
$$\left\langle \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, \{(k, v) | (k, v) \in S, k\%n = \mathtt{i}\}, t, \alpha) \land \alpha > \beta(S, \alpha) \land \mathtt{v} \neq 0 \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \right\rangle$$
$$\mathtt{LLMapAdd}(\mathtt{l}, \mathtt{k}, \mathtt{v});$$
$$\left\langle \begin{array}{l} \exists t'.\, \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{x}, (\{(k, v) | (k, v) \in S, k\%n = \mathtt{i}\} \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, t', \beta(S, \alpha)) \land \\ \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \end{array} \right\rangle$$
$$\left\{ \exists S, \alpha.\, a \Rrightarrow ((S, \alpha), (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, \beta(S, \alpha))) \right\}$$
$$\mathtt{return}\ 0;$$
$$\left\{ \exists S, \alpha.\, a \Rrightarrow ((S, \alpha), (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, \beta(S, \alpha))) \right\}$$
$$\left\langle \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, \beta(S, \alpha)) * [\mathrm{H}]_a \right\rangle$$
$$\left\langle \mathsf{HMap}(s, \mathtt{x}, (S \setminus \{(\mathtt{k}, \_)\}) \uplus \{(\mathtt{k}, \mathtt{v})\}, \beta(S, \alpha)) \right\rangle$$

Fig. 29: Proof of total correctness of `HMapAdd` function.

$\forall\beta.$
$\mathbb{V}S, \alpha.$
$\langle \mathsf{HMap}(s, \mathtt{x}, S, \alpha) \land \alpha > \beta(S, \alpha) \rangle$
abstract; substitute $s = (a, n, arr, s')$
$\langle \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * [\mathrm{H}]_a \land \alpha > \beta(S, \alpha) \rangle$
make atomic
$a : (S, \alpha) \rightsquigarrow (S \setminus \{(\mathtt{k}, \_)\}, \beta(S, \alpha)) \vdash_\tau$
$\{\exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha)\}$
open region
$\mathbb{V}S, \alpha$
$\left\langle x \mapsto n * \bigotimes_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \right\rangle$
$\mathtt{n} := [\mathtt{x}];$
$\left\langle x \mapsto n * \bigotimes_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \land \mathtt{n} = n \right\rangle$
$\{\exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha)\}$
$\mathtt{i} := \mathtt{k} \,\%\, \mathtt{n};$
$\{\exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{i} = \mathtt{k}\%\mathtt{n}\}$
open region
$\mathbb{V}S, \alpha$
$\left\langle x \mapsto n * \bigotimes_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \right\rangle$
$\mathtt{l} := [\mathtt{x} + \mathtt{i} + 1];$
$\left\langle x \mapsto n * \bigotimes_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k, v) | (k, v) \in S, k\%n = i\}, t, \alpha)) \land \mathtt{l} = arr(\mathtt{i}) \right\rangle$
$\{\exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(S, \alpha) \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i})\}$
update region
$\mathbb{V}S, t, \alpha$
$\langle \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, \{(k, v) | (k, v) \in S, k\%n = \mathtt{i}\}, t, \alpha) \land \alpha > \beta(S, \alpha) \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \rangle$
$\mathtt{b} := \mathsf{LLMapRemove}(\mathtt{l}, \mathtt{k});$
$\left\langle \begin{pmatrix} \underline{\textbf{if}}\ (\mathtt{k}, \_) \in S \\ \underline{\textbf{then}}\ \exists t'.\, \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, \{(k, v) | (k, v) \in S, k\%n = \mathtt{i}\} \setminus \{(\mathtt{k}, \_)\}, t', \beta(S, \alpha)) \land \mathtt{b} = 1 \\ \underline{\textbf{else}}\ \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, \{(k, v) | (k, v) \in S, k\%n = \mathtt{i}\}, t, \alpha) \land \mathtt{b} = 0 \end{pmatrix} \land \mathtt{i} = \mathtt{k}\%\mathtt{n} \land \mathtt{l} = arr(\mathtt{i}) \right\rangle$
$\{a \Rrightarrow ((S, \alpha), (S \setminus \{(\mathtt{k}, \_)\}, \beta(S, \alpha))) \land (\underline{\textbf{if}}\ (\mathtt{k}, \_) \in S\ \underline{\textbf{then}}\ \mathtt{b} = 1\ \underline{\textbf{else}}\ \mathtt{b} = 0)\}$
$\mathtt{return}\ \mathtt{b};$
$\{a \Rrightarrow ((S, \alpha), (S \setminus \{(\mathtt{k}, \_)\}, \beta(S, \alpha))) \land (\underline{\textbf{if}}\ (\mathtt{k}, \_) \in S\ \underline{\textbf{then}}\ \mathsf{ret} = 1\ \underline{\textbf{else}}\ \mathsf{ret} = 0)\}$
$\langle \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * [\mathrm{H}]_a \land \underline{\textbf{if}}\ (\mathtt{k}, \_) \in S\ \underline{\textbf{then}}\ \mathsf{ret} = 1\ \underline{\textbf{else}}\ \mathsf{ret} = 0 \rangle$
$\langle \mathsf{HMap}(s, \mathtt{x}, S \setminus \{(\mathtt{k}, \_)\}, \beta(S, \alpha)) \land \underline{\textbf{if}}\ (\mathtt{k}, \_) \in S\ \underline{\textbf{then}}\ \mathsf{ret} = 1\ \underline{\textbf{else}}\ \mathsf{ret} = 0 \rangle$

