A Logic for Proving Total Correctness of Blocking Algorithms

Conrad Watt

Abstract

We present Blocking-Total TaDA (B-TT), the first program logic able to prove total correctness of general blocking operations in a module-client setting.

Formal verification has enjoyed something of a renaissance recently with the advent of new logics capable of reasoning about fine-grained concurrency with an exactness not previously possible. In particular, new techniques which can abstract concurrent heap state mutation into transformations over discrete state systems have finally allowed us to realise a paradigm of "modular" concurrent proofs, where implementation details of operations such as counter increments can be fully abstracted into logical assertions for use in building proofs of more complex programs.

However, such program logics are normally concerned with only *partial* correctness of concurrent programs, a property which does not guarantee termination. Furthermore, where termination is considered, it is less common still that "blocking" programs, where threads may not terminate in isolation, are handled.

We examine the best contributions to reasoning about the total correctness of blocking programs, and, finding them inadequate, extend Total TaDA, an existing logic for total correctness of non-blocking programs, so that it can reason about blocking.

Acknowledgements

A heartfelt thanks to Philippa Gardner and Julian Sutherland for their enthusiastic support throughout every stage of this project. Thanks also to Mark Wheelhouse for his input during the early stages, and my flatmates Alberto and Kacper for their companionship during the final stretch.

Contents

1 Introduction

For years, formal verification languished in the uncomfortable position of being almost unable to tractably extract interesting results about the behaviour of low-level, heap manipulating languages in a concurrent setting. Instead of retreating to a world of expressive type systems and higher level synchronisation primitives, a few brave souls ploughed ahead, steadily iterating program logics describing pointer arithmetic and direct heap meddling in a concurrent world. It is only within the last few years that these efforts can be considered to have truly borne fruit, in the form of the first program logics capable of reasoning expressively about programs exhibiting low-level, fine-grained concurrency.

Over the course of this report, we will follow one thread of this journey, beginning with the original work of Tony Hoare, before tackling the next challenge; proving termination of fine-grained concurrent programs.

The halting problem casts a long shadow over logics attempting to decide termination, and many program logics (concurrent or otherwise) only attempt to prove “partial correctness”. A partial correctness result involves proving properties of a program on the assumption that it terminates, and makes no guarantees on its behaviour if this assumption proves to be incorrect.

For logics dealing with low-level, fine-grained concurrency, even interesting partial correctness results have been hard-won. Only a small number of logics attempt to go futher, into the realm of “total correctness”, guaranteeing termination. These logics suffer from several limitations, and we show in this project that it is possible to do better.

1.1 Contributions

We present Blocking-Total TaDA (B-TT), a new program logic capable of proving termination of concurrent module-client programs in an abstract and compositional way. We will motivate the logic’s expressive power with respect to other logics in the same space, and give a full formal model and partial proof of soundness.

$$\frac{\{P\}\ C_1\ \{Q\} \quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}\ \textit{Sequence}$$

Figure 1: An example derivation rule in Hoare logic. Establishing the triples above the bar allows us to derive the triple below.

2 Background

2.1 Hoare logic

Hoare logic [1] is one of the original systems designed to facilitate formal reasoning about computer programs. Specifying a program $\mathbb{C}$ in Hoare logic involves the derivation of "Hoare triples" of the form $\left\{P\right\}\ \mathbb{C}\ \left\{Q\right\}$, where P and Q are logical assertions about program state. A Hoare triple of this form can be intuitively understood as meaning "if P holds, executing $\mathbb{C}$ to completion will establish Q".

Hoare's original proof system for establishing these triples made no guarantees about the termination of $\mathbb{C}$; this is "partial correctness", the caveat being that nothing is asserted about the program's behaviour if it fails to terminate. In addition, the original language over which Hoare triples were established was exceedingly primitive; its "heap" consisted of only variables with integer values, and the only operations allowed over them were non-side-effecting arithmetic expressions and simple assignment. Attempts to extend the proof system to richer languages, especially ones allowing heap manipulation (for example, through pointers) met with considerable difficulty, since operations that touched memory could potentially reach any point of the heap through arbitrary pointers, and aliasing could mean that previous assertions about parts of the heap might not be preserved.

2.2 Separation logic

Separation logic [2] is a relatively recent development in reasoning about heap-manipulating programs. The assertion logic of Hoare triples is extended with operators for describing heaps, the most important of which are "points to", written $\mapsto$ and "separating conjunction", written $*$. The assertion $x \mapsto v$ describes a heap containing (just!) the value v at address x. The assertion

$P * Q$ describes a heap which can be split into two disjoint subheaps, one satisfying P and the other satisfying Q. Therefore the assertion $x \mapsto 1 * y \mapsto 2$ describes a heap containing the values 1 and 2 at addresses x and y respectively.

The key power of separation logic is that it intuitively describes local heap update; the "frame" inference rule allows us to take any heap-modifying command and infer that it also works when the original heap is extended arbitrarily with unrelated content using $*$. In particular, due to the definition of $*$, we know that the frame is a disjoint area of memory, something which was not scalably expressible before.

$$\frac{\{P\}\ \mathbb{C}\ \{Q\} \quad free(R) \cup modset(\mathbb{C}) = \emptyset}{\{P * R\}\ \mathbb{C}\ \{Q * R\}}\ Frame$$

Figure 2: The frame rule of separation logic. The side condition (often elided) constricts the frame R to reference no variable modified by C.

2.3 Proof notation

Systems based in Hoare logic often have their proof rules presented in a natural deduction style, with preconditions of a derived triple forming a tree structure. However, the typeography of such proofs breaks down with programs of more than a few lines. It is therefore customary to present program proof examples as "sketches", which share some similarities to the Fitch style of natural deduction. An example is shown in figure 3. Compare the (mostly) full proof tree to the two sketch style proofs. In both, applications of the sequence rule are shown in-line, and the names of axioms are elided. In the more detailed sketch, applications of the frame rule are shown explicitly, with the proof indented to the right of the frame bar forming the antecedent of the frame rule.

2.4 Modular verification

An ideal (Hoare triple based) proof system should be one which facilitates the following arrangement; a library or module creator proves specifications (establishes triples) for each of their functions, and any later user (or "client") of the module wishing to prove their own code may take the behaviour of a

$$\dfrac{\dfrac{\dfrac{}{\{\mathsf{emp}\}\ \mathtt{x := new(1)}\ \{\mathtt{x} \mapsto \_\}}\,\mathit{new} \qquad \dfrac{\dfrac{}{\{\mathsf{emp}\}\ \mathtt{y := new(1)}\ \{\mathtt{y} \mapsto \_\}}\,\mathit{new}}{\{\mathtt{x} \mapsto \_\}\ \mathtt{y := new(1)}\ \{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}}\,\mathit{frame}}{\{\mathsf{emp}\}\ \mathtt{x := new(1);\ y := new(1)}\ \{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}}\,\mathit{sequence} \qquad \dfrac{\dfrac{}{\{\mathtt{x} \mapsto \_\}\ [\mathtt{x}] := 4\ \{\mathtt{x} \mapsto 4\}}\,\mathit{mutate}}{\{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}\ [\mathtt{x}] := 4\ \{\mathtt{x} \mapsto 4 * \mathtt{y} \mapsto \_\}}\,\mathit{frame}}{\{\mathsf{emp}\}\ \mathtt{x := new(1);\ y := new(1);}\ [\mathtt{x}] := 4\ \{\mathtt{x} \mapsto 4 * \mathtt{y} \mapsto \_\}}\,\mathit{sequence}$$

$$\begin{array}{l}
\{\mathsf{emp}\}\\
{[\mathtt{x}]} := \mathtt{new}(1);\\
\{\mathtt{x} \mapsto \_\}\\
\text{frame}\left|\begin{array}{l}\{\mathsf{emp}\}\\ {[\mathtt{y}]} := \mathtt{new}(1);\\ \{\mathtt{y} \mapsto \_\}\end{array}\right.\\
\{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}\\
\text{frame}\left|\begin{array}{l}\{\mathtt{x} \mapsto \_\}\\ {[\mathtt{x}]} := 4\\ \{\mathtt{x} \mapsto 4\}\end{array}\right.\\
\{\mathtt{x} \mapsto 4 * \mathtt{y} \mapsto \_\}
\end{array}
\qquad\qquad
\begin{array}{l}
\{\mathsf{emp}\}\\
{[\mathtt{x}]} := \mathtt{new}(1);\\
\{\mathtt{x} \mapsto \_\}\\
{[\mathtt{y}]} := \mathtt{new}(1);\\
\{\mathtt{x} \mapsto \_ * \mathtt{y} \mapsto \_\}\\
{[\mathtt{x}]} := 4\\
\{\mathtt{x} \mapsto 4 * \mathtt{y} \mapsto \_\}
\end{array}$$

Figure 3: Three representations of the same separation logic proof (using slightly simplified rules).

call to a module function as axiomatic. In this way, only the client's own code must be proven. See Fig. 4 for an example. The proofs for a module containing the functions `oneGreater` and `twoGreater` are given, and then a client using this module uses the only the resulting specifications to prove `threeGreater`. A "proof sketch" style is used to outline the derivation steps. Each internal assertion corresponds to a mid-condition involved in sequential composition (see fig. 1). Notice that `twoGreater` could be implemented directly instead of relying on `oneGreater`, without affecting the client's proof of `threeGreater`. The only thing that is exposed to the client is the specification of the two module functions.

2.5 Predicates

The pre and post conditions of Hoare triples must often encode information about the state of objects in memory. For example code to append an item to a list needs a precondition asserting that such a list is allocated and has the appropriate structure. The formal details of how various logics define such module implementation:

```
$\{x = n\}$
function oneGreater(x) {
  $\{x = n\}$
  y := x + 1;
  $\{y = n + 1\}$
  return y;
}
$\{ret = n + 1\}$

$\{x = n\}$
function twoGreater(x) {
  $\{x = n\}$
  y := oneGreater(x);
  $\{y = n + 1\}$
  z := oneGreater(y);
  $\{z = (n + 1) + 1\}$
  return z;
}
$\{ret = n + 2\}$
```

module specification:

```
$\{x = n\}$ oneGreater $\{ret = n + 1\}$
$\{x = n\}$ twoGreater $\{ret = n + 2\}$
```

client implementation:

```
$\{x = n\}$
function threeGreater(x) {
  $\{x = n\}$
  y := twoGreater(x);
  $\{y = n + 2\}$
  z := oneGreater(y);
  $\{z = (n + 2) + 1\}$
  return z;
}
$\{ret = n + 3\}$
```

Figure 4: `oneGreater` and `twoGreater` used to prove `threeGreater`.

object assertions vary, but they generally share a similar notation.

$$\begin{aligned}
&\texttt{Lock}(\texttt{x}, \texttt{s}) \triangleq \texttt{x} \mapsto s \wedge (s = 0 \vee s = 1) \\
&\texttt{List}(\texttt{y}, \alpha) \triangleq (\texttt{y} \mapsto \text{null} \wedge \alpha = []) \vee (\exists z, a, \alpha'.\ \alpha = a : \alpha' \wedge y \mapsto z * \texttt{List}(z, \alpha'))
\end{aligned}$$

Above are two example predicates, describing a lock at location x with state s, and a list at y containing the elements α, as well as potential definitions in basic separation logic. Note that `List`'s definition is inductive if α is non-empty, and also that under these definitions the assertion $\{\texttt{Lock}(\texttt{x}, 0) * \texttt{List}(\texttt{x}, [0, 1])\}$, describing a lock and list with the same address **x**, would be equivalent to $\bot$ since $*$ enforces that both predicates must describe entirely disjoint heaps.

2.6 Historical attempts at concurrency

Hoare logic and separation logic were designed to deal with sequential programs which executed on the assumption that they had exclusive rights to the memory they touched. Many of the derivation rules of both systems are unsound if this assumption is broken. For example, the sequential composition rule of figure 1 is not correct if other code can modify the heap (potentially invalidating Q) in between the execution of C_1 and C_2. Several attempts have been made to extend Hoare logic for concurrency, and several of these extensions have been ported to separation logic, with mixed success.

2.6.1 Owicki-Gries

The prototypical proof technique for concurrent programs is the Owicki-Gries method [3]. Building on top of the initial Hoare system, the semantics of the language are extended with parallel composition, denoting two code fragments executing in parallel, written $C_1 \mid\mid C_2$. The two sequential code fragments undergoing parallel composition are commonly referred to as threads in the literature. A triple for each thread is independently derived under the old, sequential rules. Then, a triple for the parallel composed code can be derived using the parallel composition rule (fig. 5).

The non-interference side condition here is onerous. Recall the composition rule of figure 1. Ordinary sequential composition can be thought of as proving the “extended triple” $\{P\}\ C_1\ \{Q\}\ C_2\ \{R\}$, and then eliding Q to produce $\{P\}\ C_1; C_2\ \{R\}$. An individual thread will be made up of several such sequential compositions; $\{P_1\}\ C_1\ \{P_2\}\ C_2\ \{P_3\}...C_n\ \{P_n\}$ for a straight line program, but potentially with a more complex structure in the case of branching execution paths e.g. due to the if statement. The mid-conditions of each thread are P_2 to P_{n-1}. The non-interference side condition mandates that, given two threads, C_1 and C_2, for each mid-condition P_k of either thread, for every atomic C'_k anywhere in the other thread, $\{P_k\}\ C'_k\ \{P'_k\}$ implies that $P'_k \Rightarrow P_k$.

Intuitively, this can be understood as encoding that there is nothing the either thread can do to invalidate the other thread’s reasoning; each of their inference steps are invariant against every command any other thread could run at any time. Obviously there is a combinatorial explosion when composing more than two threads at once. Additionally, this scheme requires that it is possible to break the operations of a thread into atomic C_ks. A program which only knows the functions it calls by their specifications would be unable to do this, and hence such a proof system does not satisfy the ideals of modular verification.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\} \quad \textit{non-interference}}{\{P_1 \wedge P_2\}\ C_1 \mid\mid C_2\ \{Q_1 \wedge Q_2\}}\ \textit{Parallel}$$

Figure 5: The parallel rule of Owicki-Gries

2.6.2 Rely/guarantee

The often intractable task of verifying the Owicki-Gries non-interference side condition could be made easier by describing interference in a more local manner. In this paradigm, every derivation $\left\{P\right\}\ \mathbb{C}\ \left\{Q\right\}$ is paired with two additional components; its "rely" (the environmental interference the proof is robust with respect to), and its "guarantee" (which circumscribes the effect $\mathbb{C}$ itself is allowed to have on other threads) [4].

A proof of a thread C_1 conducted under a particular rely R still holds true if C_1 is composed with any thread C_2, the guarantee of which is a subset of R. This approach allows easy composition of threads in a way that Owicki-Gries cannot.

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\ C_1 \mid\mid C_2\ \{Q_1 \wedge Q_2\}}\ \textit{Parallel}$$

Figure 6: A parallel rule in the rely-guarantee style

2.7 Separation Logic

2.7.1 Concurrent separation logic

Concurrent separation logic [5] extends separation logic with a parallel rule which is analogous to the parallel rule of previous systems, but takes advantage of the ability of separation logic assertions to neatly express disjointness using $*$.

It is possible to interpret separation logic assertions as not just describing heap state, but also encoding an idea of "abstract ownership". For example, the assertion $\{\mathtt{Lock}(\mathtt{x}, 1)\}$ describes that **x** points to a lock heap object with state 1 (locked). However, it can also be interpreted as "I (the current thread) own

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \mid\mid C_2\ \{Q_1 * Q_2\}}\ \mathit{Parallel}$$

Figure 7: The disjoint parallel rule of concurrent separation logic

the lock". Under this view, where threads can only modify resources they own, the parallel rule encodes that if each thread owns only disjoint resources, their actions will never interfere with each other.

This system therefore treats ownership as an all or nothing proposition. However, this does not accurately describe many of the programming idioms we would expect to use in a concurrent setting, where multiple threads can act together to update a shared data structure, This situation is improved on in future extensions.

2.7.2 CAP

CAP [6] (Concurrent Abstract Predicates) is a development from separation logic which introduces new mechanisms for splitting access to shared resources between different threads. The "concurrent counter" is a common motivating example, and a portion of it is reproduced here from a turorial paper [7].

Imagine a counter module, with three operations, `read(x)`, `incr(x)`, and `wkincr(x)`, and the following example implementation (the details of the constructor are elided).

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

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

```
function wkincr(x) {
  r := [x];
  [x] := r + 1;
}
```

All module operations take as their sole argument a counter object at `x` with value n which may naively be represented with the predicate $\mathtt{C}(\mathtt{x}, n)$. `read(x)` simply returns the current value of the counter without modification. `wkincr(x)` increments the value by 1, and returns the old value, but is not robust against other threads changing the value of the counter (since another thread could update the counter in between the read and the write, a classic race condition). $\mathtt{incr}(\mathtt{x})$ is a version of `wkincr` suitable for concurrent use.

The CAS operation is atomic Compare-And-Swap. The value at $\mathtt{x}$ is compared with the value of $\mathtt{r}$. If they match, $\mathtt{x}$ is updated to $\mathtt{r}+1$, and the return value is 1. Otherwise, no update takes place and the return value is 0. The whole operation is considered an atomic action. Intuitively, `incr` avoids race conditions by backing off every time it detects that someone else is incrementing the counter, and tries again.

The question is therefore to determine how to specify these operations. Let us consider naive specifications, as follows.

$$\left\{\mathsf{C}(\mathtt{x},n)\right\}\ \mathtt{read}(\mathtt{x})\ \left\{\mathsf{C}(\mathtt{x},n)\land\mathtt{ret}=n\right\}\qquad\left\{\mathsf{C}(\mathtt{x},n)\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\mathsf{C}(\mathtt{x},n+1)\land\mathtt{ret}=n\right\}$$

$$\left\{\mathsf{C}(\mathtt{x},n)\right\}\ \mathtt{wkincr}(\mathtt{x})\ \left\{\mathsf{C}(\mathtt{x},n+1)\land\mathtt{ret}=n\right\}$$

While this seems to capture the sequential behaviour of the functions, these specifications give us no hope for proving concurrent clients. Even the following simple example cannot be proven with what we have seen so far, since we cannot split the counter into disjoint subheaps for each thread.

$$\begin{array}{c}
\left\{\mathsf{C}(\mathtt{x},0)\right\}\\
\left\{\mathsf{C}(\mathtt{x},0)*{???}\right\}\\
\begin{array}{c|c}
\left\{\mathsf{C}(\mathtt{x},0)\right\} & \left\{???\right\}\\
\mathtt{incr}(\mathtt{x}) & \mathtt{incr}(\mathtt{x})\\
\left\{\mathsf{C}(\mathtt{x},1)\right\} & \left\{???\right\}
\end{array}\\
\left\{\mathsf{C}(\mathtt{x},1)*{???}\right\}\\
\left\{\mathsf{C}(\mathtt{x},2)\right\}
\end{array}$$

One intuition about the unsatisfactory nature of these specifications is that `wkincr` and `incr` have been given identical specifications. Clearly there is something missing that captures the ability of other threads to access the counter concurrently with the operation's execution.

Here we turn to our earlier intuition, that predicates do not merely encode state, but also permission/ownership. What does it mean for a thread to share ownership of a predicate? Clearly it is not true in this case that the threads operate over disjoint heap locations. However, there is a sense in which their contributions are disjoint. Each thread increments the counter by 1, and we wish to conclude that the final value of the counter is the sum of their increments. Similarly, we still want to be able to express that a thread *lacks* ownership of the counter, i.e. does not contain the counter predicate in its precondition, while still allowing multiple threads to make these contributions.

Consider an augmentation of the logic where we allow the following implication to hold; $\mathsf{C}(\mathtt{x}, n) \Rightarrow \mathsf{C}(\mathtt{x}, n) * \mathsf{C}(\mathtt{x}, n)$. Clearly under a naive interpretation of the predicate as a direct description of the heap state, this is ridiculous. If we however start to consider predicates as encoding ownership, this starts to make sense, as it allows a thread owning the counter to share this ownership amongst any parallel threads it spawns. However, this rule would mean that a thread holding $\mathsf{C}(\mathtt{x}, n)$ could no longer be sure that it has exclusive access; the predicate could have been split off from the one held by a parent thread. Under this interpretation, we could not prove the specifications we proposed above since holding the predicate would not prevent other threads from modifying the counter.

