Imperial College London
Department of Computing

Reasoning with Time and Data Abstractions

Pedro da Rocha Pinto

September 2016

Supervised by Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this thesis which is not my own work has been properly acknowledged.

Pedro da Rocha Pinto

Copyright

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

In this thesis, we address the problem of verifying the functional correctness of concurrent programs, with emphasis on fine-grained concurrent data structures. Reasoning about such programs is challenging since data can be concurrently accessed by multiple threads: the reasoning must account for the interference between threads, which is often subtle. To reason about interference, concurrent operations should either be at distinct times or on distinct data.

We present TaDA, a sound program logic for verifying clients and implementations that use abstract specifications that incorporate both *abstract atomicity*—the abstraction that operations take effect at a single, discrete instant in time—and *abstract disjointness*—the abstraction that operations act on distinct data resources. Our key contribution is the introduction of *atomic triples*, which offer an expressive approach for specifying program modules.

We also present Total-TaDA, a sound extension of TaDA with which we can verify total correctness of concurrent programs, i.e. that such programs both produce the correct result and terminate. With Total-TaDA, we can specify constraints on a thread's concurrent environment that are necessary to guarantee termination. This allows us to verify total correctness for nonblocking algorithms and express lock- and wait-freedom. More generally, the abstract specifications can express that one operation cannot impede the progress of another, a new non-blocking property that we call *non-impedance*.

Finally, we describe how to extend TaDA for proving abstract atomicity for data structures that make use of *helping*—where one thread is performing an abstract operation on behalf of another—and *speculation*—where an abstract operation is determined by future behaviour.

To the people who are still alive.

Contents

1 Introduction **19**

1.1 Contributions and Thesis Outline 19

2 Background **22**

2.1 Concurrent Modules 22

2.1.1 A Spin Counter Implementation 22

2.1.2 A Ticket Lock Client 23

2.2 Proof Outlines 24

2.3 Sequential Specification 26

2.4 Auxiliary State 27

2.5 Interference Abstraction 29

2.6 Resource Ownership 30

2.7 Atomicity 33

3 Informal Development **35**

3.1 Spin Counter 36

3.1.1 Atomic Specification 36

3.1.2 Implementation 38

3.2 Ticket Lock 40

3.2.1 CAP Specification 40

3.2.2 Implementation 42

4 TaDA Logic **46**

4.1 Programming Language 46

4.2 Operational Semantics 47

4.3 Assertion Language 53

4.4 Program Logic 54

4.5 Model 61

4.5.1 Semantic Assertions 66

4.5.2 Semantic Judgements 68

4.6 Soundness 71

5 Using TaDA **90**

5.1 Spin Lock 90

5.1.1 Atomic Specification 90

5.1.2 CAP Specification 92

5.1.3 Implementation 95

5.2 Multiple Compare-and-Set (MCAS) . . . 96
5.2.1 Atomic Specification . . . 96
5.2.2 Implementation . . . 97
5.2.3 Resource Transfer . . . 99
5.3 Deque . . . 101
5.3.1 Atomic specification . . . 101
5.3.2 Implementation . . . 102

6 Related Work 107

7 Reasoning about Termination 108
7.1 Motivating Examples . . . 110
7.1.1 Atomic Specification . . . 111
7.1.2 Clients . . . 112
7.1.3 Implementations . . . 115
7.2 Logic . . . 118
7.3 Case Study: Treiber's Stack . . . 120
7.3.1 Atomic Specification . . . 121
7.3.2 Implementation . . . 121
7.4 Semantics and Soundness . . . 123
7.5 Non-blocking Properties . . . 127
7.5.1 Lock-freedom . . . 127
7.5.2 Wait-freedom . . . 129
7.5.3 Non-impedance . . . 131
7.5.4 Related Work . . . 132

8 Extending the logic 133
8.1 Motivating Examples . . . 134
8.1.1 Ticket Lock . . . 134
8.1.2 Two-step Counter . . . 138
8.2 Case Study: Michael-Scott Queue . . . 141
8.2.1 Atomic Specification . . . 143
8.2.2 Implementation . . . 143

9 Conclusions 149
9.1 Future Work . . . 149
9.1.1 Tool Support . . . 149
9.1.2 Helping/Speculation . . . 149
9.1.3 Higher-order Support . . . 150
9.1.4 Weak Memory . . . 150
9.1.5 Liveness . . . 150
9.1.6 Fault Tolerance . . . 150

List of Figures

2.1 A counter module. 23
2.2 A ticket lock implemented using the counter module. 24
2.3 Reasoning about concurrent increments using auxiliary state. 28
2.4 Reasoning about concurrent increments using interference abstraction. 29
2.5 Ownership-based reasoning for concurrent increments. 33