Fig. 30: Proof of total correctness of `HMapRemove` function.

abstract; quantify $s = (a, n, arr, s')$

make atomic

$$\mathbb{V} S, \alpha.$$
$$\left\langle \mathsf{HMap}(s, \mathtt{x}, S, \alpha) \right\rangle$$
$$\left\langle \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * [\mathrm{H}]_a \right\rangle$$
$$a : (S, \alpha) \rightsquigarrow (S, \alpha) \vdash_\tau$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * a \Rrightarrow \blacklozenge \right\}$$

open region

$$\mathbb{V} S, \alpha$$
$$\left\langle x \mapsto n * \mathop{\circledast}_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k,v) | (k,v) \in S, k\%n = i\}, t, \alpha)) \right\rangle$$
$$\mathtt{n} := [\mathtt{x}];$$
$$\left\langle x \mapsto n * \mathop{\circledast}_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k,v) | (k,v) \in S, k\%n = i\}, t, \alpha) \wedge \mathtt{n} = n \right\rangle$$

$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \right\}$$
$$\mathtt{i} := \mathtt{k} \,\%\, \mathtt{n};$$
$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{i} = \mathtt{k}\%\mathtt{n} \right\}$$

open region

$$\mathbb{V} S, \alpha$$
$$\left\langle x \mapsto n * \mathop{\circledast}_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k,v) | (k,v) \in S, k\%n = i\}, t, \alpha)) \right\rangle$$
$$\mathtt{l} := [\mathtt{x} + \mathtt{i} + 1];$$
$$\left\langle \begin{array}{l} x \mapsto n * \mathop{\circledast}_{i=0}^{n-1} (x + i + 1 \mapsto arr(i) * \exists t.\, \mathsf{LLMap}(s'(i), arr(i), \{(k,v) | (k,v) \in S, k\%n = i\}, t, \alpha)) \wedge \\ \mathtt{l} = arr(\mathtt{i}) \end{array} \right\rangle$$

$$\left\{ \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, \mathtt{n}, arr, S, \alpha) * a \Rrightarrow \blacklozenge \wedge \mathtt{i} = \mathtt{k}\%\mathtt{n} \wedge \mathtt{l} = arr(\mathtt{i}) \right\}$$