The solution is to extend the predicate to also encode a permission fraction π; $\mathsf{C}(\mathtt{x}, n, \pi)$, $0 < \pi \leq 1$. A thread holding a particular permission may split it as follows; $\mathsf{C}(\mathtt{x}, n, \pi_1 + \pi_2) \Leftrightarrow \mathsf{C}(\mathtt{x}, n, \pi_1) * \mathsf{C}(\mathtt{x}, n, \pi_2)$. Threads holding “full permission” (1) have exclusive access to the counter, while a thread holding “fractional permission” must conduct its proof under the assumption that other threads may hold the rest of the fraction.

We can prove the following specifications for the counter module functions:

$$\left\{\mathtt{C}(\mathtt{x}, n, \pi)\right\}\ \mathtt{read}(\mathtt{x})\ \left\{\mathtt{C}(\mathtt{x}, n, \pi) \land \mathtt{ret} = n\right\} \qquad \left\{\mathtt{C}(\mathtt{x}, n, 1)\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\mathtt{C}(\mathtt{x}, n+1, 1) \land \mathtt{ret} = n\right\}$$

$$\left\{\mathtt{C}(\mathtt{x}, n, 1)\right\}\ \mathtt{wkincr}(\mathtt{x})\ \left\{\mathtt{C}(\mathtt{x}, n+1, 1) \land \mathtt{ret} = n\right\}$$

This is better, but still not good enough! The only operation this specification allows us to parallelise is `read`. We still cannot handle the example with two `incr` operations in parallel. We can attempt to specify `incr` with only a π permission in the precondition. Unfortunately, it would be impossible to prove that the final value of the counter would be $n + 1$, since other threads could increment the counter between entering the function and reading the initial value. `read` is similarly affected, so the best we can do is the following:

$$\begin{array}{l}
\left\{\mathsf{C}(\mathtt{x}, n, \pi)\right\}\ \mathtt{read}(\mathtt{x})\ \left\{\exists n' \geq n, n'' \geq n'.\ \mathsf{C}(\mathtt{x}, n'', \pi) \wedge \mathtt{ret} = n'\right\} \\
\left\{\mathsf{C}(\mathtt{x}, n, \pi)\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\exists n' \geq n, n'' > n'.\ \mathsf{C}(\mathtt{x}, n'', \pi) \wedge \mathtt{ret} = n'\right\} \\
\left\{\mathsf{C}(\mathtt{x}, n, 1)\right\}\ \mathtt{wkincr}(\mathtt{x})\ \left\{\mathsf{C}(\mathtt{x}, n+1, 1) \wedge \mathtt{ret} = n\right\}
\end{array}$$

But now `read` and `incr` have almost the same spec, and we are still no closer to proving that two parallel increments in fact increase the counter's value by exactly two.

Now we can introduce the final piece of the puzzle. As mentioned before, there is a sense in which each thread's contribution to the counter is disjoint. We can interpret the counter predicate as tracking only the current thread's contribution. In the case that we hold full permission, we have the "true" value of the counter. Now, threads must split their permission in the following fashion; $\mathsf{C}(\mathtt{x}, n+m, \pi_1 + \pi_2) \Leftrightarrow \mathsf{C}(\mathtt{x}, n, \pi_1) * \mathsf{C}(\mathtt{x}, m, \pi_2)$. Finally, we can prove a better specification for `incr`, and the parallel increment example.

$$\left\{\mathsf{C}(\mathtt{x}, n, \pi)\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\mathsf{C}(\mathtt{x}, n+1, \pi) \wedge \mathtt{ret} \geq n\right\}$$

$$\begin{array}{c}
\left\{\mathsf{C}(\mathtt{x}, 0, 1)\right\} \\
\left\{\mathsf{C}(\mathtt{x}, 0, \tfrac{1}{2}) * \mathsf{C}(\mathtt{x}, 0, \tfrac{1}{2})\right\} \\
\begin{array}{l|l}
\left\{\mathsf{C}(\mathtt{x}, 0, \tfrac{1}{2})\right\} & \left\{\mathsf{C}(\mathtt{x}, 0, \tfrac{1}{2})\right\} \\
\mathtt{incr}(\mathtt{x}) & \mathtt{incr}(\mathtt{x}) \\
\left\{\mathsf{C}(\mathtt{x}, 1, \tfrac{1}{2})\right\} & \left\{\mathsf{C}(\mathtt{x}, 1, \tfrac{1}{2})\right\}
\end{array} \\
\left\{\mathsf{C}(\mathtt{x}, 1, \tfrac{1}{2}) * \mathsf{C}(\mathtt{x}, 1, \tfrac{1}{2})\right\} \\
\left\{\mathsf{C}(\mathtt{x}, 2, 1)\right\}
\end{array}$$

The concept of abstract resource ownership is a powerful tool in program verification, but the techniques described above still have significant limitations. A hint of this can be seen in the above specification for `incr`. Because the true value of the counter may be higher than the tracked contribution, we can only conclude that the return value must be $\geq n$. Subsequent systems have incorporated various notions of auxiliary "ghost" state, higher order specification, and linearisability to boost proof power. In particular, the concept of "abstract atomicity" has become central to systems wishing to enable the derivation of expressive concurrent specifications.

object	op	thread
p	`lock()`	A
p	RET	A
p	`unlock()`	A
p	`lock()`	B
p	RET	B
p	RET	A

Figure 8: An example history involving two threads.

2.7.3 Linearisability

Linearisability [8] is a property of certain modules that allows concurrent use of the module functions to be reduced to a sequential application of the same operations. This allows complicated assertions and proofs about concurrent programs to be reduced to ones about their sequential equivalents. Intuitively, a module with the linearisability property guarantees that none of its operations will expose "in progress" state to other threads observing the module.

The original formal definition of linearisability is given in terms of a "history" H. A history describes the execution of a piece of code in terms of a sequence of call and return events in different threads. Note that a call may take effect before its return event is registered. For example, fig.8 describes an execution of a program where thread A locks and unlocks the lock p but B is able to lock the lock before A returns, but presumably after A's `unlock` has changed the lock's state. A history is "sequential" if it begins with a call, and every call is immediately followed by its response without anything in between. Fig.8 is an example of a "concurrent" history, where this condition is not met.

Herlihy uses linearisability to describe a correctness condition for modules which may be used in a concurrent setting. Roughly, a module is "correct" if every possible history made up of its operations is linearisable. This is equivalent to proving that every module operation has exactly one distict atomic time point at which it appears to take effect, its "linearisation point". [9]

2.7.4 Abstraction and abstract atomicity

Whether or not we consider an operation on a module to be atomic can depend on the granularity at which threads observe the module's state changes [10]. We can give a specification for a module in terms of a state transition system.

For example, a lock could be described by the set of states $\{0, 1\}$. The `lock` operation could be described by the transition $0 \rightsquigarrow 1$ and the `unlock` operation could be described by the transition $1 \rightsquigarrow 0$. Roughly, if we can show that an implementation of either of these operations has a unique time point (the linearisation point) at which its transition occurs, we can treat this operation as atomic, since threads inspecting the state of the lock can only see the state before or after the transition (there is no intermediate or "working" state).

Consider the counter module described previously. Naively, its set of states is $\mathbb{N}$, the set of natural numbers. Its `incr` operation embodies the transition $n \rightsquigarrow n + 1$ for some n that is related to the value of the counter at the time the function is called (this is an awkward caveat, but one we will do away with later). Consider an extension of the counter with the operation `incrTwo`, which we implement by calling `incr` twice sequentially. Under the current setup, we cannot consider `incrTwo` an atomic operation; other threads inspecting the value of the counter will see multiple states depending on how far `incrTwo` has executed. There is no simple "before" and "after" point that the operation can be described in terms of.

Now, we will perform the crucial trick. We can wrap the counter in another level of abstraction. Let us define another module, "outerCounter". The outerCounter module has two states: (0) and (> 0), which can be read as "the counter is equal to 0" and "the counter is greater than 0". The `incr` operation for the counter module is *also* an operation over the outerCounter module. It will perform one of the two transitions

$$(0) \rightsquigarrow (> 0) \qquad\qquad (> 0) \rightsquigarrow (> 0)$$

but crucially, it will perform *exactly* one (if it terminates), and therefore we can still treat `incr` as atomic. The $(> 0) \rightsquigarrow (> 0)$ transition, which does not change the state, is what is known as an "abstract skip". Any thread inspecting the state of outerCounter will not be affected as a result of the abstract skip occurring, so we can always choose whether or not to count it as a linearisation point.

What about `incrTwo`? The first `incr` operation will perform one of the two transitions above, however the second `incr` will *always* be an abstract skip. Therefore we can consider the `incrTwo` operation as performing exactly one transition, and therefore it is an atomic operation for the outerCounter module.

```
incr(x);
incr(x);
v := read(x);
if (v > 1) {
  critical
}
```

Figure 9: A trivial program that must execute *critical*.

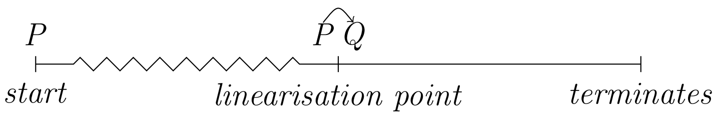
There are a few important points to make here. Any code that can be proven to implement the original counter module can also be proven to implement the outerCounter module. However, the discussed implementation of the `incrTwo` operation can only be considered atomic so long we only reason about the behaviour of every thread in terms of the outerCounter state system. This is important since proving that all the operations of a module behave atomically gives us the linearisablity correctness result as discussed above. For a counter module with `incrTwo` implemented as two `incr` calls, we can either give `incrTwo` a non-atomic specification, and break the linearisablity of the whole module, or we can abstract our view of the state system to the point that `incrTwo` appears linearisable. Obviously for this contrived example, the abstraction is hideously inexpressive (since `incr` would only guarantee to take the state of the module to (> 0), we could not even prove that the code of fig. 9 executes *critical*), but it does motivate that different levels of abstraction give us different guarantees about the atomicity of operations.

Many modern program logics provide ways of encoding the abstract atomicity of an operation. However, there are several different approaches. Iris [11] can reason about the abstract atomicity of an operation using higher order specifications. A history-based approach [12] proves abstract atomicity by reasoning about "subjective" histories. We will concentrate on a system which gives a direct, first-order specification for abstractly atomic operations; TaDA.

2.8 TaDA

TaDA (Time and Data Abstraction) [13] is a program logic which combines the expressive assertions of separation logic with notions of abstract atomicity, in order to prove "atomic specifications". An operation with an atomic specification can be treated as though it executes in a single "step". This is like the linearisability previously discussed.

Atomic specifications are expressed using "atomic triples". In its simplest form, the atomic triple is written $\vdash \mathbb{\forall} x \in X. \left\langle P(x) \right\rangle \; \mathbb{C} \; \left\langle Q(x) \right\rangle$. The meaning of the triple is most intuitively explained using the following diagram.



To explain, an atomic triple expresses that $\mathbb{C}$ contains exactly one linearisation point. P and Q are assertions about the state of shared memory. The precondition P expresses that P will hold from when $\mathbb{C}$ begins execution until its linearisation point. P is parameterised by x, which may vary within the set X up until the linearisation point. The pseudo-quantifier (or "funny for-all") to the left of the assertion is the syntax for this. When the linearisation point occurs, $\mathbb{C}$ will atomically update from P to Q, and Q can use x to refer the the value of x immediately before the linearisation point. After the linearisation point, $\mathbb{C}$ guarantees not to alter shared state any more, but makes no guarantees that Q will remain true (other threads could invalidate it). P may describe a set of possible states, either because it directly describes a disjunction of several potential states, or because of the pseudo-quantified x. The environment is free to alter the shared memory described by P so long as it remains within the set. For now, since we are describing partial correctness, all bets are off if $\mathbb{C}$ fails to terminate.

It important to stress that property embodied here is *abstract* atomicity. It is possible to describe updates to shared memory at various levels of granularity, only some of which may appear to be abstractly atomic. It will now be shown how TaDA formalises atomic update and these different levels of abstraction.

2.8.1 Formalising an atomic update

TaDA owes much to the formalism of CAP, using predicates to abstract ownership (and more generally, the existence) of particular resources. For example, a lock may be described by the abstract predicate $\mathsf{L}(r, \mathtt{x}, s)$, with $s \in \{\top, \bot\}$. $\mathtt{x}$ is, again, the memory location of the lock. r is a parameter which uniquely identifies certain internal components of the module. In some older papers it was elided, making the logic subtly unsound. A simple atomic specification for `unlock` might look something like the following.

$$\vdash \left\langle \mathsf{L}(r, \mathtt{x}, \top) \right\rangle \; \mathtt{unlock} \; \left\langle \mathsf{L}(r, \mathtt{x}, \bot) \right\rangle$$

Such a specification is only possible due to the unique guarantees of the atomic triple. A non-atomic specification could not have this post-condition, as it would have to account of the possibility of the lock being re-locked before `unlock` function returns.

The atomic specification for `lock` is more complicated, and makes use of the pseudo-quantifier.

$$\vdash \mathbb{V} l \in \mathbb{B}. \left\langle \mathsf{L}(r, \mathtt{x}, l) \right\rangle \; \mathtt{lock}(\mathtt{x}) \; \left\langle \mathsf{L}(r, \mathtt{x}, \top) \land \neg l \right\rangle$$

$\mathbb{B}$ is the set of booleans. Because of the pseudo-quantifier, we have encoded that the environment is allowed to arbitrarily lock and unlock the lock until the linearisation point occurs. The $\neg l$ condition in the post-condition enforces that the lock must have been unlocked immediately before the linearisation point (which is when `lock` actually locks the lock).

To prove these specifications against an implementation, it is necessary to provide an interpretation ($\triangleq$ definition) for the abstract predicate. This is done in terms of "regions" and "guards", which further abstract the shared memory underlying the predicate. A region describing the memory of a lock might look something like $\mathbf{Lock}(\mathtt{x}, s)$ with $s \in \{0, 1\}$. Regions are associated with a set of states, and a state transition system that defines the atomic updates that can be made. Here the set of states for **Lock** are $\{0, 1\}$, and the transition system is as follows.

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

The gist here is that to prove an atomic specification for an implementation of `unlock`, we must show that its code makes exactly one of the transitions here across the entire execution of the function. The G parameterisation of the state transition is the "guard". A transition is only allowed if the correct guard is held in the precondition. We will see the power of this later, but for **Lock**, every transition will be guarded by the same guard, G. The interpretation of L in terms of **Lock** is close to trivial here, but this is not always the case. It is provided in fig. 10. Since multiple regions of the same type may be present, the region and its associated guard are given a unique "region identifier" a. The

$$\mathsf{L}(a, \mathtt{x}, \bot) \triangleq \mathbf{Lock}_a(\mathtt{x}, 0) * [\mathrm{G}]_a$$
$$\mathsf{L}(a, \mathtt{x}, \top) \triangleq \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a$$

Figure 10: Interpretation of $\mathsf{L}(a, \mathtt{x}, s)$.

guard is included in the interpretation to encode that holding the predicate means that the thread has permission to perform the region's transitions. In general this is not always the case.

Proving that a specific line of code executes an action in a region's transition system involves the use of two rules which can be considered the powerhouses of TaDA; "make atomic" and "update region".

2.8.2 Make atomic

The TaDA paper initially explains the make atomic rule using a simpler variant which is reproduced in fig. 11. This rule is TaDA's instrument for proving that a block of code can be considered abstractly atomic. We are allowed to conclude that $\mathbb{C}$ can be described by an atomic triple updating , if we are able to prove the following;:

- $\mathbb{C}$ performs exactly one update to the region $\mathbf{t}_a(x)$.
- The local thread holds the correct guards for this action.

. The first restriction is enforced by the unfulfilled "atomicity tracking component" $a \Mapsto \blacklozenge$. The state of a region with region id a may only be altered locally by non-atomic code through the "update region" rule, which requires an unfulfilled atomicity tracking component in the precondition, and produces a "fulfilled" atomicity tracking component which records the transition. Since these tokens are only generated once within a make atomic, we can be sure that only one update has occurred.

The second restriction is enforced by the top antecedent. $\mathcal{T}_{\mathbf{t}}(\mathrm{G})$ defines the set of transitions allowed by the guards G. We pick a subset of these allowed transitions and stick them on the left hand side of the turnstile in a place known as the "atomicity context". The update region rule will only allow a transition that is contained in the atomicity context. Let us consider the proof of a simple implementation of unlock given in fig. 12, given in an elaborated sketch style. The "abstract; quantify a" line unwraps the abstract predicate into its interpretation. The make atomic rule prepares the unfulfilled atomicity

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

Figure 11: Simplified make atomic.

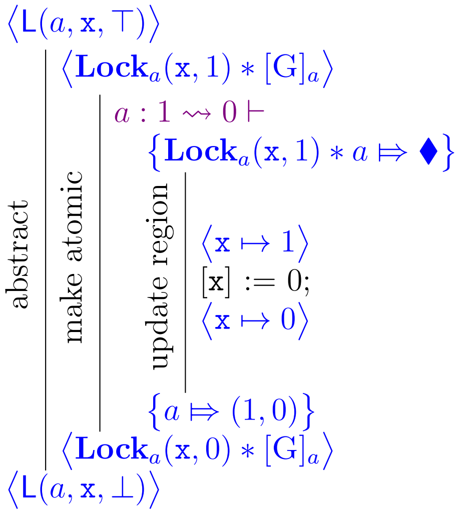

$\langle \mathsf{L}(a, \mathtt{x}, \top) \rangle$
$\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \rangle$
$a : 1 \rightsquigarrow 0 \vdash$
$\{\mathbf{Lock}_a(\mathtt{x}, 1) * a \Rrightarrow \blacklozenge\}$
$\langle \mathtt{x} \mapsto 1 \rangle$
$[\mathtt{x}] := 0;$
$\langle \mathtt{x} \mapsto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

Figure 12: Proof of `unlock`(x).

tracking component, and adds the action to be fulfilled to the atomicity context. Executing the line "[x] := 0;" satisfies the action and therefore fulfills the atomicity tracking component. The make atomic rule can now update the region since we have concluded with a fulfilled atomicity tracking component.

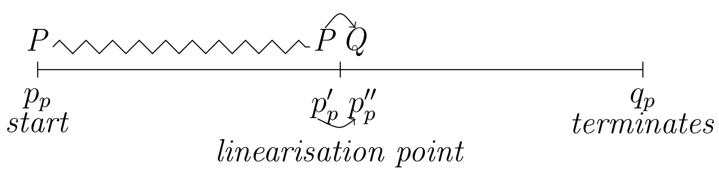
2.8.3 The full TaDA triple

The generalised form of the TaDA atomic triple contains not just information about the state of shared memory, but also "private state" assertions.

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

Here is a visual representation of the transformation of the private state over the course of the execution of $\mathbb{C}$.

The private and public parts of the assertions are separated by a vertical bar. The right hand side is the old assertion we discussed before. The left



hand side of the assertion is called the "private" part. Its meaning is slightly different. Assertions may only be put on the private side if they are *stable* against anything the environment may do. This is defined by the transition system of the various regions. If the local thread does not have exclusive permission to perform actions on the region it is making an assertion over (for example if the region's actions are predicated by guards that other thread's might hold) then the private state cannot contain any assertion that could be invalidated by this action taking place. In the lock example, other threads can possibly hold the guard G, so the private side of an assertion could never contain the assertion $\mathbf{Lock}(\mathtt{x}, 0)$, since another thread could lock the lock and invalidate this.

2.8.4 The non-atomic triple

In TaDA, triples of the form $\{P\}\ \mathbb{C}\ \{Q\}$, which specify the execution of a non-atomic $\mathbb{C}$, are purely syntactic sugar for the atomic triple $\langle P \mid \top \rangle\ \mathbb{C}\ \langle Q \mid \top \rangle$. The assertion effectively states that we have no information at all about linearisation points that occur during the execution of $\mathbb{C}$, and that the assertions P and Q, being in the private side, must be fully stable against any possible region updates. This mirrors a rely-guarantee style stability condition, except the actions the assertions must be stable with respect to are determined by which guards are held locally and the transition systems of the regions.