3.1 Proof outline for the `read` operation. 40
3.2 Proof outline for the `incr` operation. 41
3.3 Proof outline for the `wkincr` operation. 42
3.4 A ticket lock implementation using the counter module. 42
3.5 Proof outline for the `release` operation. 44
3.6 Proof outline for the `acquire` operation. 45

4.1 Programming language proof rules of the TaDA logic. 56
4.2 Atomicity proof rules of the TaDA logic. 57
4.3 Auxiliary proof rules of the TaDA logic. 59

5.1 Derivation of the `release` specification. 94
5.2 Derivation of the `acquire` specification. 94
5.3 Spin lock operations. 94
5.4 Proof outline for the `acquire` operation. 96
5.5 The abstract specification for the MCAS module. 97
5.6 Multiple compare-and-set module operations. 98
5.7 Proof of the `dcas` implementation. 100
5.8 Snark deque operations. 103
5.9 Examples of a deque before and after performing `popLeft`, which uses `3cas` to updated pointers c, d and e. 104
5.10 Proof of the `popLeft` implementation. 106

7.1 Spin counter operations. 111
7.2 Proof of a sequential client of the counter. 113
7.3 Proof of a concurrent client of the counter. 115
7.4 Proof of total correctness of increment. 117
7.5 Backoff increment. 118
7.6 Proof of total correctness of backoff increment. 119
7.7 Stack operation specifications. 120
7.8 Treiber's stack operations. 122

7.9 Proof of total correctness of Treiber's stack `pop` operation. 124
7.10 Proof of total correctness of Treiber's stack `push` operation. 125

8.1 Axioms for proxy modalities. 137
8.2 Auxiliary predicate describing the shared resources of the ticket lock. 138
8.3 Proof of correctness of the `acquire` operation. 139
8.4 Proof of correctness of the `release` operation. 140
8.5 Auxiliary predicate describing the set of configurations for the two-step counter. . . . 140
8.6 Proof of correctness of the `readVal` operation. 142
8.7 Proof of correctness of the `incrVal` operation. 142
8.8 Michael-Scott Queue implementation. 144
8.9 Auxiliary predicates to describe the shared resources of the Michael-Scott queue. . . . 146
8.10 Proxies and witnesses for the Michael-Scott queue. 146
8.11 Proof of correctness of `dequeue` operation. 148

List of Theorems

2.1 Remark (Conventions) 25
2.2 Remark (On disjoint resources) 31

3.1 Remark (On the abstractly-typed parameters) 37

4.1 Definition (Variable Names) 46
4.2 Definition (Expressions) 46
4.3 Definition (Boolean Expressions) 46
4.4 Definition (Function Names) 47
4.5 Definition (Commands) 47
4.6 Definition (Function Bodies) 47
4.7 Definition (Programs) 47
4.8 Definition (Program Values) 47
4.9 Definition (Variable Store) 47
4.10 Definition (Semantics of Expressions) 48
4.11 Definition (Semantics of Boolean Expressions) 48
4.12 Definition (Extended Commands) 48
4.13 Definition (Continuations) 48
4.14 Definition (Threads) 49
4.15 Definition (Labels of Atomic Commands) 49
4.16 Definition (Function Environment) 49
4.17 Definition (Thread Operational Semantics) 49
4.18 Definition (Extend Commands Operational Semantics) 49
4.19 Definition (Thread Identifiers) 51
4.20 Definition (Thread Pools) 51
4.21 Definition (Heaps) 51
4.22 Definition (Program States) 51
4.23 Definition (Thread Pool Operational Semantics) 51
4.24 Definition (Addresses) 52
4.25 Definition (Interpretation of Atomic Commands) 52
4.26 Definition (Program Operational Semantics) 53
4.27 Definition (Logical Expressions) 53
4.28 Definition (Assertions) 53
4.29 Definition (Function Specification Context) 54
4.30 Definition (Atomic Judgements) 55
4.31 Definition (Proof System) 55
4.32 Definition (Program Variables Sets) 58