update region

$$\mathbb{V} S, t, \alpha$$
$$\left\langle \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, \{(k,v) | (k,v) \in S, k\%\mathtt{n} = \mathtt{i}\}, t, \alpha) \wedge \mathtt{i} = \mathtt{k}\%\mathtt{n} \wedge \mathtt{l} = arr(\mathtt{i}) \right\rangle$$
$$\mathtt{v} := \mathsf{LLMapLookup}(\mathtt{l}, \mathtt{k});$$
$$\left\langle \begin{array}{l} \exists S'.\ \mathsf{LLMap}(s'(\mathtt{i}), \mathtt{l}, S', t, \alpha) \wedge (((\mathtt{k}, \mathtt{v}) \in S') \vee (\mathtt{v} = 0 \wedge (\mathtt{k}, \_) \notin S')) \wedge \\ S' = \{(k,v) | (k,v) \in S, k\%\mathtt{n} = \mathtt{i}\} \wedge \mathtt{i} = \mathtt{k}\%\mathtt{n} \wedge \mathtt{l} = arr(\mathtt{i}) \end{array} \right\rangle$$

$$\left\{ \exists S, \alpha.\, a \Rrightarrow ((S, \alpha), (S, \alpha)) \wedge (((\mathtt{k}, \mathtt{v}) \in S) \vee (\mathtt{v} = 0 \wedge (\mathtt{k}, \_) \notin S)) \right\}$$
$$\mathtt{return\ v};$$
$$\left\{ \exists S, \alpha.\, a \Rrightarrow ((S, \alpha), (S, \alpha)) \wedge (((\mathtt{k}, \mathsf{ret}) \in S) \vee (\mathsf{ret} = 0 \wedge (\mathtt{k}, \_) \notin S)) \right\}$$
$$\left\langle \exists S, \alpha.\, \mathbf{HashMap}_a(s', \mathtt{x}, n, arr, S, \alpha) * [\mathrm{H}]_a \wedge (((\mathtt{k}, \mathsf{ret}) \in S) \vee (\mathsf{ret} = 0 \wedge (\mathtt{k}, \_) \notin S)) \right\rangle$$
$$\left\langle \mathsf{HMap}(s, \mathtt{x}, S, \alpha) \wedge (((\mathtt{k}, \mathsf{ret}) \in S) \vee (\mathsf{ret} = 0 \wedge (\mathtt{k}, \_) \notin S)) \right\rangle$$

Fig. 31: Proof of total correctness of `HMapLookup` function.

E Hash Set

E.1 Code

```
function makeHashSet() {
  x := makeHMap();
  return x;
}

function add(x, v) {
  HMapAdd(x, v, 1);
}

function contains(x, v) {
  b := HMapLookup(x, v);
  return b;
}

function remove(x, v) {
    b := HMapRemove(x, v);
    return b;
}
```

Fig. 32: Hash Set operations.

E.2 Specifications

$$\vdash_\tau \left\{\mathsf{emp}\right\} \ \mathtt{makeHashSet}() \ \left\{\exists s.\, \mathsf{HashSet}(s, \mathsf{ret}, \emptyset, \alpha)\right\}$$

$$\vdash_\tau \mathbb{W} vs, \alpha. \left\langle \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \right\rangle \ \mathtt{contains}(\mathtt{x}, \mathtt{v}) \ \left\langle \begin{array}{c} \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \wedge \\ \underline{\mathbf{if}} \ \mathtt{v} \in vs \ \underline{\mathbf{then}} \ \mathsf{ret} = 1 \ \underline{\mathbf{else}} \ \mathsf{ret} = 0 \end{array} \right\rangle$$

$$\forall \beta.\ \vdash_\tau \mathbb{W} vs, \alpha. \left\langle \begin{array}{c} \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \wedge \\ \alpha > \beta(vs, \alpha) \end{array} \right\rangle \ \mathtt{add}(\mathtt{x}, \mathtt{v}) \ \left\langle \mathsf{HashSet}(s, \mathtt{x}, \{\mathtt{v}\} \cup vs, \beta(vs, \alpha)) \right\rangle$$

$$\forall \beta.\ \vdash_\tau \mathbb{W} vs, \alpha. \left\langle \begin{array}{c} \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \wedge \\ \alpha > \beta(vs, \alpha) \end{array} \right\rangle \ \mathtt{remove}(\mathtt{x}, \mathtt{v}) \ \left\langle \begin{array}{c} \mathsf{HashSet}(s, \mathtt{x}, vs \setminus \{\mathtt{v}\}, \beta(vs, \alpha)) \wedge \\ \underline{\mathbf{if}} \ \mathtt{v} \in vs \ \underline{\mathbf{then}} \ \mathsf{ret} = 1 \ \underline{\mathbf{else}} \ \mathsf{ret} = 0 \end{array} \right\rangle$$

Fig. 33: Hash Set operation specifications.