2.8.5 Update region

The final question is how TaDA ensures that a piece of code satisfies the atomicity tracking component. To fully explain this, we must introduce the concept of "region interpretations". Like an abstract predicate, a region can be "opened up" to reveal a lower level representation. However, this is done in a far more restricted fashion. A region may *only* be opened for a single atomic update, after which it is closed again. This ensures that other threads with access to the region will never observe it in the middle of an update. See fig. 13 for an example. Several rules will reference the region interpretation function

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

Figure 13: Example region interpretation.

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

Figure 14: The update region rule.

I, which need only needs to be defined if the region is opened directly in the proof. The full update region rule is give in fig. 14. This also introduces the last annotation to the left hand side of the turnstile: the “level”. A region can only be opened (via the region interpretation function) if the proof is at the same level of the region ($\mathbf{t}_a^{\lambda}(x)$ where λ is the region's level). The proof's level then decreases by one. This ensures that regions defined in terms of each other cannot be opened in an infinite loop, which would be unsound. In many proof sketches the level will be elided, since it is only important if the proof contains a recursively defined region.

Note that update region must be able to handle both the case where the linearisation point occurs, and the case where it does not. This is represented by the disjunction in both the antecedent and consequent of the update region rule. In the case where the update is a simple memory cell mutation, the “didn't happen” case is trivially false. However this is not the case when the update is a CAS, as can be seen in the lock proof of fig. 15. Notice how the two cases of the CAS neatly mirror the two cases of the update region rule. Once the loop terminates, we know that the CAS has succeeded, and therefore that the region has been updated and the linearisation point has occurred.

2.8.6 Proving a client

TaDA's expressivity comes from its ability to layer abstractions. Interpretations of regions and abstract predicates are permitted to contain further regions and abstract predicates which represent memory at a lower level. In this way, the

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

Figure 15: Proof of `lock(x)`

state space of a higher level region can offer an abstracted or subsetted view of the behaviour of a lower one.

Let us look at a concrete example, which will also serve to illustrate TaDA's weakening rules for using atomic code as part of a non-atomic program. The TaDA style specification for a counter module is given below. Note that the client does not need to know about the interpretations used to prove the counter implementation.

$$\vdash \forall n \in \mathbb{N}. \left\langle \mathtt{C}(a, \mathtt{x}, n) \right\rangle \ \mathtt{read}(\mathtt{x}) \ \left\langle \mathtt{C}(a, \mathtt{x}, n) \land \mathtt{ret} = n \right\rangle$$
$$\vdash \forall n \in \mathbb{N}. \left\langle \mathtt{C}(a, \mathtt{x}, n) \right\rangle \ \mathtt{incr}(\mathtt{x}) \ \left\langle \mathtt{C}(a, \mathtt{x}, n+1) \land \mathtt{ret} = n \right\rangle$$
$$\vdash \left\langle \mathtt{C}(a, \mathtt{x}, n) \right\rangle \ \mathtt{wkincr}(\mathtt{x}) \ \left\langle \mathtt{C}(a, \mathtt{x}, n+1) \land \mathtt{ret} = n \right\rangle$$

Note that for `wkincr`, the lack of pseudoquantifier indicates that this specification will only hold when the operation is executed in an environment where the value of the counter does not vary, i.e. where there are no concurrent increments occurring.

We will prove that the following clients both increment an initially zero-valued counter to exactly two.

2.8.7 A sequential client

```
incr(x);
incr(x);
```

We will define a thin wrapping region for `C` in the client, the only effect of which will be to encode that no other thread has permission to increment the counter. Let us call this region **CCounter**. The transition system and guard algebra for the client is as follows:

$$\forall n \in \mathbb{N}.\ \mathrm{H} \ :\ n \rightsquigarrow n+1 \qquad\qquad \mathrm{H} \bullet \mathrm{H} = \bot$$

The equivalence $\mathrm{H} \bullet \mathrm{H} = \bot$ encodes that if one thread holds H, no other thread can, since their composition is equal to $\bot$. A consequence of this is that the guard cannot be split; $\mathrm{H} \Rightarrow \mathrm{H} * \mathrm{H}$ does not hold. This means that if we hold [H] in a non-atomic assertion, we do not need to worry about stability with respect to transitions guarded by [H], since no other thread can make them.

The region interpretation of **CCounter** is as follows:

$$I(\mathbf{CCounter}_a(r, x, n)) \triangleq \mathtt{C}(r, x, n)$$

We define an abstract predicate to represent the client at the counter level, `CC`. Its only effect is to encode the client's view of its exclusive permission to the counter. The interpretation of `CC` is as follows:

$$\mathtt{CC}(a, r, x, n) \triangleq \mathbf{CCounter}_a(r, x, n) * [\mathrm{H}]_a$$

Now we can proceed with the proof, which can be found in fig. 16. Note that if [H] could be held by another thread, the non-atomic line $\{\mathbf{CCounter}_a(r, \mathtt{x}, 1) * [\mathrm{H}]_a\}$ would not be stable since other regions could increment the counter, and would have to be weakened to $\{\exists n \geq 1.\, \mathbf{CCounter}_a(r, \mathtt{x}, n) * [\mathrm{H}]_a\}$, destroying the proof.

```
$\{\mathsf{CC}(a, r, \mathtt{x}, 0)\}$
  abstract
    $\{\mathbf{CCounter}_a(r, \mathtt{x}, 0) * [\mathrm{H}]_a\}$
      weaken
        $\langle \mathbf{CCounter}_a(r, \mathtt{x}, 0) * [\mathrm{H}]_a \rangle$
          use atomic
            $\langle \mathsf{C}(r, \mathtt{x}, 0) \rangle$
            incr(x);
            $\langle \mathsf{C}(r, \mathtt{x}, 1) \rangle$
        $\langle \mathbf{CCounter}_a(r, \mathtt{x}, 1) * [\mathrm{H}]_a \rangle$
    $\{\mathbf{CCounter}_a(r, \mathtt{x}, 1) * [\mathrm{H}]_a\}$
      weaken
        $\langle \mathbf{CCounter}_a(r, \mathtt{x}, 1) * [\mathrm{H}]_a \rangle$
          use atomic
            $\langle \mathsf{C}(r, \mathtt{x}, 1) \rangle$
            incr(x);
            $\langle \mathsf{C}(r, \mathtt{x}, 2) \rangle$
        $\langle \mathbf{CCounter}_a(r, \mathtt{x}, 2) * [\mathrm{H}]_a \rangle$
    $\{\mathbf{CCounter}_a(r, \mathtt{x}, 2) * [\mathrm{H}]_a\}$
$\{\mathsf{CC}(a, r, \mathtt{x}, 2)\}$
```

Figure 16: Sequential counter client proof

2.8.8 A concurrent client

$$\mathtt{incr(x);} \;\Big\|\; \mathtt{incr(x);}$$

We will define a client region **CCounter'** which tracks the updates of each branch separately. Its states will be tuples of (n_1, n_2). The state transition system and guard algebra are as follows.

$$\forall n, n' \in \mathbb{N}.\ \mathrm{J} : (n, n') \rightsquigarrow (n{+}1, n') \qquad \forall n, n' \in \mathbb{N}.\ \mathrm{K} : (n, n') \rightsquigarrow (n, n'{+}1)$$

$$\mathrm{H} = \mathrm{J} \bullet \mathrm{K} \qquad \mathrm{J} \bullet \mathrm{J} = \bot \qquad \mathrm{K} \bullet \mathrm{K} = \bot$$

This guard algebra encodes that each branch will take exclusive permission for updating one half of the “counter”. The region interpretation of **CCounter'** is as follows.

$$I(\mathbf{CCounter'}_a(r, x, n, n')) \triangleq \mathtt{C}(r, x, n + n')$$

The client’s abstract predicate $\mathtt{CC}'$ will be as interpreted as follows:

$$\mathtt{CC}'(a, r, x, n + n') \triangleq \mathbf{CCounter'}_a(r, x, n, n') * [\mathrm{H}]_a$$

$$\{\mathsf{CC}'(a, r, \mathtt{x}, 0)\}$$

abstract

$$\{\mathbf{CCounter'}_a(r, \mathtt{x}, 0, 0) * [\mathrm{H}]_a\}$$

$$\{\mathbf{CCounter'}_a(r, \mathtt{x}, 0, 0) * \mathbf{CCounter'}_a(r, \mathtt{x}, 0, 0) * [\mathrm{J}]_a * [\mathrm{K}]_a\}$$

$$\begin{array}{l||l}
\{\exists n'.\, \mathbf{CCounter'}_a(r, \mathtt{x}, 0, n') * [\mathrm{J}]_a\} & \{\exists n.\, \mathbf{CCounter'}_a(r, \mathtt{x}, n, 0) * [\mathrm{K}]_a\} \\
\quad \text{weaken} & \quad \text{weaken} \\
\quad \forall n' \in \mathbb{N}. & \quad \forall n \in \mathbb{N}. \\
\quad \langle \mathbf{CCounter'}_a(r, \mathtt{x}, 0, n') * [\mathrm{J}]_a \rangle & \quad \langle \mathbf{CCounter'}_a(r, \mathtt{x}, n, 0) * [\mathrm{K}]_a \rangle \\
\qquad \text{use atomic} & \qquad \text{use atomic} \\
\qquad \langle \mathsf{C}(r, \mathtt{x}, n') \rangle & \qquad \langle \mathsf{C}(r, \mathtt{x}, n) \rangle \\
\qquad \mathtt{incr}(\mathtt{x}); & \qquad \mathtt{incr}(\mathtt{x}); \\
\qquad \langle \mathsf{C}(r, \mathtt{x}, n' + 1) \rangle & \qquad \langle \mathsf{C}(r, \mathtt{x}, n + 1) \rangle \\
\quad \langle \mathbf{CCounter'}_a(r, \mathtt{x}, 1, n') * [\mathrm{J}]_a \rangle & \quad \langle \mathbf{CCounter'}_a(r, \mathtt{x}, n, 1) * [\mathrm{K}]_a \rangle \\
\{\exists n'.\, \mathbf{CCounter'}_a(r, \mathtt{x}, 1, n') * [\mathrm{J}]_a\} & \{\exists n.\, \mathbf{CCounter'}_a(r, \mathtt{x}, n, 1) * [\mathrm{K}]_a\}
\end{array}$$

$$\{\exists n, n'.\, \mathbf{CCounter'}_a(r, \mathtt{x}, n, 1) * \mathbf{CCounter'}_a(r, \mathtt{x}, 1, n') * [\mathrm{J}]_a * [\mathrm{K}]_a\}$$

$$\{\mathbf{CCounter'}_a(r, \mathtt{x}, 1, 1) * [\mathrm{H}]_a\}$$

$$\{\mathsf{CC}'(a, r, \mathtt{x}, 2)\}$$

Figure 17: Parallel counter client proof

The proof of the concurrent client can be count in fig. 17. Notice how each individual thread loses information about the other thread's contribution when it only holds its own guard. The region assertions maybe be split freely as seen in the non-atomic lines immediately before the parallel rule, since holding one does not give permission to modify it; this is the job of the guard. After the parallel rule, we know that there is only one possible state of the **CCounter'** region given each thread's knowledge and the guards we hold.

While the client region defined here is very targetted towards proving this example, note that we are still able to use the general form of the module's specification. This is a key power of TaDA. By layering regions and predicates on top of each other, "gimmicky" formalisms may augment more general ones precisely where they are needed by a client's proof, instead of having to introduce them at the level of the module specification.

2.9 Total correctness

As stressed previously, all of the systems described so far only prove partial correctness with respect to a specification. The program the proof is conducted over may loop or recurse infinitely, and in this case the proof makes no guarantees about its behaviour. Termination is a useful property, so it makes sense to

```
x := 10;              x := 10;
while (x > 0) {       while (1) {
  x := x − 1;           x := x − 1;
}                     }
```

Figure 18: Two trivial while programs. The first program can be proven to terminate.

try and prove it! Of course trying to build a proof system that decides the halting problem is doomed to failure (leaving aside that partial correctness is also undecidable in general!), so existing systems *severely* restrict the space of provable programs.

2.9.1 Loop variants

Take a trivial while programming language with no recursion or parallelism. Now the only way a program can fail to terminate is if a loop never has its condition falsified. Our aim is therefore to find a variant for every while loop; a value on a finite decreasing chain that decreases with each loop iteration [14]. If such a value is found, then the loop must terminate, since the value cannot decrease infinitely. See fig. 18 for examples. In the first example, `v` forms a variant for the while loop, since $\texttt{v} \geq 0$ is an invariant of the loop and `v` decreases each iteration. In the second example, even though `v` decreases each iteration it is *not* a variant since it will not decrease finitely. This system is very neat, but clearly restricting ourselves to such a language is not satisfactory.

$$\frac{\forall \gamma \leq \alpha.\, \lambda; \mathcal{A} \vdash_{\tau} \{P(\gamma) \land \mathbb{B}\}\ \mathbb{C}\ \{\exists \beta.\, P(\beta) \land \beta < \alpha\}}{\lambda; \mathcal{A} \vdash_{\tau} \{P(\alpha)\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \{\exists \beta.\, P(\beta) \land \alpha \geq \beta \land \neg \mathbb{B}\}}$$

Figure 19: The while rule of Total TaDA

$$\mathit{Loop}: \boxed{\dfrac{\dfrac{\dfrac{}{\forall \gamma \leq 10.\ \vdash_{\tau} \{\texttt{x} = \gamma\}\ \texttt{x} := \texttt{x} - 1\ \{\texttt{x} = \gamma - 1\}}\ \mathit{assign}}{\forall \gamma \leq 10.\ \vdash_{\tau} \{0 < \texttt{x} \leq 10 \land \texttt{x} = \gamma\}\ \texttt{x} := \texttt{x} - 1\ \{\exists \beta.\, 0 \leq \texttt{x} \leq 10 \land \texttt{x} = \beta \land \beta < \gamma\}}\ \mathit{frame, cons}}{\vdash_{\tau} \{0 \leq \texttt{x} \leq 10 \land \texttt{x} = 10\}\ \texttt{while}\ (\texttt{x} > 0)\ \texttt{x} := \texttt{x} - 1\ \{\exists \beta.\, 0 \leq \texttt{x} \leq 10 \land \texttt{x} = \beta \land 10 \geq \beta \land \texttt{x} \leq 0\}}\ \mathit{while}}$$

$$\dfrac{\dfrac{}{\vdash_{\tau} \{\mathit{emp}\}\ \texttt{x} := 10\ \{\texttt{x} = 0 \land 0 < \texttt{x} \leq 10\}}\ \mathit{assign, cons} \qquad \boxed{\mathit{Loop}}}{\vdash_{\tau} \{\mathit{emp}\}\ \texttt{x} := 10;\ \texttt{while}\ (\texttt{x} > 0)\ \{\texttt{x} := \texttt{x} - 1\}\ \{\texttt{x} = 0\}}\ \mathit{sequence, cons}$$

Figure 20: An instance of the while rule with an instantiated (in)variant.

2.10 Total TaDA

The notion of loop variants can be extended into the concurrent language of TaDA. Total TaDA [15] is an extension to TaDA which proves total correctness. A triple in the Total TaDA system has the same meaning as in TaDA, with the added guarantee that $\mathbb{C}$ will terminate. There are two ways in which a statement in the language of TaDA can fail to terminate; bottomless recursion, and while loops which never have their condition invalidated. The added complication is that, since TaDA is a concurrent language, a while loop condition may be affected by the actions of other threads. In some circumstances, it is still possible to determine a variant for the loop, except now other threads may be the ones causing it to decrement.

The Total TaDA while rule is given in fig. 19. The $P(\gamma)$ assertion describes a loop invariant P which is parameterised by a variant γ which decreases each time the body is executed. A proof of the termination of a simple while loop program is given in tree style in fig. 20. Notice that the invariant contains the assertion $x = \gamma$, which forms the variant for the loop. The initial value of the variant, α, is set to 10. Now we will look at a more complicated example, where the variant of the loop is decreased as a result of actions by the environment, not just the local thread.

2.10.1 Termination of increment

Consider the counter example described previously. For convenience, the `incr` function is reproduced below.

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

We previously discussed a (partial) specification for this function in TaDA; `incr` would be $\vdash \mathbb{V}n \in \mathbb{N}.\left\langle \mathsf{C}(r, \mathtt{x}, n) \right\rangle\ \mathtt{incr}(x)\ \left\langle \mathsf{C}(r, \mathtt{x}, n + 1) \wedge \mathtt{ret} = n \right\rangle$. To prove that the above implementation satisfies this specification, we must give an interpretation of the abstract predicate where the states of the counter are the natural numbers, and `incr` embodies the transition $n \rightsquigarrow n + 1$.

Now we must consider total correctness. Intuitively, what is the termination condition for the loop? So long as only a finite number of increments run in parallel, eventually the CAS will succeed and the loop will terminate. This is because the CAS will only fail if at least one of the `incr` calls succeeded in modifying the counter, thus terminating. Eventually either our local CAS will succeed, or every other `incr` call will succeed first, but then we are guaranteed to succeed immediately after.

This arrangement can be thought of in terms of a "global variant" across the entire counter module. Call the number of $n \rightsquigarrow n + 1$ operations allowed to occur α. Each time we iterate around the loop, there are two possibilities. Either we succeed, decreasing α by 1, or our CAS fails, which means another $n \rightsquigarrow n + 1$ operation has occurred since our last loop iteration, and α must have decreased by 1. Clearly there cannot be an infinite number of increments in parallel, since we would run out of α.

We can encode this intuition into our abstract predicate and counter region. Now the counter predicate will be parameterised by an ordinal number which, as long as it is greater than 0, provides permission to increment the counter. This is enforced by the underlying region's transition system.

The interpretation of the counter predicate C is as follows:

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

The region's transition system is defined as follows:

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

This forces any action incrementing the counter to decrease the ordinal.

The region interpretation is defined as follows:

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

```
$\forall\beta.\ \forall\!\!\!\forall n \in \mathbb{N}, \alpha$
$\langle \mathsf{C}(r, \mathtt{x}, n, \alpha) \wedge \alpha > \beta(n, \alpha) \rangle$
abstract
  $\langle \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * [\mathrm{G}]_r \wedge \alpha > \beta(n, \alpha) \rangle$
  make atomic
    $r : (n, \alpha) \wedge n \in \mathbb{N} \wedge \alpha > \beta(n, \alpha) \leadsto (n+1, \beta(n, \alpha)) \vdash_\tau$
    $\{\exists n, \alpha.\ \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \wedge \alpha > \beta(n, \alpha)\}$
    b := 0;
    $\{\exists n, \alpha.\ \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \wedge \mathtt{b} = 0 \wedge \alpha > \beta(n, \alpha)\}$
    while (b = 0) {
      $\forall\gamma$
      $\{\exists n, \alpha.\ \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \wedge \gamma \geq \alpha > \beta(n, \alpha)\}$
      open region
        $\langle \mathtt{x} \mapsto n \wedge \gamma \geq \alpha > \beta(n, \alpha) \rangle$
          v := [x];
        $\langle \mathtt{x} \mapsto n \wedge \mathtt{v} = n \wedge \gamma \geq \alpha > \beta(n, \alpha) \wedge (n > \mathtt{v} \Rightarrow \gamma > \alpha) \rangle$
      $\{\exists n, \alpha.\ \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \wedge \gamma \geq \alpha > \beta(n, \alpha) \wedge (n > \mathtt{v} \Rightarrow \gamma > \alpha)\}$
      update region
        $\forall\!\!\!\forall n \in \mathbb{N}$
        $\langle \mathtt{x} \mapsto n \wedge \gamma \geq \alpha > \beta(n, \alpha) \wedge (n > \mathtt{v} \Rightarrow \gamma > \alpha) \rangle$
          b := CAS(x, v, v + 1);
        $\left\langle \begin{array}{l} \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \gamma > \alpha \wedge \mathtt{x} \mapsto n \\ \quad \underline{\mathbf{else}}\ \gamma > \alpha \wedge \mathtt{x} \mapsto n + 1 \wedge \mathtt{v} = n \end{array} \right\rangle$
      $\left\{ \begin{array}{l} \exists n, \alpha. \gamma \geq \alpha > \beta(n, \alpha) \wedge \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \mathbf{Counter}_r(\mathtt{x}, n, \alpha) * r \Mapsto \blacklozenge \wedge \gamma > \alpha \\ \quad \underline{\mathbf{else}}\ r \Mapsto (\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha)) \end{array} \right\}$
    }
    $\{\exists n, \alpha. r \Mapsto (n, \alpha), (n + 1, \beta(n, \alpha)) \wedge \mathtt{v} = n\}$
    return v;
    $\{\exists n, \alpha. r \Mapsto (n, \alpha), (n + 1, \beta(n, \alpha)) \wedge \mathtt{ret} = n\}$
  $\langle \mathbf{Counter}_r(\mathtt{x}, n + 1, \beta(n, \alpha)) * [\mathrm{G}]_r \wedge \mathtt{ret} = n \rangle$
$\langle \mathsf{C}(r, \mathtt{x}, n + 1, \beta(n, \alpha)) \wedge \mathtt{ret} = n \rangle$
```

Figure 21: Total correctness of `incr`.

Now, we can proceed with the proof of termination. It is given in fig 21. The line v := [x]; allows us to establish a known value for the ordinal. We can relate this value to the value of the ordinal at the point of the CAS. If the CAS fails, we know that a further increment operation must have occurred in the environment since the read. This means we have proven that the ordinal decreases, even if we do not succeed in the update on this iteration. The reason we prove the specification for all functions β with $\alpha > \beta(n, \alpha)$ is so that the client calling `incr` can choose how much to decrement the ordinal by. The

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

Figure 22: A non-deterministic number of increments.

client can only perform a finite number of increments, since they are forced to strictly decrease α every update.

The reason ordinal numbers are used, rather than merely the natural numbers, is because this allows to handle non-deterministic numbers of calls. For example, the environment could execute a non-deterministic but finite number of increment actions, and `incr` would still guarantee to terminate. The Total TaDA paper gives the example of a loop which executes a random number of iterations which is reproduced in fig. 22. Since the value of the the abstract predicate's ordinal must be chosen at the point of the object's creation, there is no natural number we can pick which would allow us to fulfil our proof obligations that we always have enough "fuel" left to execute the increment inside the loop.

The correct solution is to choose the initial value of the counter's ordinal to be ω, the "first uncountable ordinal". The proof rule of fig. 19 requires us to prove the body $\forall\gamma$ (where γ will be an ordinal related to the variable `m`, our variant). Our specification for `incr` allows us to choose any β we want to decrement the ordinal, i.e. the client has complete control over how much the ordinal should be decremented. In the case that the counter ordinal is equal to ω, it can be decremented to $\mathtt{m} - 1$ as a result of the call. Otherwise, it can be decremented by 1, since every ordinal number less than ω is a natural number.

Total TaDA is a surprisingly neat and compact extension to TaDA which can prove total correctness. However it must restrict itself to a certain subclass of concurrent programs, namely those which are "lock-free". It is arguable that the programs for which a termination result would be most valuable lie outside this class.

2.11 Total correctness: blocking algorithms

Herlihy characterises concurrent programs according to their termination conditions [16]. Each condition forms an axis along which the space of concurrent programs may be subdivided.

A concurrent program or algorithm is one that executes across multiple threads simultaneously. Herlihy gives formal classification of the *progress* of these threads in terms of histories (as mentioned previously). Since the details of his formalisation are orthogonal to those of TaDA, his classifications will be explained intuitively.

2.11.1 Maximal vs minimal progress

If at least one thread of a concurrent program is guaranteed to be "working" at any time, the program is characterised as making "minimal progress". If no thread can be indefinitely prevented from working, this is "maximal progress". An (extreme) example of a program characterised by maximal progress would be one where no threads access shared memory, and only perform local calculations which terminate. An example of minimal progress is the concurrent `incr` implementation used as an example in previous sections. At least one concurrent call to `incr` is guaranteed to succeed, but this may cause other threads to experience *delay* as they may have to execute an extra loop iteration in their own calls.

In the setting of Total TaDA, where we have a local proof and a stability condition abstracting all other threads as a nebulous "environment", we characterise operations as causing delay if they decrease the ordinal of their region, since this is our way of enforcing that such delay must be finite. `incr` causes delay, but `read` does not. In the Herlihy setting, this means that a concurrent program with multiple increments in parallel only guarantees minimal progress.

2.11.2 Blocking vs non-blocking

A blocking algorithm is one in which some thread may be unable to execute until another thread makes an "unblocking action". In particular, it may be the case that the thread would not terminate if it were executed in isolation. If no thread can experience this, the algorithm is "non-blocking".

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

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

Figure 23: A spin lock implementation.

```
function lock(x) {
  t := incr(x.next);
  v := 0;
  while (v < t) {
  v := read(x.owner);
  }
}

function unlock(x) {
  incr(x.owner);
}
```

Figure 24: A ticket lock implementation.

Total TaDA has no mechanism for modelling blocking actions/threads/algorithms. Any operation that can be proven to terminate in Total TaDA is necessarily non-blocking.

2.11.3 A periodic table of programs

Herlihey gives classifications to algorithms based on which combination of the previous attributes they exhibit (his original paper defines a class in between blocking and non-blocking, but this is uninteresting to us since it depends on properties of the OS scheduler, which is not modelled in TaDA).

	non-blocking	**blocking**
maximal	wait-free	starvation-free
minimal	lock-free	deadlock-free

Program classes which guarantee minimal progress are a superset of their maximal progress equivalents. From this we can see that Total TaDA handles the *lock-free* class of concurrent program.

A deadlock-free module operation only guarantees termination if the environment guarantees to only perform a finite number of operations during its local execution. Otherwise, the environment could continue to do work infinitely, satisfying the definition of minimal progress without the local operation ever

```
                  [v] := 0;
b := 0;                 ||
while (b = 0) {         ||  lock(x);
  lock(x);              ||  unlock(x);
  unlock(x);            ||  [v] := 1;
  b := [v];             ||
}                       ||
```

Figure 25: Terminates if lock is starvation free.

finishing. Furthermore, since it is blocking, the environment must terminate in an "unblocked state" which allows the operation to finish if it has not already.

Spinlock is an example of a deadlock-free module. A possible implementation is given in fig. 23. We can see that if the local thread is running `lock`, it will only terminate if the environment only performs a finite number of `lock` operations itself, since otherwise the local thread's CAS could fail forever. Furthermore, the environment must eventually leave the lock in an unlocked state.

Ticket lock is an example of a starvation-free module (assuming the counters it uses are wait-free). An implementation is given in fig. 24. Even if the environment is running an infinite number of lock-unlocks, eventually `owner` will increment to the value of `next` that was read at the start of the local thread's lock operation. The environment still needs to guarantee it will always release a held lock, however.

Fig. 25 is an example of a client program that guarantees to terminate using ticket lock, but not with spinlock.

The aim of this project is to extend Total TaDA to handle programs from the deadlock-free class. We will first examine arguably the two most prominent logics that reason about the termination of deadlock-free programs, and discuss both the lessons that can be learned from them, and the significant compromises made by the existing work.

2.12 LiLi

Lili [17] is a rely guarantee style system which incorporates the separating conjunction of separation logic, and certain aspects of linearisability from

Herlihy's work to prove termination of module operations. There are a few key concepts in LiLi that are relevant to our proposed extension.

2.12.1 Definite actions

LiLi judgements enforce that a thread will carry out a set of "definite actions" as part of its execution. In turn, it relies on the environment carrying out a set of definite actions in order to guarantee its own termination. For example, a thread may be required to fulfill the definite action $(L \mapsto 1) \leadsto (L \mapsto 0)$ which encodes that if it ever locks the lock L, it must release it. Unfortunately, LiLi's reasoning requires that the exact thread ID currently holding the lock is encoded somewhere inside the resource's state, so in a real proof the definite action might look something more like $(L \mapsto t) \leadsto (L \mapsto 0)$ where t is a thread id. This means that LiLi can only abstract the environment in a limited way since the number of threads must be fixed in advance of the proof.

2.12.2 Unblocking conditions

A loop inside a blocking program will not have a variant. For instance, this is the loop of a spin lock implementation (with the lock at location x).

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

If the lock is locked, the CAS will *never* succeed, and therefore the loop will never terminate. However, if the lock is unlocked, the loop is *guaranteed* to terminate. We can say that $L \mapsto 0$ is the "unblocking condition" of the loop. The loop is guaranteed to terminate if this condition is established and preserved. Since the local thread knows the actions any other threads are guaranteed to make (via the definite actions) it is possible to reason that the unblocking condition is inevitably established.

2.12.3 Limitations

As mentioned, LiLi's assertions about the actions of the environment are explicitly parameterised by thread IDs. This is anathema to TaDA's more abstracted view of the environment. It seems intuitive that we must have some sort of formalisation of what the environment *must* do to guarantee local termination, as opposed to just what it *may* do. However, LiLi solves this problem by reasoning about a ordered, finite queue of actions by each thread.

There are more fundamental mismatches between the goals of LiLi and Total TaDA. LiLi aims to solve a far simpler problem - namely that the operations of a module terminate and appear linearisable. In this way, LiLi's correctness condition is much like Herlihy's original linearisability condition. LiLi cannot provide a `lock` specification that a client can use to reason about its own termination, since it provides no mechanism for a client to verify that it satisfies the `lock` operation's required definite actions. In fact, the language of LiLi does not include function calls at all! Even within a module operation proof, references to `lock` and `unlock` calls are merely typographical shorthand for an inlined loop or similar series of statements which functions as a lock.

Furthermore, LiLi's style of reasoning scales poorly if the operations of a module do not all perform the same blocking actions. Every substantial proof in the paper is of a module embodying some variation of a counter increment or list push/pop, which is implemented as a critical section surrounded by some flavour of lock-unlock. This is a very local style of blocking, since no module operation can terminate without unblocking every other operation. We will see in general that this is not the case.

2.12.4 Lessons

The way the language of Total TaDA blocks is similar to the way LiLi blocks - a while loop condition which relies on another thread/the environment performing an update. It seems intuitive that we will have to determine the loop's unblocking condition, and guarantee that the environment performs actions that eventually satisfy and preserve it.

However, LiLi's formalism offers us no help in doing this, since the sacrifices they have made to ensure their rules are sound (no clients, no thread forking, no function calls, poor non-local blocking) constrict their expressivity to the point that porting their restrictions into the world of TaDA would leave us with almost no logic at all, due to the way TaDA is built to facilitate modular proofs.

2.13 Boström and Müller

This work [18] offers a far more complete logic with fewer odd restrictions. However, their memory model is based on message passing, and does not include a shared heap. This means that all inter-thread blocking is mediated by language primitives. The only way for any thread to wait on the update of another is through the `lock`, (channel) `receive` and (thread) `join` operations, which are given as part of the semantics of the language and not as module specifications. In this way, it is impossible for a thread to experience arbitrary blocking that is not mediated through one of these constructs. Because of this, it is possible to build a bespoke "token language" for each construct. For example, a `lock` operation produces a "releases" obligation token that must be consumed by a corresponding unlock, while a `receive` operation on a channel must either consume a "sends" credit token produced by an earlier `send` to the channel, or produce a sends obligation token, representing that it is blocking on a `send` that may not have occurred yet.

2.13.1 Lessons

Separation logic in general is based on a model with a shared heap, so any module can potentially block in a way that is not possible with a message passing setup. The equivalent to their approach in Total TaDA would be that specifying a module for use by a client also means specifying a token language to go with it, which describes exactly how it blocks (in some way this would be analogous to the existing guard tokens, which abstract both the thread's and the environment's permission to update the region). However, the fundamental nature of Total TaDA's assertions means that this technique would almost certainly be unsound, since $P * Q \Rightarrow P$ is always sound; i.e. we could forget tokens whenever we want, even if they denote an obligation to unblock.

3 B-TT: extending Total TaDA

We want to show that every possible execution of a blocking program terminates. We can view each execution as a trace of updates to shared state, but in particular we only care about updates to module instances which might be involved in blocking. Take the following code as an example:

$$\begin{array}{c} \alpha_1\ \texttt{lock(x);} \\ \begin{array}{l||l} \alpha_2\ \texttt{lock(x);} & \texttt{unlock(x);}\ \beta \\ \alpha_3\ \texttt{unlock(x);} & \end{array} \end{array}$$

Every real execution of this code can be characterised by some sequence of the timestamps marked in red. Indeed, because of the way `lock` blocks, there is only one possible sequence: $\alpha_1 \to \beta \to \alpha_2 \to \alpha_3$.

We can view any program as inducing a set of all possible traces of its blocking actions. We give proof rules that allow us to soundly determine a superset of these possible traces. If we can show that this superset terminates, we know that the real program will terminate. Ideally, we want this superset to be as close to the real set of possible traces as possible.

We define our set of possible traces using two restrictions. The first is the sequential ordering of the code. That is, two operations appearing one after the other in the code must necessarily happen one after the other in any trace. The second is the consistency of state transitions. For example, a `lock` operation will only take effect if it makes a transition from state 0 to state 1, so no trace will have two `lock`s on the same lock immediately in sequence. Linearisability allows us to determine the existence of the atomic timepoints at which operations appear to take effect.

We give rules for deriving and resolving these restrictions. What follows is the formalisation of what has been intuitively described above.

We give an extension to the Total TaDA system, named B-TT. The while (now "while1"), parallel, sequence, make atomic, and if rules have been modified. New rules "while2", "while3", and "prefix" have been added. All other proof rules remain the same, except for the addition of the "action ordering context", which is written π on the left hand side of the turnstile.

The system will be concerned with the deriving of actions of the form (a, X, Q, α), where a uniquely identifies the region being updated, X denotes the possible states consistent with the update, α is an "abstract timestamp" uniquely identifying when the action takes place, and Q is a "transition function". Q is already used in the make atomic rule of TaDA to denote what behaviour is *possible* when performing an atomic update. Now we extract additional information from it; transitions which are *not* possible. Combining the information from X and Q allows us to describe an operation's unblocking condition.

Initially, a special case of the rules will be presented, followed by a discussion of various possible extensions.

Total TaDA assumes the existence of a set of “region type names” (the $\mathbf{t}$ of $\mathbf{t}_a^\lambda(x)$) named ARType. B-TT further assumes the existence of a set $\mathsf{ABRType}$ the “blocking region type names” which is a subset of ARType. The predicate $\text{blocking}(a)$ takes a region ID and returns true if the region referred to by the region ID has a blocking type. $\text{blocking}(a)$ is true for every a which is a region ID in $\mathsf{AProgram}$. This is enforced by the proof rules. “Blocking actions” are atomic updates which take place over a blocking region.

We define the action ordering context as follows:

$$\mathsf{AProgram} \triangleq \mathcal{P}(\mathsf{Action}_{\mathcal{O}}) \times \mathsf{WFPOrder}_{\mathcal{O}}$$

That is, the action context is a tuple containing both a set of blocking actions $\mathsf{Action}_{\mathcal{O}}$, and a well-founded ordering $\mathsf{WFPOrder}_{\mathcal{O}}$ over their “abstract timestamps”. A blocking action is defined as:

$$\mathsf{Action}_{\mathcal{O}} \triangleq \coprod_{a \in \mathsf{RId}} \mathcal{P}(\mathsf{RState}_a) \times (\mathsf{RState}_a \to \mathcal{P}(\mathsf{RState}_a)) \times \mathcal{O}$$

This denotes a 4-tuple where the types of the latter three values are parameterised by the value of the first. The first element is the RId, the region ID over which the action is being performed (as a of $\mathbf{t}_a^\lambda(x)$). The second element is a set of the states of region a, denoting the set of possible states from which the action can take place. The third element is the state transformation function (as Q of $a : x \in X \rightsquigarrow Q(x)$ in the atomicity context). The fourth element is the abstract timestamp, which is guaranteed by the proof rules to be unique across all the actions in the action ordering context.

The ordering over the abstract timestamps is simply defined as

$$\mathsf{WFPOrder}_{\mathcal{O}} \triangleq \left\{ < \mid < \subset \mathcal{O}^2, < \text{well-founded} \right\}$$

We will use $\emptyset$ as syntactic sugar for $(\emptyset, \emptyset) \in \mathsf{AProgram}$

The triple

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

Can be intuitively understood as meaning that $\mathbb{C}$ will terminate if and only if it carries out all actions in π in an order consistent with the timestamp ordering. Furthermore, the environment is assumed to eventually unblock all actions in $\mathcal{A}$ before they occur. We will define this more precisely later.

We define sequence and union over $\mathsf{APrograms}$ π_1, π_2 to represent the effects of executing the programs represented by them sequentially, or in parallel, respectively.

union:

$$(A_1, <_1) \cup (A_2, <_2) = (A_1 \cup A_2, <_1 \cup <_2)$$

sequence:

$$(A_1, <_1); (A_2, <_2) = (A_1 {\cup} A_2, <_1 \cup <_2 \cup \{(a_1, a_2) | (\_, \_, \_, a_2) \in A_2, (\_, \_, \_, a_1) \in A_1\})$$

That is, if π_2 is executed after π_1, the ordering relation will enforce that the abstract timestamps of π_1 are all less than those of π_2.

Each modified or additional proof rule will be explained in turn to informally motivate their correctness. Where it is non-trivial, a more rigorous proof will be given later.

3.1 Rules

$$\frac{\pi_1; \lambda; \mathcal{A} \vdash \left\{P_1\right\} \mathbb{C}_1 \left\{Q_1\right\} \qquad \pi_2; \lambda; \mathcal{A} \vdash \left\{P_2\right\} \mathbb{C}_2 \left\{Q_2\right\} \quad \mathit{distinct}_{\mathcal{O}}(\pi_1, \pi_2)}{\pi_1 \cup \pi_2; \lambda; \mathcal{A} \vdash \left\{P_1 * P_2\right\} \mathbb{C}_1 \parallel \mathbb{C}_2 \left\{Q_1 * Q_2\right\}} \; \mathit{Parallel}$$

$$\frac{\pi_1; \lambda; \mathcal{A} \vdash \left\{P\right\} \mathbb{C}_1 \left\{R\right\} \qquad \pi_2; \lambda; \mathcal{A} \vdash \left\{R\right\} \mathbb{C}_2 \left\{Q\right\} \quad \mathit{distinct}_{\mathcal{O}}(\pi_1, \pi_2)}{(\pi_1; \pi_2); \lambda; \mathcal{A} \vdash \left\{P\right\} \mathbb{C}_1; \mathbb{C}_2 \left\{Q\right\}} \; \mathit{Sequence}$$

These are simply the old parallel and sequence rule, additionally enforcing the relations on blocking actions. The side condition enforces that the timestamps of both $\mathbb{C}$s are entirely distinct.

$$\frac{\pi; \lambda; \mathcal{A} \vdash \left\{P \land \mathbb{B}\right\} \mathbb{C}_1 \left\{Q\right\} \qquad \pi; \lambda; \mathcal{A} \vdash \left\{P \land \neg\mathbb{B}\right\} \mathbb{C}_2 \left\{Q\right\}}{\pi; \lambda; \mathcal{A} \vdash \left\{P\right\} \texttt{if } (\mathbb{B})\ \mathbb{C}_1 \texttt{ else } \mathbb{C}_2 \left\{Q\right\}} \; \mathit{If}$$

This is the almost the same as the old if rule, except it enforces that both branches must carry out the same blocking actions. This is a slightly unsatisfactory restriction, but a potential improvement will be discussed in a later section.