E.3 Definitions

$$I(\mathbf{HSet}_a(s, x, vs, \alpha)) \triangleq HMap(s, x, vs \times \{1\}, \alpha)$$

$$HashSet((a, s'), x, vs, \alpha) \triangleq \mathbf{HSet}_a(s', x, vs, \alpha) * [\mathrm{I}]_a$$

$$\mathrm{I} : \forall v, vs, \alpha, \beta < \alpha.\ (vs, \alpha) \rightsquigarrow (vs \cup \{v\}, \beta)$$
$$\mathrm{I} : \forall v, vs, \alpha, \beta < \alpha.\ (vs, \alpha) \rightsquigarrow (vs \setminus \{v\}, \beta)$$
$$\mathrm{I} : \forall vs, \alpha, \beta < \alpha.\ (vs, \alpha) \rightsquigarrow (vs, \beta)$$

E.4 Proofs

$$
\begin{array}{l}
\mathbb{V} vs, \alpha. \\
\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \right\rangle \\
\quad \text{abstract; substitute } s = (a, s') \\
\quad \left\langle \mathbf{HSet}_a(s', x, vs, \alpha) * [\mathrm{I}]_a \right\rangle \\
\quad\quad \text{make atomic} \\
\quad\quad \left\{ \exists vs, \alpha.\ \mathbf{HSet}_a(s', x, vs, \alpha) * a \Rightarrow \blacklozenge \right\} \\
\quad\quad\quad \text{update region} \\
\quad\quad\quad \mathbb{V} vs, \alpha. \\
\quad\quad\quad \left\langle HMap(s', x, vs \times \{1\}, \alpha) \right\rangle \\
\quad\quad\quad \mathtt{b} := \mathtt{HMapLookup}(\mathtt{x}, \mathtt{v}); \\
\quad\quad\quad \left\langle HMap(s', x, vs \times \{1\}, \alpha) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathtt{b} = 1\ \underline{\mathbf{else}}\ \mathtt{b} = 0 \right\rangle \\
\quad\quad \left\{ \exists vs, \alpha.\ a \Rightarrow ((vs, \alpha), (vs, \alpha)) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathtt{b} = 1\ \underline{\mathbf{else}}\ \mathtt{b} = 0 \right\} \\
\quad\quad \mathtt{return\ b}; \\
\quad\quad \left\{ \exists vs, \alpha.\ a \Rightarrow ((vs, \alpha), (vs, \alpha)) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\} \\
\quad \left\langle \mathbf{HSet}_a(s', x, vs, \alpha) * [\mathrm{I}]_a \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\rangle \\
\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\rangle
\end{array}
$$

Fig. 34: Proof of total correctness of Hash Set `contains` operation.

$$
\begin{array}{l}
\forall \beta. \\
\mathbb{V} vs, \alpha. \\
\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \land \alpha > \beta(vs, \alpha) \right\rangle \\
\quad \text{abstract; substitute } s = (a, s') \\
\quad \left\langle \mathbf{HSet}_a(s', x, vs, \alpha) * [\mathrm{I}]_a \land \alpha > \beta(vs, \alpha) \right\rangle \\
\quad\quad \text{use atomic} \\
\quad\quad \mathbb{V} vs, \alpha. \\
\quad\quad \left\langle HMap(s', x, vs \times \{1\}, \alpha) \land \alpha > \beta(vs, \alpha) \right\rangle \\
\quad\quad \mathtt{HMapAdd}(\mathtt{x}, \mathtt{v}, 1); \\
\quad\quad \left\langle HMap(s', x, (vs \cup \{\mathtt{v}\}) \times \{1\}, \beta(vs, \alpha)) \right\rangle \\
\quad \left\langle \mathbf{HSet}_a(s', x, vs \cup \{\mathtt{v}\}, \beta(vs, \alpha)) * [\mathrm{I}]_a \right\rangle \\
\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs \cup \{\mathtt{v}\}, \beta(vs, \alpha)) \right\rangle
\end{array}
$$

Fig. 35: Proof of total correctness of Hash Set `add` operation.