3.1.1 While rules

The while rules of the system are crucial, as they are how we describe blocking. We will give a more restricted case of the rules here and describe how they can be broadened later.

$$\frac{\forall \gamma \leq \alpha.\, \emptyset; \lambda; \mathcal{A} \vdash \left\{ P(\gamma) \right\} \mathbb{C} \left\{ \exists \beta.\, P(\beta) \land \beta < \alpha \right\}}{\emptyset; \lambda; \mathcal{A} \vdash \left\{ P(\alpha) \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \exists \beta.\, P(\beta) \land \alpha \geq \beta \land \neg \mathbb{B} \right\}} \; While1$$

This is the former while rule of Total TaDA. For now, we enforce that no blocking actions take place within its body. We will lift this restriction later.

$$\frac{\begin{array}{c} T = X \setminus Q^{-1}(\emptyset) \qquad \mathcal{A} = a : x \in X \rightsquigarrow Q(x), \mathcal{A}' \qquad \mathsf{blocking}(a) \\ \forall \gamma \leq \alpha.\, \emptyset; \lambda'; \mathcal{A} \vdash \left\{ \begin{array}{c} \exists x \in X.\, P(x, \gamma) * \\ \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge \land \mathbb{B} \end{array} \right\} \mathbb{C} \left\{ \begin{array}{c} \exists x \in X, \beta.\, P(x, \beta) * \\ \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge \land \gamma \geq \beta \end{array} \right\} \\ \forall \gamma \leq \alpha.\, \emptyset; \lambda'; \mathcal{A} \vdash \left\{ \begin{array}{c} \exists x \in T.\, P(t, \gamma) * \\ \mathbf{t}_a^{\lambda}(t) * a \Mapsto \blacklozenge \land \mathbb{B} \end{array} \right\} \mathbb{C} \left\{ \begin{array}{c} \exists x \in T, \beta.\, P(t, \beta) * \\ \mathbf{t}_a^{\lambda}(t) * a \Mapsto \blacklozenge \land \gamma > \beta \end{array} \right\} \end{array}}{\emptyset; \lambda'; \mathcal{A} \vdash \left\{ \begin{array}{c} \exists x \in X.\, P(x, \alpha) * \\ \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge \end{array} \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{c} \exists x \in X, \beta.\, P(x, \beta) * \\ \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge \land \alpha \geq \beta \land \neg \mathbb{B} \end{array} \right\}} \; While2$$

This rule is the first which reasons that a loop can terminate in the presence of blocking. An existing function of the atomicity tracking component $a \Mapsto \blacklozenge$ is to restrict the interference the environment is allowed to cause to the set X. We now interpret the atomicity tracking component as not only constraining the environment's interference to X, but also guaranteeing the environment will unblock the Q part of the transition $a : x \in X \rightsquigarrow Q(x)$, since the transition cannot occur when $Q(x) = \emptyset$. The set of states in which Q is unblocked is given by T in the first antecedent.

We know that the environment will eventually constrain the state of the region to T (in a finite amount of time), and we also know that because the atomicity tracking component remains unfulfilled, the local thread does not affect the state of the region. If the loop has not terminated by the point the environment does this, we will begin executing the case in the third antecedent, where we have proven that an ordinal constantly decreases. Thus the proof of termination is analogous to the lock-free case, except we must wait to be unblocked first.

$$\dfrac{\begin{array}{c} \mathcal{A} = a : x \in X \rightsquigarrow Q(x), \mathcal{A}' \qquad \mathsf{blocking}(a) \\ T = X \setminus Q^{-1}(\emptyset) \qquad \forall a \in \mathrm{dom}(\mathcal{A}').\, \neg\mathsf{blocking}(a) \qquad T \subseteq X' \subseteq X \\ \emptyset; \lambda; \mathcal{A} \vdash \left\{ \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \wedge \mathbb{B} \right\} \mathbb{C} \left\{ \begin{array}{l} (\exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \wedge \mathbb{B}) \vee \\ \left(\begin{array}{l} \exists x \in X', \exists y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg\mathbb{B} \end{array} \right) \end{array} \right\} \\ \emptyset; \lambda; \mathcal{A} \vdash \left\{ \exists x \in T.\, P(x) * \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \wedge \mathbb{B} \right\} \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg\mathbb{B} \end{array} \right\} \end{array}}{\emptyset; \lambda; \mathcal{A} \vdash \left\{ \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \wedge \mathbb{B} \right\} \texttt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg\mathbb{B} \end{array} \right\}} \; \mathit{While3}$$

This rule allows a blocking action to talk place inside a loop that is blocking on the same region. It also enforces that the loop will terminate after the linearisation point has taken place. In the case that the environment has not constrained the interference, we may or may not terminate. However, once the environment has constrained the interference, we guarantee that the linearisation point will occur.

This rule could be formulated in terms of decreasing ordinals, but in practice we found it severely complicated client proofs for little expressive benefit.

$$\dfrac{\pi; \lambda; \mathcal{A} \vdash \left\{ P \right\} \mathbb{C} \left\{ \exists x \in X.\, Q(x) * \mathbf{t}_a^\lambda(x) \right\}}{((\pi; \{(a, \{\between\}, \lambda x.\, X, \alpha)\}, \emptyset)); \lambda; \mathcal{A} \vdash \left\{ P \right\} \mathbb{C} \left\{ \exists x \in X.\, Q(x) * \mathbf{t}_a^\lambda(x) \right\}} \; \mathit{Prefix}$$

This rule allows us to generate an action for a blocking region which is newly created, allowing us to keep track of its initial state in the list of blocking actions. Since this action is modeled as taking place after all existing ones, the rule cannot be used to “fake” the creation of a module at an inappropriate time. $\between$ is the “start unit”, which all ordinary Qs implicitly map to $\emptyset$.

$$\frac{\begin{array}{c} a \notin \mathcal{A} \quad X' \subseteq X \quad \{(x,y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \underline{\mathbf{if}}\ \mathsf{blocking}(t)\ \underline{\mathbf{then}}\ \pi = \left(\{(a, X', Q, \alpha)\}, \emptyset\right) \cup \pi'\ \underline{\mathbf{else}}\ \pi = \pi' \\ \pi'; \lambda'; a : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash \begin{array}{c} \left\{p_p * \exists x \in X.\, \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge\right\} \\ \mathbb{C} \\ \left\{\exists x \in X', y \in Q(x).\, q_p(x,y) * a \Mapsto (x,y)\right\} \end{array} \end{array}}{\pi; \lambda'; \mathcal{A} \vdash \forall x \in X. \left\langle p_p \middle| \mathbf{t}_a^{\lambda}(x) * [\mathrm{G}]_a \right\rangle\ \mathbb{C}\ \ \exists y \in Q(x). \left\langle q_p(x,y) \middle| \mathbf{t}_a^{\lambda}(y) * [\mathrm{G}]_a \right\rangle}\ \mathit{Makeatomic}$$

Now we come to the revised make atomic rule. This is nearly the same as before, except that if the atomic update is to a blocking region, we update the action ordering context to keep track of the new action. Note that we do not know for sure when this action takes place in relation to others in the code, so it is not ordered with respect to them.

3.2 Termination

As previously mentioned, establishing a triple of the form

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

guarantees (roughly) that $\mathbb{C}$ will terminate if and only if it performs all the blocking actions in π. This is not yet a guarantee of termination! We must first enumerate another guarantee made by the triple, and show how this is used to establish termination.

3.2.1 Minimal actions

For a given π, a blocking action $act \in \pi$ is "minimal" if $\neg\exists act'. \in \pi$ s.t. $p_4(act') <_\pi p_4(act)$ where p_4 is the fourth projection. Intuitively, there is no action in π known to definitely occur before it.

An action $(a, X, Q, \alpha) \in \pi$ is considered unblocked if the state of the region a, given by x, has $Q(x) \neq \emptyset$.

Let $min(\pi)$ denote the set of minimal actions for π.

The semantics of the triple above guarantee that if $\mathbb{C}$ is run for a sufficient amount of time, maintaining the same set of minimal unblocked actions, one of them will eventually occur.

Of course, since the environment can alter the state of the regions, we cannot guarantee that the minimal unblocked actions will not change as execution progresses. To guarantee termination in isolation, we must ask the question "if the environment never changes the states of the blocking regions, is $\mathbb{C}$ guaranteed to terminate?". This is the function of the *terminates* rule.

3.3 The terminates rule

$$\frac{\pi; \lambda; \mathcal{A} \vdash \forall\!\!\!\forall x \in X. \left\langle p_p \,\middle|\, p(x) \right\rangle \ \mathbb{C} \ \ \exists\!\!\!\exists y \in Y(x). \left\langle q_p(x,y) \,\middle|\, q(x,y) \right\rangle \qquad \mathit{nec\text{-}consist}(\pi)}{\pi; \lambda; \mathcal{A} \vdash_{term} \forall\!\!\!\forall x \in X. \left\langle p_p \,\middle|\, p(x) \right\rangle \ \mathbb{C} \ \ \exists\!\!\!\exists y \in Y(x). \left\langle q_p(x,y) \,\middle|\, q(x,y) \right\rangle} \ \mathit{terminates}$$

The semantics of the $\vdash_{term}$ triple really do guarantee that $\mathbb{C}$ terminates, assuming π provides a complete account of every transition the blocking regions will undergo. Note that we can freely turn a $\vdash_{term}$ triple back into an ordinary Total TaDA triple if we wish to continue the proof or use it as part of a larger one.

3.3.1 The necessary consistency side-condition

This side condition on π enforces that, given the ordering of actions enforced by $<_\pi$ and assuming that the blocking region states are never changed by the environment, no matter which minimal actions occur in which order, every execution described by π will eventually carry out all actions.

Intuitively, the action ordering context induces a set of traces of linearisation points such that every execution of the program is guaranteed to fulfil one of them. These traces are analogous to a linearisable history in the Herlihy sense. The condition encodes that every possible trace executes all actions, and therefore the program must terminate. Formal definition of a special case of the condition that is sufficient for the next few examples will be given here. We will see later how it can be generalised.

Take the set of abstract timestamps of π, $\mathcal{O}_\pi$. Consider a finite sequence s_π of abstract timestamps such that every timestamp is in $\mathcal{O}_\pi$, and no timestamp appears more than once. We refer to the action associated with a timestamp α as act_α.

Define the region trace $s_\pi|r$ as the subsequence of s_π containing exactly the α s.t. $p_1(act_\alpha) = r$.

We define the "consistency" of a region trace. For now, we will define a special case of consistency. We assume that for all $act \in p_1(\pi)$, taking $Q = p_3(act)$, for all $x \in p_2(act)$, $Q(x)$ maps to the same (non-empty) set (for convenience, we will refer to this set of states, as Y_Q). We will motivate later that such a restriction defines the class of regions that can be simulated by a "general region" defined within the programming language of TaDA, and show how it can be relaxed.

A region trace is consistent if $(s_\pi|r)_0 =\emptyset$ and, for all α_1, α_2,
$succ_{s_\pi}(\alpha_1) = \alpha_2 \Rightarrow Y_{p_3(act_{\alpha_1})} \subseteq p_2(act_{\alpha_2})$

That is, the state transitions of the actions in sequence match up with each other.

We say s_π is an "abstract trace" of π if it is a maximal sequence such that the following conditions are met:

- $\alpha_1 <_{s_\pi} \alpha_2 \Rightarrow \neg(\alpha_2 <_\pi \alpha_1)$
- $succ_{s_\pi}(\alpha_1) = \alpha_2 \Rightarrow (\forall \alpha_3 \neq \alpha_1 . \, (\alpha_3 <_\pi \alpha_2) \Rightarrow (\alpha_3 <_{s_\pi} \alpha_1))$
- for all a, $s_\pi|a$ is consistent.

nec-consist(π) holds iff every abstract trace s_π of π contains all timestamps in $\mathcal{O}_\pi$.

We give an equivalent formulation, and encode it as a logic program, in appendix A.

3.3.2 Relation to histories and linearisability

It has long been understood that Herlihy's module linearisability implies the existence of a linearisation point for each module opearation. TaDA offers a more nuanced notion of "conditional" linearisability; a specification may, through the pseudoquantifier, enforce additional restrictions on the environment that are needed for the operation to appear to have a linearisation point.

A linearisable history (in the Herlihy sense) can always be equivalently viewed as a series of linearisation points executing one after another. Similarly, any execution of a program for which a TaDA triple can be proven can be viewed as executing a series of linearisation points, since we have proven that every use of each module operation necessarily respects the conditions for them to appear linearisable.

Herlihy's correctness condition views a module implementation as the set of all histories describing the possible interleavings of its operations. Calculating the abstract traces of π for *nec-consist*(π) can be thought of as constructing a Herlihy style implementation description of the code.

Herlihy also requires his histories to be "legal", that is, for each object mentioned in the history, the subhistory of just the operations on that object lies within its implementation. This is equivalent to our consistency check on all region traces $s_\pi|a$.

3.4 Example: spin lock

3.4.1 Module specification

We can, for the first time, prove a specification for spin lock such that we can prove the termination of several interesting clients.

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

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

The proof of `lock` is given by fig. 26. The proof of `unlock` is given by fig. 27. The interpretations remain identical to the TaDA proof seen earlier, except now the **Lock** region has a blocking type. The following specifications are proven:

$$(a, \{0\}, B, \alpha), \emptyset \vdash \forall l \in \mathbb{B}. \left\langle \mathsf{L}(a, \mathtt{x}, l) \right\rangle \ \mathtt{lock(x)} \ \left\langle \mathsf{L}(a, \mathtt{x}, \top) \wedge \neg l \right\rangle$$

$$(a, \{1\}, U, \alpha'), \emptyset \vdash \left\langle \mathsf{L}(a, \mathtt{x}, \top) \right\rangle \ \mathtt{unlock} \ \left\langle \mathsf{L}(a, \mathtt{x}, \bot) \right\rangle$$

with $B(0) = \{1\}$, $B(1) = \{\}$, $U(0) = \{0\}$, $U(1) = \{0\}$.

```
$(a, \{0\}, Q, \alpha) \vdash$
with $Q(0) = \{1\}$, $Q(1) = \{\}$
$\mathbb{\forall} l \in \mathbb{B}.$
$\langle \mathsf{L}(a, \mathtt{x}, l) \rangle$
  abstract; quantify $a$; $y :=$ if $l$ then 1 else 0
  $\mathbb{\forall} y \in \{0, 1\}\,.$
  $\langle \mathbf{Lock}_a(\mathtt{x}, y) * [\mathrm{G}]_a \rangle$
    make atomic
    $a : y \in \{0, 1\} \rightsquigarrow Q(y) \vdash$
      $\{\exists y \in \{0, 1\}\,.\,\mathbf{Lock}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
      b := 0;
      $\left\{\begin{array}{l}\exists y \in \{0, 1\}\,.\,\mathbf{Lock}_a(\mathtt{x}, y) * \\ (a \Mapsto (0, 1) \wedge \mathtt{b} = 1 \vee a \Mapsto \blacklozenge \wedge \mathtt{b} = 0)\end{array}\right\}$
      while (b = 0) {
        while3
        with $T = \{0\}$
          $\{\exists y \in \{0, 1\} \,/\, \{0\}\,.\,\mathbf{Lock}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
            update region
            $\mathbb{\forall} n \in \{0, 1\} \,/\, \{0\}\,.$
            $\langle \mathtt{x} \mapsto n \rangle$
            b := CAS(x, 0, 1);
            $\left\langle\begin{array}{l}(\mathtt{x} \mapsto 1 \wedge n = 0 \wedge \mathtt{b} = 1) \vee \\ (\mathtt{x} \mapsto n \wedge n \neq 0 \wedge \mathtt{b} = 0)\end{array}\right\rangle \,/\, \langle (\mathtt{x} \mapsto 1 \wedge n = 0 \wedge \mathtt{b} = 1) \rangle$
          $\left\{\begin{array}{l}\exists y \in \{0, 1\}\,.\,\mathbf{Lock}_a(\mathtt{x}, y) * \\ (a \Mapsto (0, 1) \wedge \mathtt{b} = 1 \vee a \Mapsto \blacklozenge \wedge \mathtt{b} = 0)\end{array}\right\} \,/\, \{(a \Mapsto (0, 1) \wedge \mathtt{b} = 1\}$
      }
      $\{\exists y \in \{0\}, z \in Q(y).\, a \Mapsto (y, z) \wedge \mathtt{b} = 1\}$
  $\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \wedge y = 0 \rangle$
$\langle \mathsf{L}(a, \mathtt{x}, \top) \wedge \neg l \rangle$
```

Figure 26: Proof of `lock(x)`. The red assertions represent the proof of the loop body under the subsetted interference.

3.4.2 A sequential client

Consider the simple client given by

```
lock(x);
unlock(x);
```

We can easily prove that this client terminates. We will use the `IsLock`, `Locked`, and **CAPLock** abstractions of the TaDA paper: that is, we define abstract

$(a, \{1\}, Q', \alpha') \vdash$
with $Q'(0) = \{0\}$, $Q'(1) = \{0\}$
$\langle \mathsf{L}(a, \mathtt{x}, \top) \rangle$

abstract; quantify a

$\langle \mathbf{Lock}_a(\mathtt{x}, 1) * [\mathrm{G}]_a \rangle$

make atomic

$a : 1 \rightsquigarrow 0 \vdash$
$\{\mathbf{Lock}_a(\mathtt{x}, 1) * a \Mapsto \blacklozenge\}$

update region

$\langle \mathtt{x} \mapsto 1 \rangle$
$[\mathtt{x}] := 0;$
$\langle \mathtt{x} \mapsto 0 \rangle$

$\{a \Mapsto (1, 0)\}$
$\{\exists y \in \{1\}, z \in Q'(y).\, a \Mapsto (y, z)\}$
$\langle \mathbf{Lock}_a(\mathtt{x}, 0) * [\mathrm{G}]_a \rangle$
$\langle \mathsf{L}(a, \mathtt{x}, \bot) \rangle$

Figure 27: Proof of $\mathtt{unlock}(x)$.

predicates `IsLock`, `Locked` and a region **CAPLock** such that

$$\begin{aligned} \mathtt{isLock}(a, x) &\triangleq \exists s \in \{\bot, \top\}\,.\, \mathbf{CAPLock}_a(x, s) \\ \mathtt{Locked}(a, x) &\triangleq \mathbf{CAPLock}_a(x, \top) * [\mathrm{K}]_a \\ I(\mathbf{CAPLock}_a(x, s)) &\triangleq \mathtt{L}(x, s) \end{aligned}$$

with the transition system of **CAPLock** defined as

$$\mathbf{0} \;:\; 0 \rightsquigarrow 1 \qquad \mathrm{G} \;:\; 1 \rightsquigarrow 0 \qquad \mathrm{G} \bullet \mathrm{G} = \bot$$

where $\mathbf{0}$ is the "empty guard", held by all threads.

A proof of the triple
$\pi \vdash \left\{ \mathtt{isLock}(a, \mathtt{x}) \right\}\ \mathtt{lock(x); unlock(x)}\ \left\{ \mathtt{isLock}(a, \mathtt{x}) \right\}$
where $\pi = \{(a, \{\}, \lambda x.\, \{0\}, \alpha), (a, \{0\}, B, \alpha'), (a, \{1\}, U, \alpha'')\}, (\alpha < \alpha' < \alpha'')$ is given in fig. 28

To show $\pi \vdash_{term} \left\{ \mathtt{isLock}(a, \mathtt{x}) \right\}\ \mathtt{lock(x); unlock(x)}\ \left\{ \mathtt{isLock}(a, \mathtt{x}) \right\}$, we can observe that only one sequence satisfies the definition of an abstract trace; $\alpha \to \alpha' \to \alpha''$ and that this sequence contains all timestamps.

```
$\pi \vdash$
$\{\mathsf{IsLock}(a, \mathtt{x})\}$
abstract
    $(a, \{\}, \lambda x.\, \{0\}, \alpha), \emptyset \vdash$
    $(a, \{0\}, B, \alpha'), \emptyset \vdash$
    $\{\exists s \in \{\bot, \top\}\,.\, \mathbf{CAPLock}_a(\mathtt{x}, s)\}$
    weaken
        $\forall\!\!\forall s \in \{\bot, \top\}$
        $\langle \mathbf{CAPLock}_a(\mathtt{x}, s) \rangle$
        use atomic
            $\langle \mathsf{L}(\mathtt{x}, s) \rangle$
            lock(x);
            $\langle \mathsf{L}(\mathtt{x}, \top) \rangle$
        $\langle \mathbf{CAPLock}_a(\mathtt{x}, \top) * [\mathrm{K}]_a \rangle$
    $(a, \{1\}, U, \alpha''), \emptyset \vdash$
    $\{\mathbf{CAPLock}_a(\mathtt{x}, \top) * [\mathrm{K}]_a\}$
    weaken
        $\langle \mathbf{CAPLock}_a(\mathtt{x}, \top) * [\mathrm{K}]_a \rangle$
        use atomic
            $\langle \mathsf{L}(\mathtt{x}, \top) \rangle$
            unlock(x);
            $\langle \mathsf{L}(\mathtt{x}, \bot) \rangle$
        $\langle \mathbf{CAPLock}_a(\mathtt{x}, \bot) \rangle$
    $\{\exists s \in \{\bot, \top\}\,.\, \mathbf{CCounter}_a(\mathtt{x}, s)\}$
$\{\mathsf{IsLock}(a, \mathtt{x})\}$
```

Figure 28: Proof of simple sequential client. Note that we apply the prefix rule across an implicit "skip" statement at the head of the program.

3.4.3 A concurrent client

```
lock(x);    ||  lock(x);
unlock(x);  ||  unlock(x);
```

Note that with the thread-compositionality of our approach, we can use the specification of the previously proven sequential client to immediately deduce the specification of this client. The proof is given in fig. 29.

$\pi \cup \pi'$ expanded is
$\{(a, \{\}, \lambda x.\, \{0\}, \alpha), (a, \{0\}, B, \alpha'), (a, \{1\}, U, \alpha''),$
$(a, \{0\}, B, \beta), (a, \{1\}, U, \beta')\},$
$(\alpha < \alpha' < \alpha'', \beta < \beta').$

There are two possible abstract traces: $\alpha \to \alpha' \to \alpha'' \to \beta \to \beta'$ or $\alpha \to \beta \to \beta' \to \alpha' \to \alpha''$. Both contain all timestamps.

$$\begin{array}{c}
\pi \cup \pi' \vdash \\
\left\{\mathsf{IsLock}(a, \mathtt{x})\right\} \\
\begin{array}{l||l}
\pi \vdash & \pi' \vdash \\
\left\{\mathsf{IsLock}(a, \mathtt{x})\right\} & \left\{\mathsf{IsLock}(a, \mathtt{x})\right\} \\
\mathtt{lock(x);} & \mathtt{lock(x);} \\
\mathtt{unlock(x);} & \mathtt{unlock(x);} \\
\left\{\mathsf{IsLock}(a, \mathtt{x})\right\} & \left\{\mathsf{IsLock}(a, \mathtt{x})\right\}
\end{array} \\
\left\{\mathsf{IsLock}(a, \mathtt{x})\right\}
\end{array}$$

Figure 29: Concurrent client proof. π' is π with fresh timestamps and the initialisation event removed (which is always sound to do).

$$\begin{array}{c}
\pi \cup \pi' \vdash \\
\left\{\mathsf{IsLock}(\mathsf{a}, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\} \\
\begin{array}{l||l}
\pi \vdash & \pi' \vdash \\
\left\{\mathsf{IsLock}(a, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\} & \left\{\mathsf{IsLock}(a, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\} \\
\mathtt{lock(x);} & \mathtt{lock(y);} \\
\mathtt{lock(y);} & \mathtt{lock(x);} \\
\mathtt{unlock(y);} & \mathtt{unlock(x);} \\
\mathtt{unlock(x);} & \mathtt{unlock(y);} \\
\left\{\mathsf{IsLock}(a, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\} & \left\{\mathsf{IsLock}(a, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\}
\end{array} \\
\left\{\mathsf{IsLock}(a, \mathtt{x}) * \mathsf{IsLock}(b, \mathtt{y})\right\}
\end{array}$$

Figure 30: A *very* high level sketch of the triple proof for the deadlock example.

3.4.4 Failure to prove a deadlocking client

```
lock(x);   || lock(y);
lock(y);   || lock(x);
unlock(y); || unlock(x);
unlock(x); || unlock(y);
```

Above is the canonical deadlocking example. Each thread acquires the locks in a different order. If the left thread executes `lock(x)`, then the right thread executes `lock(y)`, the program will fail to terminate. If the logic is sound, we expect to fail to prove this code terminates. A proof of the non-terminating triple is given in fig. 30.

π in this case could be
$\{(a, \{\}, \lambda x. \{0\}, \alpha_1), (a, \{0\}, B, \alpha_2), (b, \{0\}, B, \alpha_3), (b, \{1\}, U, \alpha_4), (a, \{1\}, U, \alpha_5)\}$, $(\alpha_1 < \alpha_2 < \alpha_3 < \alpha_4 < \alpha_5)$ (depending on where we arbitrarily choose to invoke the prefix rule to get the initial state of the lock)

π' could be
$\{(b,\{\},\lambda x.\,\{0\},\beta_1),(b,\{0\},B,\beta_2),(a,\{0\},B,\beta_3),(a,\{1\},U,\beta_4),(b,\{1\},U,\beta_5)\}$,
$(\beta_1 < \beta_2 < \beta_3 < \beta_4 < \beta_5)$

Here, $\alpha_1 \to \beta_1 \to \alpha_2 \to \beta_2$ is a possible abstract trace. Since this does not contain all abstract timestamps, we cannot prove $\vdash_{term}$. This example is encoded in the logic program in appendix A.

3.4.5 A final concurrent client

$$\begin{array}{c} \alpha \text{ prefix} \\ \alpha_1\ \texttt{lock(x);} \\ \begin{array}{l||l} \alpha_2\ \texttt{lock(x);} & \texttt{unlock(x);}\ \beta \\ \alpha_3\ \texttt{unlock(x);} & \end{array} \end{array}$$

Without explicitly laying out the proof, we annotate the program with abstract timestamps for each operation. The only abstract trace is $\alpha \to \alpha_1 \to \beta \to \alpha_2 \to \alpha_3$. Note that $\alpha_1 \to \alpha_2$ is not permitted since the region trace for a must be consistent.

An example giving a more explicit description of the trace process can be found in appendix C.

4 Further Improvements

The initial logic showcased above preserves much of the modularity and proof style of the preceding TaDA based systems. However, it has several unsatisfactory limitations which limit the programs it is able to reason about. The following sections will highlight possible extensions which improve its expressivity. Many of these are orthogonal to the majority of the proof of soundness, but add significant computational complexity to the *nec-consist* check. It is likely that in order to tackle hard programs that require these extensions, the check will need to be mechanised.

4.1 More general modules

"We assume that for all $act \in p_1(\pi)$, taking $Q = p_3(act)$, for all $x \in p_2(act)$, $Q(x)$ maps to the same set (for convenience, we will refer to this set of states, as Y_Q)."

The previously discussed special case of region trace consistency bounds the considered modules to those which can be simulated by a "general region" defined as follows.

The general region **GRegion**'s states are the natural numbers, and its state transition system is defined as follows:

$$\forall n, m \in \mathbb{N}.\ \mathbf{0}\ :\ n \rightsquigarrow m$$

We define the operations across **GRegion** as NtoM where N and M are sets. For each N, M, the semantics of `NtoM(x)` is as follows: "block until the **GRegion** at `x` enters a state in N, then atomically and nondeterministically update the state of **GRegion** to one of the states in M."

For any program satisfying our special case which we can prove a non-terminating triple for, we can replace calls to operations on blocking modules with their equivalent NtoM function(s), assuming the states of the original regions are no more than countably infinite and return values are not checked. The resulting program can be proven to terminate iff the original one can.

Motivation: Any module function `f(x)` over blocking regions satisfying our special case will perform one or more actions of the form (r, X, Y, α). Simply replacing the function call with `XtoY(x)` for each action will generate an equivalent action ordering context for the whole program.

For example, $\{0\}$to$\{1\}$(x) and $\{0, 1\}$to$\{0\}$(x) together simulate a spin lock.

We can therefore directly consider clients calling `NtoM` functions as a way of motivating the expressivity of our logic. For example, we can prove the client given by

$$\begin{array}{r||l} \alpha_1\ \texttt{0to1(x);} & \texttt{1to2(x);}\ \beta_1 \\ \alpha_2\ \texttt{2to1(x);} & \texttt{1to0(x);}\ \beta_2 \end{array}$$

terminates.

What about functions that do not fit this special case? The simplest example is the counter module. We can trivially extend it to be blocking by adding the function `waitFor2(x)`, which blocks until the value of the counter is at least 2. We must make the counter region a blocking one in order to handle the unblocking condition. Now the ordinary `incr` operation must produce actions, since it is updating a blocking region. These actions will be of the form $(a, \mathbb{N}, Q, \alpha)$ but $Q(n) = \{n + 1\}$ for all n, which does not fit our special case (for reference, the actions generated by `waitFor2` would be of the form

$(a, \mathbb{N} \geq 10, id, \alpha))$.

We can relax our special case restriction on Q, but this will require us to redefine the consistency of a region trace to something that is harder to calculate.

For a general Q, the definition of consistency of a region trace is as follows:

Define $X_n = p_2(act_{(s_\pi|r)_n})$, that is, X_n is the X set of the n^{th} timestamped action in the region trace.

Define $Q_n = p_3(act_{(s_\pi|r)_n})$, that is, Q_n is the Q function of the n^{th} timestamped action in the region trace.

A region trace is consistent if $X_0 =\between$ and

$$\begin{array}{l}
\forall y_0 \in Q_0(\between).\ Q_1(y_0) \neq \emptyset \wedge X_1 \subseteq y_0 \\
\quad \forall y_1 \in Q_1(y_0).\ Q_2(y_1) \neq \emptyset \wedge X_2 \subseteq y_1 \wedge \\
\qquad \ldots \\
\qquad\quad \forall y_{n-1} \in Q_{n-1}(y_{n-2}).\ Q_n(y_{n-1}) \neq \emptyset \wedge X_n \subseteq y_{n-1}
\end{array}$$

This is not the easiest condition to calculate, but it does allow us to handle more interesting modules.

4.1.1 Even more general modules

For more generality, we must handle the possibility of the linearisation point of a module operation depending on the value of its arguments.

Consider the example above, where we extended a counter module with `waitFor2(x)` in order to make it blocking. Suppose instead that we extend it with the function `waitFor(x, n)`, which blocks until the program counter is at least `n`. Such a function could be implemented as follows:

```
function waitFor(x, n) {
  v := [x];
  while (v < n) {
    v := [x];
  }
}
```

The linearisation point of this function is at the point the loop exits. An action must be generated to encode the unblocking condition.

For a general action (r, X, Q, α), X is the set of possible values of the region immediately before the linearisation point. However, this now varies depending on the value of `n`. So in general X must be a function of the program arguments.

We can define $X(\mathtt{n})$ as $\mathbb{N} \geq \mathtt{n}$ and then give rules for binding the value of `n` to the set of possible values of the argument at the call site (call this Z) by making each Action a 5-tuple of (r, X, Q, Z, α). The definition of region trace consistency (assuming Z_n can refer to n^{th} Z set in sequence) would then be

$$
\begin{array}{l}
\forall z_0 \in Z_0, z_1 \in Z_1 ... z_{n-1} \in Z_{n-1}, z_n \in Z_n.\\
\quad \forall y_0 \in Q_0(\between).\ Q_1(y_0) \neq \emptyset \wedge X_1(z_1) \subseteq y_0 \wedge\\
\quad\quad \forall y_1 \in Q_1(y_0).\ Q_2(y_1) \neq \emptyset \wedge X_2(z_2) \subseteq y_1 \wedge\\
\quad\quad\quad ...\\
\quad\quad\quad\quad \forall y_{n-1} \in Q_{n-1}(y_{n-2}).\ Q_n(y_{n-1}) \neq \emptyset \wedge X_n(z_n) \subseteq y_{n-1}
\end{array}
$$

4.1.2 Most general modules

For ultimate generality, we must handle the case where the unblocking condition of a module operation depends on the value of its arguments. A sketch of an extension to do so is given.

Consider the following very silly extension to the counter module:

```
function hangIfLess(x, n) {
  v := [x];
  while (v < n) {}
}
```

Now Q must not only depend on the state of the counter, also on the value of `n`. We can define $Q(s, \mathtt{n})$ as

$$
\begin{array}{ll}
id & \text{if } s \geq \mathtt{n}\\
\emptyset & \text{otherwise}
\end{array}
$$

Obviously the proof rules would have to be updated to deal with the new arity of Q. For example, the definition of region trace consistency would need to be extended as follows:

$$\begin{array}{l}
\forall z_0 \in Z_0, z_1 \in Z_1 ... z_{n-1} \in Z_{n-1}, z_n \in Z_n. \\
\quad \forall y_0 \in Q_0(\between, z_0).\ Q_1(y_0, z_1) \neq \emptyset \wedge X_1(z_1) \subseteq y_0 \wedge \\
\quad\quad \forall y_1 \in Q_1(y_0, z_1).\ Q_2(y_1, z_2) \neq \emptyset \wedge X_2(z_2) \subseteq y_1 \wedge \\
\quad\quad\quad ... \\
\quad\quad\quad\quad \forall y_{n-1} \in Q_{n-1}(y_{n-2}, z_{n-1}).\ Q_n(y_{n-1}, z_n) \neq \emptyset \wedge X_n(z_n) \subseteq y_{n-1}
\end{array}$$

4.2 Improving loops

As previously mentioned, the current proof rules do not allow the bodies of while loops to have non-empty action ordering contexts. Consider the following program.

```
v := 10;             ||
while (v > 0) {      ||
  lock(x);           ||   lock(x);
  unlock(x);         ||   unlock(x);
  v := v − 1;        ||
}                    ||
```

We cannot prove that this program terminates without re-writing the left thread by unrolling the loop. The reason we have this restriction is that we have no ability to encode actions as an invariant; we must be able to exactly enumerate the updates to blocking regions that occur.

Note that this is only a restriction on blocking regions, we can still conduct every proof in the extended system that was possible in the original Total TaDA, by making every region have a non-blocking type.

One solution to this problem is to extend our action ordering contexts to contain something like a fixpoint. For example, we could have the following alternative *while*1 rule:

$$\frac{\forall \gamma \leq \alpha.\, \pi; \lambda; \mathcal{A} \vdash \Big\{ P(\gamma) \Big\}\ \mathbb{C}\ \Big\{ \exists \beta.\, P(\beta) \wedge \beta < \alpha \Big\}}{\mathsf{fix}(\pi); \lambda; \mathcal{A} \vdash \Big\{ P(\alpha) \Big\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \Big\{ \exists \beta.\, P(\beta) \wedge \alpha \geq \beta \wedge \neg\mathbb{B} \Big\}}\ While1$$

where $\mathsf{fix}(\pi)$ encodes that π will be executed a finite but arbitrary (possibly 0) number of times sequentially. To prove *nec-consist*($\mathsf{fix}(\pi)$), a requirement for termination, we would have to perform an induction to show that for each possible number of executions n, *nec-consist*$(pi; pi...)$ holds.

To improve the computability of the check, we can observe that the number of iterations is bounded by α. Encoding this bound, i.e. $\mathsf{fix}_\alpha(\pi)$, would allow us to avoid the induction, since we now simply encode a case analysis of every possible unrolling up to α.

This extension is sufficient to prove the termination of the code above, however nested $\mathsf{fix}_\alpha(\pi)$s in the program ordering context would result in a blow-up during the unrolling, analogous to the issues encountered in bounded model checking of nested loops.

4.2.1 If branching

Similarly, the addition of a bounded fix allows us to relax our restriction on the branches of if statements. If we extend the syntax of fix to explicitly track its unrolled value e.g. $\mathsf{fix}_{n \leq \alpha}$, we can model the actions π occurring in the if and else branch as $\mathsf{fix}_{n \leq 1}(\pi)$ and $\mathsf{fix}_{1-n}(\pi)$ respectively. A more general if rule might be

$$\dfrac{\begin{array}{cc} b = P \Rightarrow \neg\mathbb{B} \;?\; 0 : 1 & b' = P \Rightarrow \mathbb{B} \;?\; 0 : 1 \\ \pi; \lambda; \mathcal{A} \vdash \left\{P \land \mathbb{B}\right\} \mathbb{C}_1 \left\{Q\right\} & \pi'; \lambda; \mathcal{A} \vdash \left\{P \land \neg\mathbb{B}\right\} \mathbb{C}_2 \left\{Q\right\} \end{array}}{\mathsf{fix}_{n \leq b}(\pi) \cup \mathsf{fix}_{(1-n) \land b'}(\pi); \lambda; \mathcal{A} \vdash \left\{P\right\} \texttt{if } (\mathbb{B})\ \mathbb{C}_1 \texttt{ else } \mathbb{C}_2 \left\{Q\right\}} \; If$$

5 The model

See appendix B for a full description of the original model of TaDA, reproduced from the technical report [19]. The model of B-TT is presented in terms of modifications to this existing model.

We give a full formalisation for the original, restricted rules, although this is more for brevity than because of any greater difficulty. Proving the version of our rules allowing extended model behaviour would require trivial modifications to the definition of *happens* and privative atomic satisfaction. Introducing fix for more general loops would leave the model almost unchanged, but would add tedious housekeeping to the proof of soundness for little benefit.

The $a \Mapsto \blacklozenge$ assertion in the syntactic proof is taken as shorthand for $\exists n.\, a \Mapsto \blacklozenge_n$

We define the timestamp valuation context $\mathcal{O} \to \mathbb{N}^+$, written at, which maps abstract timestamps to the concrete amount of time remaining before they must occur. We say $consist(\pi, at)$ if $(\alpha, \beta) \in p_2(\pi) \Rightarrow at(\alpha) < at(\beta)$.

The atomic tracking separation algebra is redefined to be

$$(\mathsf{AState} \times \mathsf{AState}) \uplus \bigcup_{n \in \mathbb{N}} \{\blacklozenge_n\} \uplus \{\lozenge\}$$

The rely relation is redefined to be

$$\frac{g \# g' \quad (s, s') \in \mathcal{T}_{t(n)}(g')^* \quad \begin{array}{l}(d(a) \in \{\blacklozenge_n, \lozenge\} \Rightarrow s' \in \mathrm{dom}(\mathcal{A}(a))) \\ (d(a) = \blacklozenge_0 \wedge \mathsf{blocking}(a) \Rightarrow \mathcal{A}(a)[s'] \neq \emptyset)\end{array}}{(r[a \mapsto n], h, b, \gamma[a \mapsto g], \rho[a \mapsto s], d) \ \mathrm{R}_{\mathcal{A}} \ (r[a \mapsto n], h, b, \gamma[a \mapsto g], \rho[a \mapsto s'], d)}$$

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

The second rule is identical to TaDA. The first rule encodes that, in addition to the usual meaning in TaDA, holding $\blacklozenge_0$ guarantees that the environment will not cause the local thread to block.

The guarantee relation is redefined to be

$$\begin{array}{l} \varphi \ \mathrm{G}_{\lambda;\mathcal{A}} \ \varphi' \overset{\text{def}}{\iff} \forall a. \, (\exists \lambda' \geq \lambda. \, r_\varphi(a) = (\lambda', -, -)) \implies \rho_\varphi(a) = \rho_{\varphi'}(a) \wedge \\ \qquad \forall a \in \mathrm{dom}\,\mathcal{A}. \left(\begin{array}{r} \left(\begin{array}{r} d_\varphi(a) = d_{\varphi'}(a) \wedge d_\varphi(a) \neq \blacklozenge_{>0} \wedge \\ \rho_\varphi(a) = \rho_{\varphi'}(a) \end{array} \right) \vee \\ \left(\begin{array}{r} d_\varphi(a) = \blacklozenge_n \wedge d_{\varphi'}(a) = \blacklozenge_{n'} \wedge \\ 0 \leq n' < n \wedge \rho_\varphi(a) = \rho_{\varphi'}(a) \end{array} \right) \vee \\ \left(\begin{array}{r} d_\varphi(a) = \blacklozenge_n \wedge d_{\varphi'}(a) = (\rho_\varphi(a), \rho_{\varphi'}(a)) \\ \wedge \, (\rho_\varphi(a), \rho_{\varphi'}(a)) \in \mathcal{A}(a) \end{array} \right) \end{array} \right) \end{array}$$

This modification is crucial to several soundness proofs. It guarantees that every world-to-world step the local thread makes will strictly decrease the values of any $\blacklozenge_n$ held towards $\blacklozenge_0$. This means that after enough execution, every $\blacklozenge_n$ will become $\blacklozenge_0$. Holding $\blacklozenge_0$ gives us extra information about the stability and state of the associated region.