$\forall \beta.$

$\mathbb{V} vs, \alpha.$

$\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs, \alpha) \land \alpha > \beta(vs, \alpha) \right\rangle$

abstract; substitute $s = (a, s')$

$\left\langle \mathbf{HSet}_a(s', x, vs, \alpha) * [\mathrm{I}]_a \land \alpha > \beta(vs, \alpha) \right\rangle$

make atomic

$\left\{ \exists vs, \alpha.\ \mathbf{HSet}_a(s', x, vs, \alpha) * a \Rrightarrow \blacklozenge \land \alpha > \beta(vs, \alpha) \right\}$

update region

$\mathbb{V} vs, \alpha.$

$\left\langle HMap(s', x, vs \times \{1\}, \alpha) \land \alpha > \beta(vs, \alpha) \right\rangle$

```
b := HMapRemove(x, v);
```

$\left\langle \begin{array}{l} HMap(s', x, (vs \setminus \{(\mathtt{v}, \_)\}) \times \{1\}, \alpha) \land \\ \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathtt{b} = 1\ \underline{\mathbf{else}}\ \mathtt{b} = 0 \end{array} \right\rangle$

$\left\{ \exists vs, \alpha.\ a \Rrightarrow ((vs, \alpha), (vs \setminus \{\mathtt{v}\}, \beta(vs, \alpha))) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathtt{b} = 1\ \underline{\mathbf{else}}\ \mathtt{b} = 0 \right\}$

```
return b;
```

$\left\{ \exists vs, \alpha.\ a \Rrightarrow ((vs, \alpha), (vs \setminus \{\mathtt{v}\}, \beta(vs, \alpha))) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\}$

$\left\langle \mathbf{HSet}_a(s', x, vs \setminus \{\mathtt{v}\}, \beta(vs, \alpha)) * [\mathrm{I}]_a \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\rangle$

$\left\langle \mathsf{HashSet}(s, \mathtt{x}, vs \setminus \{\mathtt{v}\}, \beta(vs, \alpha)) \land \underline{\mathbf{if}}\ \mathtt{v} \in vs\ \underline{\mathbf{then}}\ \mathsf{ret} = 1\ \underline{\mathbf{else}}\ \mathsf{ret} = 0 \right\rangle$

Fig. 36: Proof of total correctness of Hash Set `remove` operation.

F Module dependencies

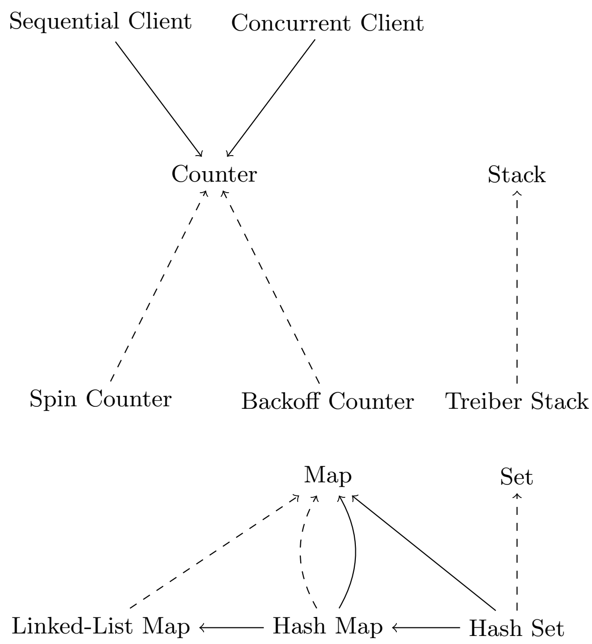

Fig. 37: Dependencies between module implementation, abstract modules and clients in this paper.