The primitive atomic satisfaction judgement is redefined to be

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

where Env is defined as

$$\begin{aligned}
\varphi Env \varphi' \overset{\text{def}}{\iff} & \varphi = (r, h_1, b_1, \gamma_1, \rho) \wedge \varphi' = (r, h_1, b_1, \gamma_1, \rho') \wedge \\
& \forall a.\, (\exists \lambda' \geq \lambda.\, r_{\varphi}(a) = (\lambda', -, -)) \implies \rho(a) = \rho'(a) \wedge \\
& \forall a \notin \operatorname{dom} \mathcal{A}.\, \rho(a) = \rho'(a) \wedge \\
& \forall a \in \operatorname{dom} \mathcal{A}. \left(\begin{array}{c}
\left(\begin{array}{r} d_{\varphi}(a) = \blacklozenge_0 \wedge \\ \mathsf{blocking}(a) \end{array} \right) \Rightarrow \left(\begin{array}{r} \left(\begin{array}{r} \mathcal{A}(a)[\rho(a)] \neq \emptyset \wedge \\ \rho(a) = \rho'(a) \end{array} \right) \vee \\ \left(\mathcal{A}(a)[\rho'(a)] \neq \emptyset \right) \end{array} \right) \\
\wedge \\
d_{\varphi}(a) \neq \blacklozenge_0 \vee \neg \mathsf{blocking}(a) \Rightarrow \left(\rho(a) = \rho'(a) \right)
\end{array} \right)
\end{aligned}$$

That is, if we are at a world with $\blacklozenge_0$, the environment will atomically unblock the associated region.

We define *(*k-step execution), $\xrightarrow{[a]_k}_k$ as

$$\begin{aligned}
\langle \mathbb{C}, s \rangle \xrightarrow{[a]_k}_k \langle \mathbb{C}', s' \rangle \overset{\text{def}}{\iff} & \exists k' \leq k, \mathbb{C}_1 ... \mathbb{C}_{k'-1}, s_1 ... s_{k'-1}. \\
& [a]_k = [\alpha_1 ... \alpha_{k'}] \wedge \langle \mathbb{C}, s \rangle \xrightarrow{\alpha_1} \langle \mathbb{C}_1, s_1 \rangle ... \langle \mathbb{C}_{k'-1}, s_{k'-1} \rangle \xrightarrow{\alpha_{k'}} \langle \mathbb{C}', s' \rangle \\
& \wedge \mathbb{C}' \neq \mathtt{skip} \Rightarrow k' = k
\end{aligned}$$

We define the relation *happens* : $\mathsf{View}_{\mathcal{A}} \times \mathsf{View}_{\mathcal{A}} \times \mathcal{P}(\mathsf{Action}_{\mathcal{O}})$ as

$$\begin{aligned}
happens(p, q, act_{\mathcal{O}}) \overset{\text{def}}{\iff} & \forall r \in \mathsf{View}_{\mathcal{A}}, \varphi \in p * r, \varphi' \in q * r. \\
& \forall (a, X, Y_Q, \_) \in act_{\mathcal{O}}.\, \rho_{\varphi}(a) \in X \wedge \rho_{\varphi'}(a) \in Y \wedge \\
& a \notin \operatorname{dom} d_{\varphi'}
\end{aligned}$$

happens keeps track of the updates between two worlds that are not in the atomic tracking components of the worlds.

We define the relation $agree : \mathsf{AProgram} \times \mathsf{AProgram} \times \mathcal{P}(\mathsf{Action}_{\mathcal{O}})$ as

$$\begin{aligned} agree(\pi, \pi', act_{\mathcal{O}}) \overset{\text{def}}{\iff} & \forall (a, X, Y) \in act_{\mathcal{O}}. \\ & \quad \exists \alpha.\, (a, X, Y, \alpha) \in min(\pi) \land (a, X, Y, \alpha) \notin \pi' \land \\ & \quad\quad \forall \beta \neq \alpha.\, \exists X', Y'.\, (a, X', Y', \beta) \in \pi \Rightarrow (a, X', Y', \beta) \in \pi' \end{aligned}$$

This encodes that the actions that are removed between π and π' are a superset of $act_{\mathcal{O}}$.

Semantic judgements are defined as follows

The semantic judgement

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

where

- $\pi \in \mathsf{AProgram}$ is an action ordering context;
- at is a timestamp valuation context such that $consist(\pi, at)$;
- $\lambda \in \mathsf{Level}$ 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 $p'_p \in \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$, $p''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathrm{dom}\,\mathcal{A}}$, $\pi', \pi'' \in \mathsf{AProgram}$ such that

  $$\lambda; \mathcal{A} \vDash \langle p_p(s) * p(\mathbf{x})\rangle a \langle p'_p(s') * p(\mathbf{x}) \vee \exists \mathbf{y} \in Q(\mathbf{x}).\, p''_p(\mathbf{x}, \mathbf{y}, s') * q(\mathbf{x}, \mathbf{y})\rangle$$

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

  and for all $\mathbf{y} \in Q(\mathbf{x})$, $at''; \pi''; \lambda; \mathcal{A}; \Omega(\mathbf{x}, \mathbf{y}) \vDash \left\{ p''_p(\mathbf{x}, \mathbf{y}) \right\}\ \mathbb{C}'\ \left\{ q_p(\mathbf{x}, \mathbf{y}) \right\}$.

  Furthermore, $\pi', \pi'' \subseteq \pi$ and for all $act_{\mathcal{O}}, act'_{\mathcal{O}}$ such that
  $happens(p_p(s) * p(\mathbf{x}), p'_p(s') * p(\mathbf{x}), act_{\mathcal{O}})$,
  $happens(p_p(s) * p(\mathbf{x}), \exists \mathbf{y} \in Q(\mathbf{x}).\, p''_p(\mathbf{x}, \mathbf{y}, s') * q(\mathbf{x}, \mathbf{y}), act'_{\mathcal{O}})$
  we have $agree(\pi, \pi', act_{\mathcal{O}})$ and $agree(\pi, \pi'', act'_{\mathcal{O}})$

  In addition, for all $(a, X, Y_Q, \alpha) \in p_1(\pi)$ such that $at(\alpha) = 1$, $(a, X, Y_Q, \alpha) \notin p_1(\pi') \cup p_1(\pi'')$

  Finally, if $\mathbb{C}' = \texttt{skip}$, $\pi' = \pi'' = \emptyset$.

  at' is defined as $\forall (a, X, Y_Q, \alpha) \in p_1(\pi').\, at'(\alpha) = at(\alpha) - 1$
  at'' is defined as $\forall (a, X, Y_Q, \alpha) \in p_1(\pi'').\, at''(\alpha) = at(\alpha) - 1$

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

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

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

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

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

6 Soundness

The non-trivial rule proofs follow:

6.0.1 Proof of while2

Let α be an ordinal. Assume:

$$T = X \setminus Q^{-1}(\emptyset) \qquad \mathcal{A} = a : x \in X \rightsquigarrow Q(x), \mathcal{A}' \qquad \mathsf{blocking}(a)$$

$$\forall \gamma \le \alpha.\, \emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x, \gamma) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n.\, a \Mapsto \blacklozenge_n \land \mathbb{B} \end{array} \right\} \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \gamma \ge \beta \end{array} \right\} \tag{1}$$

$$\forall \gamma \le \alpha.\, \emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in T.\, P(t, \gamma) * \\ \quad \mathbf{t}_a^\lambda(t) * \exists n.\, a \Mapsto \blacklozenge_n \land \mathbb{B} \end{array} \right\} \mathbb{C} \left\{ \begin{array}{l} \exists x \in T, \beta.\, P(t, \beta) * \\ \quad \mathbf{t}_a^\lambda(t) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \gamma > \beta \end{array} \right\} \tag{2}$$

We must prove

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x, \alpha) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n.\, a \Mapsto \blacklozenge_n \end{array} \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \alpha \ge \beta \land \neg \mathbb{B} \end{array} \right\}$$

It is sufficient to show

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x, \alpha) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n.\, a \Mapsto \blacklozenge_n \end{array} \land \mathbb{B} \right\} \mathbb{C}; \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \\ \quad \alpha \ge \beta \land \neg \mathbb{B} \end{array} \right\} \tag{3}$$

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x, \alpha) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n.\, a \Mapsto \blacklozenge_n \end{array} \land \neg \mathbb{B} \right\} \texttt{skip} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \alpha \ge \beta \land \neg \mathbb{B} \end{array} \right\} \tag{4}$$

since these are the two reduction cases in the small-step semantics. Clearly the second case is trivially true.

For the first case, we will prove that after enough executions of the body, we can show that the loop terminates via trans-finite induction.

By (1) and the sequencing lemma, we have

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \alpha \ge \beta \end{array} \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \\ \quad \alpha \ge \beta \land \neg \mathbb{B} \end{array} \right\} \tag{5}$$

Furthermore, from the definition of the guarantee relation, we know $n = n'' = 0 \lor n > n''$.

Again, we know (5) will expand in one of two ways:

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \alpha \geq \beta \end{array} \right\} \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \\ \quad \alpha \geq \beta \land \neg \mathbb{B} \end{array} \right\} \tag{6}$$

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \alpha \geq \beta \end{array} \right\} \texttt{skip} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \alpha \geq \beta \land \neg \mathbb{B} \end{array} \right\} \tag{7}$$

(7) is again trivial. For (8), take the case where $n'' > 0$. We can "run" the body again via (1) and the sequencing rule, giving us the same assertion as (5) except with some $n''' < n''$. We can continue doing this until we reach 0. So let us assume that $n'' = 0$, since we can just run until it becomes 0 otherwise.

By expanding (5) to (6), we are implicitly making the argument against the small step semantics and the definition of the primitive atomic satisfaction judgement. Since we have

$$\lambda'; \mathcal{A} \vDash \left\langle \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \\ \alpha \geq \beta \end{array} \right\rangle \texttt{skip} \left\langle \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \\ \alpha \geq \beta \end{array} \right\rangle$$

we have

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n''.\, a \Mapsto \blacklozenge''_n \land \alpha \geq \beta \end{array} \right\} \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \\ \quad \alpha \geq \beta \land \neg \mathbb{B} \end{array} \right\}$$

However, in the case that $n'' = 0$, the following judgement holds:

$$\lambda'; \mathcal{A} \vDash \left\langle \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge_0 \land \\ \alpha \geq \beta \end{array} \right\rangle \texttt{skip} \left\langle \begin{array}{l} \exists x \in T, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge_0 \land \\ \alpha \geq \beta \end{array} \right\rangle$$

and therefore we have

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in T, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * a \Mapsto \blacklozenge_0 \land \alpha \geq \beta \end{array} \right\} \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^{\lambda}(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \\ \quad \alpha \geq \beta \land \neg \mathbb{B} \end{array} \right\} \tag{8}$$

Now we can perform a trans-finite induction. We can assume the following holds for all $\gamma < \alpha$:

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x, \gamma) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n.\, a \Mapsto \blacklozenge_n \end{array} \right\} \mathtt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \gamma \geq \beta \wedge \neg\mathbb{B} \end{array} \right\}$$

By applying the the sequencing lemma and (2) to (8), we get

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in T, \beta'.\, P(x, \beta') * \\ \quad \mathbf{t}_a^\lambda(t) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \beta > \beta' \end{array} \right\} \mathtt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \\ \quad \alpha \geq \beta \wedge \neg\mathbb{B} \end{array} \right\}$$

Now, since $T \subseteq X$, we can apply the consequence lemma, to obtain

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(t) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \alpha > \beta \end{array} \right\} \mathtt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X, \beta.\, P(x, \beta) * \\ \quad \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \\ \quad \alpha \geq \beta \wedge \neg\mathbb{B} \end{array} \right\}$$

This holds by the inductive hypothesis. Therefore we have succeeded.

6.0.2 Proof of while3

Assume

$$\mathcal{A} = a : x \in X \rightsquigarrow Q(x), \mathcal{A}' \qquad \mathsf{blocking}(a)$$

$$T = X \setminus Q^{-1}(\emptyset) \qquad \forall a \in \mathrm{dom}(\mathcal{A}').\, \neg\mathsf{blocking}(a) \qquad T \subseteq X' \subseteq X$$

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ \exists n.\, a \Mapsto \blacklozenge_n \wedge \mathbb{B} \end{array} \right\} \mathbb{C} \left\{ \begin{array}{l} (\exists x \in X.\, PP(x) * \mathbf{t}_a^\lambda(x) * \exists n'.\, a \Mapsto \blacklozenge_{n'} \wedge \mathbb{B}) \vee \\ \left(\begin{array}{l} \exists x \in X', \exists y \in Q(x). \\ \quad R(x, y) * a \Mapsto (x, y) \wedge \neg\mathbb{B} \end{array} \right) \end{array} \right\} \tag{9}$$

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \exists x \in T.\, P(x) * \mathbf{t}_a^\lambda(x) * a \Mapsto \blacklozenge \wedge \mathbb{B} \right\} \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x, y) * a \Mapsto (x, y) \wedge \neg\mathbb{B} \end{array} \right\} \tag{10}$$

We must prove

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ \exists n.\, a \Mapsto \blacklozenge_n \wedge \mathbb{B} \end{array} \right\} \mathtt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x, y) * a \Mapsto (x, y) \wedge \neg\mathbb{B} \end{array} \right\} \tag{11}$$

It is sufficient to show

$$\emptyset; \lambda; \mathcal{A} \vdash \left\{ \begin{array}{l} \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ \exists n.\, a \Mapsto \blacklozenge_n \wedge \mathbb{B} \end{array} \right\} \mathbb{C}; \mathtt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x, y) * a \Mapsto (x, y) \wedge \neg\mathbb{B} \end{array} \right\} \tag{12}$$

since the other expansion case holds trivially.

We can apply the sequencing lemma and (9) to (12) to get

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \begin{array}{l} \left(\begin{array}{l} \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \mathbb{B} \end{array} \right) \lor \\ \left(\begin{array}{l} \exists x \in X', \exists y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right) \end{array} \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \tag{13}$$

Unrolling and applying the consequence lemma gives us two cases (again, by the definition of the guarantee relation, $n' < n$):

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X.\, P * \mathbf{t}_a^\lambda(x) * \\ \exists n'.\, a \Mapsto \blacklozenge_{n'} \land \mathbb{B} \end{array} \right\} \mathbb{C}; \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \tag{14}$$

$$\emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X', \exists y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \texttt{skip} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \tag{15}$$

Clearly (15) holds trivially. For (14), which we can see is the same as (12), we can again inductively argue that, by continually applying the sequencing lemma and (9), we either reach the (15) case, or n' will reach 0.

If n' reaches 0, we have

$$\lambda'; \mathcal{A} \vDash \left\langle \begin{array}{l} \exists x \in X.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ a \Mapsto \blacklozenge_0 \land \mathbb{B} \end{array} \right\rangle \texttt{skip} \left\langle \begin{array}{l} \exists x \in T.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ a \Mapsto \blacklozenge_0 \land \mathbb{B} \end{array} \right\rangle$$

And therefore we have

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in T.\, P(x) * \mathbf{t}_a^\lambda(x) * \\ a \Mapsto \blacklozenge_0 \land \mathbb{B} \end{array} \right\} \mathbb{C}; \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \tag{16}$$

Applying the sequencing lemma and (10), we have

$$\emptyset; \lambda; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\} \texttt{while } (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \land \neg\mathbb{B} \end{array} \right\}$$

Unrolling and applying the consequence lemma gives us

$$\begin{array}{l} \text{ﬄ} \quad \emptyset; \lambda'; \mathcal{A} \vDash \left\{ \bot \right\} \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg \mathbb{B} \end{array} \right\} \\ \hline \text{ﬄ} \quad \emptyset; \lambda'; \mathcal{A} \vDash \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg \mathbb{B} \end{array} \right\} \texttt{skip} \left\{ \begin{array}{l} \exists x \in X', y \in Q(x). \\ \quad R(x,y) * a \Mapsto (x,y) \wedge \neg \mathbb{B} \end{array} \right\} \end{array}$$

Both of these cases hold trivially. Therefore we have succeeded.

6.0.3 Proof of make atomic

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

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

then

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

This lemma is stated without proof in the original TaDA paper.

Lemma 2. *If, for* $p \in \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $q, \omega \in \coprod_{x \in X} Q(x) \to \mathsf{View}_{\mathrm{dom}(\mathcal{A})}$, $x \in X$, $y \in Q(x)$

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

then

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

where $\pi = \pi' \cup (a, X, Q, \alpha)$

This lemma follows necessarily from the definition of a world with atomicity tracking and *happens*. The consequent represents the view in which the region is updated in the world instead of in the atomicity tracking. Since one fewer action is registered in the atomicity tracking component, one more action must be registered in π. Due to time constraints, a fully detailed proof of this lemma was not completed.

Assume

$$\begin{gathered} a \notin \mathcal{A} \quad X' \subseteq X \quad \{(x, y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \underline{\textbf{if}}\ \mathsf{blocking}(t)\ \underline{\textbf{then}}\ \pi = (\{(a, X', Q, \alpha)\}, \emptyset) \cup \pi'\ \underline{\textbf{else}}\ \pi = \pi' \\ \exists act'.\, act'; \pi'; \lambda'; \mathcal{A} \vdash \begin{array}{c} \left\{p_p * \exists x \in X.\, \mathbf{t}_a^{\lambda}(x) * a \Rrightarrow \blacklozenge\right\} \\ \mathbb{C} \\ \{\exists x \in X', y \in Q(x).\, q_p(x, y) * a \Rrightarrow (x, y)\} \end{array} \end{gathered}$$

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

We must prove

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

The majority of the proof is identical to the original TaDA proof, and so for the most part this proof is reproduced from the paper. Our new obligation is to show that act and π satisfy the obligations on them laid out in the semantics.

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

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

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

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

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

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

Let $\overline{r} = r * [\mathrm{G}]_a * a \Rrightarrow -$. ($\overline{r}$ 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 $\overline{\varphi} = \varphi \bullet a \Rrightarrow \blacklozenge$. By construction, $\lfloor \varphi \rfloor_\lambda = \lfloor \overline{\varphi} \rfloor_\lambda$. We have that $\overline{\varphi} \in (p_p(s) * \exists x \in X.\, \mathbf{t}_a^{\lambda}(x) * a \Rrightarrow \blacklozenge) * \overline{r}$.

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

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

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

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

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

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

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

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

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

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

The remaining cases are simpler, or follow similar reasoning.

6.0.4 Proof of terminates

Due to time constraints, we do not attempt a full formal proof of the correctness of our procedure to check the allowed abstract traces. We give a sketch of how such a proof might be carried out.

Given a $\mathbb{C}$ with an associated π under our model, we know that, since the environment unblocks us finitely, we will have a k-step execution to `skip` for some k, which will always carry out every action in π.

We know, because of the *nec-consist* side condition, that every abstract trace of $\mathbb{C}$ is consistent with respect to every region's transition system and total over the set of π's abstract timestamps.

This should be enough to show that after every transition, we are inevitably left in exactly the correct state for the next transition to happen, without needing special action from the environment. We do this by matching up the next blocking action made by $\mathbb{C}$ with a subset of the traces identified by *nec-consist*, and showing that subsequent actions conform to these traces by induction.

7 Final evaluation

The objective of this project was to extend Total TaDA to prove the termination of blocking algorithms without losing the properties of TaDA that make it a useful logic; i.e. modularity and abstraction. This core objective was met, with some caveats. The initial extension proposed has several flaws; it restricts both the permitted region state transitions, and how module operations may be used (e.g. in loops). We presented and commented on extensions that incrementally relax these restrictions.

Arguably, some of these extensions introduce checks which are not tractable for humans to verify, but this is already an existing trend in separation logic as researchers continue to push the envelope in facilitating expressive proofs about concurrent programs. TaDA already suffers from this, with complex proof rules and subtle arguments about stability that leave a lot of room for human error. Through the logic program encoding in appendix A, we sketch how some of the more onerous checks of our system may be mechanised.

Considering our proof system with all extensions, the biggest remaining flaw is that we cannot reason about potentially infinite traces of actions. Guaranteeing that only finite actions are performed by each thread is a necessary condition for proving termination of deadlock-free operations anyway, but is stronger than necessary for starvation-free ones. For example, we cannot prove that the code of fig. 31 terminates (only) with a starvation-free lock. In the final section, we will discuss further work which could allow us to reason about infinite traces, and thus take advantage of the weaker restrictions starvation-free operations place on the environment.

For the most part, we maintain abstraction of the internal implementation of modules. TaDA itself breaks abstraction in a minor way since abstract predicates must expose the IDs of their internal regions. Our action system relies on this to uniquely identify actions, ingraining this further into the logic. Furthermore, if we have nesting blocking regions, each must generate its own action, since in general the inner regions could also be referenced by a different region.

Aside from this, the logic maintains most of the proof style of TaDA. We

```
                [v] := 0;
b := 0;              ||
while (b = 0) {      ||
  lock(x);           ||  lock(x);
  unlock(x);         ||  unlock(x);
  b := [v];          ||  [v] := 1;
}                    ||
```

Figure 31: Terminates if lock is starvation free.

maintain the thread modularity and compositionality that make TaDA an attractive, powerful system. We allow blocking regions to be layered freely with other regions and abstract predicates.

7.1 Comparison with existing work

Even with the caveats and restrictions we have laid out, in many ways this system represents a significant step forward from previous work on blocking termination. No other logic exists which can prove termination of blocking algorithms in the client-module paradigm while maintaining abstraction.

Earlier, we discussed the limiting assumptions made by each of the two most prominent blocking termination logics. LiLi assumes a fixed number of threads, cannot prove clients, and cannot easily prove module operations that leave the module in a blocked state. Our new system does not have any of these issues. Due to the last limitation mentioned, LiLi would certainly struggle to prove even an implementation of the “general region” that characterises the module space of our logic without extensions. We give such a proof in appendix C.

Müller’s work does not handle the specification of arbitrarily blocking module functions. We can, and furthermore we do so for a language that allows arbitrary manipulation of shared memory, instead of restricting the language to message passing in order to manage the interference of concurrent threads.

8 Closing thoughts and future work

As previously mentioned, this project is able to reason about the total correctness of modules that no other existing logic can, in a way that preserves compositionality, modularity, and abstraction.

TaDA, the logic on which Total TaDA, and thus B-TT, is based, has the fundamental limitation that it is unable to prove programs that involve helping, where another thread may complete the local thread's state transition on its behalf. An extension to TaDA is in development which extends TaDA to handle helping, and when it is complete it would be valuable to investigate if the extensions of B-TT could be ported onto it.

There is still work to be done in lifting some of the remaining restrictions on the expressivity of the logic. In the extended logic for more general regions, the chain of Q state transformation functions which are built up from the initial state can effectively be thought of as a continuation structure. It is possible that reformulating the action ordering context into an explicit continuation would allow us to express non-determinism, iteration, and infinite traces more neatly.

In closing, B-TT opens up a wider class of terminating program for consideration than any previous logic is able to handle. We fully intend to continue work on it to deliver a conference-worthy contribution.

9 Bibliography

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[2] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, (Washington, DC, USA), pp. 55–74, IEEE Computer Society, 2002.

[3] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, pp. 279–285, May 1976.

[4] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP*, 1983.

[5] P. W. OHearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, pp. 271–307, Apr. 2007.

[6] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, (Berlin, Heidelberg), pp. 504–528, Springer-Verlag, 2010.

[7] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, "Steps in modular specifications for concurrent modules (invited tutorial paper)," *Electronic Notes in Theoretical Computer Science*, vol. 319, pp. 3 – 18, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, July 1990.

[9] V. Vafeiadis, "Automatically proving linearizability," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, (Berlin, Heidelberg), pp. 450–464, Springer-Verlag, 2010.

[10] G. Boudol and I. Castellani, "Concurrency and atomicity," *Theor. Comput. Sci.*, vol. 59, pp. 25–84, July 1988.

[11] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 637–650, ACM, 2015.

[12] I. Sergey, A. Nanevski, and A. Banerjee, "Specifying and verifying concurrent algorithms with histories and subjectivity," in *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, (New York, NY, USA), pp. 333–358, Springer-Verlag New York, Inc., 2015.

[13] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, *TaDA: A Logic for Time and Data Abstraction*, pp. 207–231. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[14] R. Floyd, “Assigning meanings to programs,” in *Proceedings of the American Mathematical Society Symposia on Applied Mathematics 19*, pp. 19–31, 1967.

[15] P. da Rocha Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland, “Modular termination verification for non-blocking concurrency,” in *ECOOP*, 2015.

[16] M. Herlihy and N. Shavit, “On the nature of progress,” in *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS’11, (Berlin, Heidelberg), pp. 313–328, Springer-Verlag, 2011.

[17] H. Liang and X. Feng, “A program logic for concurrent objects under fair scheduling,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, (New York, NY, USA), pp. 385–399, ACM, 2016.

[18] P. Boström and P. Müller, “Modular verification of finite blocking in non-terminating programs,” in *ECOOP*, 2015.

[19] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, “Tada: A logic for time and data abstraction. tech. rep., imperial college london (2014).”

[20] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, “Views: compositional reasoning for concurrent programs,” in *POPL*, pp. 287–300, 2013.

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

Appendices

A Encoding nec-consist as a logic program

We give an equivalent condition to the previously defined one which is more amenable to encoding.

Take the set of abstract timestamps of π, $\mathcal{O}_\pi$. Assert: for all $\alpha \in \mathcal{O}_\pi$ there is exactly one $act \in p_1(\pi)$ with $p_4(act) = \alpha$. We refer to this action as act_α.

Consider an asymmetric, deterministic binary relation o_π across $(\mathcal{O}_\pi \uplus \{start, end\})^2$.

Take the transitive closure of o_π, written as o_π^*. We define $o_\pi|a$ the "region trace" for region id a as follows:

$$\{(\alpha_1, \alpha_2) \mid o_\pi^*(\alpha_1, \alpha_2) \;\wedge\; sameregion(\alpha_1, \alpha_1, \alpha_2) \;\wedge\; (\neg\exists\alpha_3.\, o_\pi^*(\alpha_1, \alpha_3) \;\wedge\; o_\pi^*(\alpha_3, \alpha_2) \\ \wedge\; sameregion(\alpha_1, \alpha_3, \alpha_2))\}$$

where $sameregion(\alpha_1, \alpha_2, \alpha_3) \triangleq \alpha_1 = start \vee \alpha_3 = end \vee p_1(act_{\alpha_1}) = p_1(act_{\alpha_2}) = p_1(act_{\alpha_3}) = a$.

Assert: $o_\pi|a$ is asymmetric and antitransitive.

We define the "consistency" of a region trace. We assume that for all $act \in p_1(\pi)$, taking $Q = p_3(act)$, for all $x \in p_2(act)$, $Q(x)$ maps to the same set (for convenience, we will refer to this set of states, as Y_Q).

A region trace $o_\pi|a$ is consistent if $o_\pi|a(\alpha_1, \alpha_2)$ implies that for all α_1, α_2, $Y_{p_3(act_{\alpha_1})} \subseteq p_2(act_{\alpha_2}) \vee (\alpha_1 = start \wedge p_2(act_{\alpha_2}) = \emptyset) \vee \alpha_2 = end$.

We say o_π is an "abstract trace" of π if it is a maximal relation such that the following conditions are met:

- $o_\pi(\alpha_1, \alpha_2) \Rightarrow o_\pi^*(start, \alpha_1) \vee \alpha_1 = start$
- $o_\pi^*(\alpha_1, \alpha_2) \Rightarrow \neg(\alpha_2 <_\pi \alpha_1)$
- $o_\pi(\alpha_1, \alpha_2) \Rightarrow (\forall \alpha_3 \neq \alpha_1.\, (\alpha_3 <_\pi \alpha_2) \Rightarrow o_\pi^*(\alpha_3, \alpha_1))$
- for all a, $o_\pi|a$ is consistent.

. *nec-consist*(π) holds iff every abstract trace o_π of π has $o_\pi^*(start, end)$.

The logic program encoding of this condition is as follows. Each answer set of the program represents a possible counterexample. The condition is proven if the program has no answer sets.

```
time(start).
time(end).

gt(A, C) :- time(A), time(B), time(C), gt(A, B), gt(B, C).

sameRegion(A, B):- region(R), time(A), time(B),
                   action(R, _, _, A), action(R, _, _, B).

% o relation must be asymmetric and deterministic
incon(A, B) :- time(A), time(B), A = B.
incon(A, B) :- time(A), time(B), time(C), o(A, C), C != B.
incon(A, B) :- time(A), time(B), time(C), o(C, B), C != A.

% ordering conditions
incon(A, B) :- time(A), time(B), gt(B, A).
incon(A, B) :- time(A), time(C), gt(C, B), not reachable(C, A).

% check consistency of region traces (calculated dynamically)
incon(A, B) :- sameRegion(B, L), lastUpdate(R, A, L),
               not consistSequence(L, B).

% dynamically check that this addition to the region trace is
% consistent
consistSequence(A, B) :- action(R, SA1, SA2, A),
                         action(R, SB1, SB2, B),
                         #subList(SA2, SB1).

% helper predicates to calculate region trace
lastUpdate(R, A, L) :- reachable(L, A), action(R, _, _, L),
                       not updateBetween(L, A).
updateBetween(Ut, T) :- sameRegion(Ut, Other), Ut != Other,
                        reachable(Ut, Other), reachable(Other, T).

o(A, B):- time(A), time(B), reachable(start, A),
          not incon(A, B).

% transitive relation (reflexive for convenience)
reachable(A, B) :- time(A), time(B), A = B.
reachable(A, B) :- o(A, C), reachable(C, B).
```

```
% attempt to find a deadlocking example - failure indicates
% termination
:- reachable(start, end).

% sanity check user hypothesis
badAction :- Action(R, S1, S2, A), stateList(R, Ss),
             not #subList(S1, Ss).
badAction :- Action(R, S1, S2, A), stateList(R, Ss),
             not #subList(S2, Ss).

:- badAction.

% -----------------------------------------------------
% encoding the 2-lock deadlock example as a hypothesis
% should find two deadlocking counterexamples

region(lockA).
region(lockB).

time(a1).
time(a2).
time(a3).
time(a4).

time(b1).
time(b2).
time(b3).
time(b4).

gt(start, a1).
gt(a1, a2).
gt(a2, a3).
gt(a3, a4).
gt(a4, end).

gt(start, b1).
gt(b1, b2).
gt(b2, b3).
gt(b3, b4).
gt(b4, end).

stateList(lockA, [0, 1]).
stateList(lockB, [0, 1]).

action(lockA, [0], [1], a1).
action(lockB, [0], [1], a2).
action(lockB, [1], [0], a3).
action(lockA, [1], [0], a4).

action(lockB, [0], [1], b1).
action(lockA, [0], [1], b2).
action(lockA, [1], [0], b3).
action(lockB, [1], [0], b4).
```

B The model of TaDA

What follows is the original definition of TaDA's model as it appears in the technical report [19].

B.1 Operational Semantics

The operational semantics of our language are given in Fig. 32 and Fig. 33.

B.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 [20]) 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.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 32: Small-step operational semantics for threads, $\xrightarrow{a}_{\gamma}$. The parameter γ is fixed, and not shown.

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

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

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

Figure 33: Small-step operational semantics for thread pools, $\xrightarrow{a}_{\gamma}$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $p \in \mathsf{WPred} \overset{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathsf{World})$ 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:

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

(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 $\varphi \in \mathsf{AWorld}_{\mathcal{R}} \overset{\text{def}}{=} \mathsf{World} \times (\mathcal{R} \to (\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\})$ 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 $\mathcal{A} \in \mathsf{AContext} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{fin} \mathsf{AState} \rightharpoonup \mathcal{P}(\mathsf{AState})$ 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:

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

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

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

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

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

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

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

We call the stable predicates *views* (as in [20]) 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

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

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

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

[1] Here, we have avoided having region interpretations directly referring to region interpretations. Impredicative CAP [21] 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.

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:

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

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

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

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

This is lifted to predicates: $p{\downharpoonright_1} \stackrel{\text{def}}{=} \bigcup_{\varphi \in p} \varphi{\downharpoonright_1}$. The one-step collapse is iterated to give the multi-step collapse: $p{\downharpoonright_{n+1}} \stackrel{\text{def}}{=} (p{\downharpoonright_n}){\downharpoonright_1}$.

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

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

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

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

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

This operation is lifted to predicates in the usual manner.

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

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

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

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

B.2.1 Semantic Judgements

In the Views Framework [20], primitive atomic actions are abstracted to relations on views by means of an atomic satisfaction judgement. Here, we have an analogous judgement, but which is more complex as it expresses the role of an action in performing an abstractly-atomic operation. To express this role, we conceptually divide the view into a private and a public part. A thread is at liberty to do as it pleases with the private part (subject to preserving all stable frames). The public part, however, must be maintained invariant by the thread until it performs its abstract atomic action, at which point it updates the public part accordingly and thereafter loses access to it. The primitive atomic satisfaction judgement therefore incorporates five assertions: p_p, the precondition for the private part; p, the precondition for the public part; p'_p, the postcondition for the private part where the atomic update does not happen; q, the postcondition for the public part (when an atomic update does happen — otherwise p plays the role); and q_p, the postcondition for the private part where the atomic update does happen.

Definition 1 (Primitive Atomic Satisfaction Judgement)**.** The primitive atomic satisfaction judgement $\lambda; \mathcal{A} \vDash \langle p_p \mid p \rangle \; a \; \langle p'_p \mid - \rangle + \langle q_p \mid q \rangle$, 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:

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

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

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

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

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

where

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

similarly parametrised by $\mathbf{x} \in X$ and $\mathbf{y} \in Y$,

is defined to be the 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

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

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

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

- If $\mathbb{C} = \mathtt{skip}$ 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 $\lambda; \mathcal{A}; \Omega \vDash \mathbb{\forall} \mathbf{x} \in \mathbf{1}.\, \langle p | \mathsf{true} \rangle\ \mathbb{C}\ \mathbb{\exists} \mathbf{y} \in \mathbf{1}.\, \langle q | \mathsf{true} \rangle$.

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.

C Proofs involving the general region

We conduct the proof with a simple interpretation of the region as a single memory cell.

```
$(a, \{1\}, Q, \alpha) \vdash$
with $Q(1) = \{2\}$, otherwise $Q(n) = \{\}$
$\forall\!\!\!\forall y \in \mathbb{N}.$
$\langle \mathbf{GRegion}_a(\mathtt{x}, y) \rangle$
make atomic
  $a : y \in \mathbb{N} \rightsquigarrow Q(y) \vdash$
    $\{\exists y \in \mathbb{N}.\, \mathbf{GRegion}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
    b := 0;
    $\left\{\begin{array}{l} \exists y \in \mathbb{N}.\, \mathbf{GRegion}_a(\mathtt{x}, y) * \\ (a \Mapsto (1,2) \land \mathtt{b} = 1 \lor a \Mapsto \blacklozenge \land \mathtt{b} = 0) \end{array}\right\}$
    while (b = 0) {
    while3
        with $T = \{1\}$
          $\{\exists y \in \mathbb{N} / \{1\}.\, \mathbf{GRegion}_a(\mathtt{x}, y) * a \Mapsto \blacklozenge\}$
          update region
            $\forall\!\!\!\forall n \in \mathbb{N} / \{1\}.$
            $\langle \mathtt{x} \mapsto n \rangle$
            b := CAS(x, 1, 2);
            $\left\langle\begin{array}{l} (\mathtt{x} \mapsto 2 \land n = 1 \land \mathtt{b} = 1) \lor \\ (\mathtt{x} \mapsto n \land n \neq 1 \land \mathtt{b} = 0) \end{array}\right\rangle$ / $\langle(\mathtt{x} \mapsto 2 \land n = 1 \land \mathtt{b} = 1)\rangle$
          $\left\{\begin{array}{l} \exists y \in \mathbb{N}.\, \mathbf{GRegion}_a(\mathtt{x}, y) * \\ (a \Mapsto (1,2) \land \mathtt{b} = 1 \lor a \Mapsto \blacklozenge \land \mathtt{b} = 0) \end{array}\right\}$ / $\{(a \Mapsto (1,2) \land \mathtt{b} = 1\}$
    }
    $\{\exists y \in \{1\}, z \in Q(y).\, a \Mapsto (y, z) \land \mathtt{b} = 1\}$
$\langle \mathbf{GRegion}_a(\mathtt{x}, 2) \rangle$
```

Figure 34: Proof of an example general region operations, 1to2(x).

For a client proof, we can prove termination without caring particularly about a tight specification. Assuming that the value of the region is initially 0,

$$
\begin{array}{c}
\pi \vdash \\
\{\mathbf{GRegion}_a(0)\} \;/\; \{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} \\
\begin{array}{c||c}
\pi' \vdash & \pi'' \vdash \\
\{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} & \{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} \\
\alpha_1 \ \texttt{0to1(x);} & \texttt{1to2(x);} \ \beta_1 \\
\{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} & \{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} \\
\alpha_2 \ \texttt{2to1(x);} & \texttt{1to0(x);} \ \beta_2 \\
\{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\} & \{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\}
\end{array} \\
\{\exists n \in \mathbb{N}.\, \mathbf{GRegion}_a(n)\}
\end{array}
$$

Figure 35: Proof of an example general region client.

with actions are α (the prefix action), $\alpha_1, \alpha_2, \beta_1, \beta_2$, π will be

$$
\begin{array}{l}
\left\{ \begin{array}{l} (a, \{\between\}, \lambda x.\, \{0\}, \alpha), (a, \{0\}, Q_{0to1}, \alpha_1), (a, \{2\}, Q_{2to1}, \alpha_2), \\ (a, \{1\}, Q_{1to2}, \beta_1), (a, \{1\}, Q_{1to0}, \beta_2) \end{array} \right\}, \\
\alpha < \alpha_1 < \alpha_2, \ \alpha < \beta_1 < \beta_2
\end{array}
$$

There are 325 sequences containing the timestamps no more than once. We can reduce this by only considering the ones with consistent region traces. The sequences with consistent region traces are as follows:

$$
\begin{array}{l}
\alpha \\
\alpha \rightarrow \alpha_1 \\
\alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \\
\alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta_1 \\
\alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta_1 \rightarrow \beta_2
\end{array}
$$

only $\alpha \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \beta_1 \rightarrow \beta_2$ is a maximal sequence, and it contains all actions, so we have proven that the program terminates.

Now let us conduct the proof without assuming that the initial state of the region is 0. The only difference will be the precondition, and thus the state encoded in the prefix action. π will be

$$
\begin{array}{l}
\left\{ \begin{array}{l} (a, \{\between\}, \lambda x.\, \mathbb{N}, \alpha), (a, \{0\}, Q_{0to1}, \alpha_1), (a, \{2\}, Q_{2to1}, \alpha_2), \\ (a, \{1\}, Q_{1to2}, \beta_1), (a, \{1\}, Q_{1to0}, \beta_2) \end{array} \right\}, \\
\alpha < \alpha_1 < \alpha_2, \ \alpha < \beta_1 < \beta_2
\end{array}
$$

Now we have a counter example trace - $\alpha \rightarrow \beta_1$ is a maximal trace which respects all our restrictions but does not contain all actions. Therefore we cannot prove that the program terminates. Note that this is not the same as proving that it does not terminate.