4.33 Definition (Modified Sets) 58
4.34 Definition (Variable Names) 61
4.35 Definition (Guards and Guard Algebras) 61
4.36 Definition (Abstract States and Transition Systems) 62
4.37 Definition (Abstract Region Types) 62
4.38 Definition (Abstract Predicates) 62
4.39 Definition (Levels) 62
4.1 Remark (On levels) 62
4.40 Definition (Region Identifiers) 62
4.41 Definition (Region Assignments) 62
4.42 Definition (Guard Assignments) 63
4.43 Definition (Region States) 63
4.44 Definition (Worlds) 63
4.45 Definition (World Predicates) 63
4.2 Remark (Intuitionistic Interpretation) 64
4.46 Definition (Worlds with Atomic Tracking) 64
4.47 Definition (Atomicity Context) 64
4.48 Definition (Rely Relation) 64
4.49 Definition (Stable Predicates) 65
4.50 Definition (Region Interpretation) 65
4.51 Definition (Abstract Predicate Interpretation) 65
4.52 Definition (Region Collapse) 65
4.53 Definition (Abstract Predicate Collapse) 66
4.3 Remark (On the interpretation of abstract predicates) 66
4.54 Definition (Reification) 66
4.55 Definition (Guarantee Relation) 66
4.4 Remark (On reification) 66
4.56 Definition (Variable Interpretations) 67
4.57 Definition (Logical Expressions Semantics) 67
4.58 Definition (Assertion Semantics) 67
4.59 Definition (Primitive Atomic Satisfaction Judgement) 68
4.60 Definition (View Shift) 68
4.61 Definition (Semantic Judgement) 68
4.62 Definition (Function Environment and Function Context Agreement) 70
4.63 Definition (Semantic Judgement with Function Context) 70
4.1 Theorem (Soundness of Commands) 71
4.2 Lemma (Skip Rule) 71
4.3 Lemma (Sequencing Rule) 71
4.4 Lemma (Loop Rule) 74
4.5 Lemma (Conditional Rule) 75
4.6 Lemma (Stack Frame) 75
4.7 Lemma (FunctionCall Rule) 77

4.8 Lemma (Fork Rule) . . . 78
4.9 Lemma (Allocation Rule) . . . 79
4.10 Lemma (Assignment Rule) . . . 79
4.11 Lemma (Lookup Rule) . . . 79
4.12 Lemma (Mutation Rule) . . . 80
4.13 Lemma (CompareAndSet Rule) . . . 80
4.14 Lemma (OpenRegion Rule) . . . 80
4.15 Lemma (UseAtomic Rule) . . . 82
4.16 Lemma (UpdateRegion Rule) . . . 83
4.17 Lemma (Drop Context) . . . 86
4.18 Lemma (MakeAtomic Rule) . . . 86
4.19 Lemma (Frame Rule) . . . 88
4.20 Lemma (Substitution Rule) . . . 88
4.21 Lemma (Consequence Rule) . . . 88
4.22 Lemma (AExists Rule) . . . 88
4.23 Lemma (AWeakening1 Rule) . . . 88
4.24 Lemma (AWeakening2 Rule) . . . 89
4.25 Lemma (AWeakening3 Rule) . . . 89

5.1 Remark (On alternative implementations) . . . 96

7.1 Definition (Semantic Judgement) . . . 123
7.1 Lemma (Loop Rule) . . . 126
7.2 Definition ((m, n)-bounded General Client) . . . 128
7.2 Theorem (Hoffmann *et al.* [30]) . . . 128
7.3 Theorem (Lock-freedom) . . . 128
7.4 Theorem (Wait-freedom) . . . 129
7.5 Theorem (Wait-free Operation) . . . 130

Acknowledgements

Foremost, I am grateful to my supervisor, Philippa Gardner, for her continued encouragement, enthusiasm and guidance during this research. I am equally thankful to Thomas Dinsdale-Young for his constant support, mentoring and patience. I am also indebted to Gian Ntzik for all the discussions throughout the years that have vastly improved my work.

I would like to thank my amazing collaborators — Kristoffer Just Andersen, Lars Birkedal, Mike Dodds, Julian Sutherland, Mark Wheelhouse, Shale Xiong — who have been wonderful to work with. As well as my colleagues at Imperial who make up the stimulating and friendly environment in which I have been privileged to work.

Finally, I would like to thank my family and my friends for the unconditional support and encouragement to pursue my interests and believing in me. In particular, my wonderful wife, Anita, whose love and support made this possible.

1 Introduction

The specification and verification of concurrent program modules is a difficult problem. When concurrent threads work with shared data, the resulting behaviour can be complex. Two abstractions provide useful simplifications: *atomicity*, stating that operations effectively act at distinct times; and *disjointness*, stating that operations effectively act on disjoint resources. While programmers routinely work with sophisticated combinations of these two abstractions, existing reasoning techniques, in contrast, tend to have limitations with respect to one or the other.

Two of the most widely used and influential techniques related to these two abstractions are *linearisability* [26], a correctness condition that specified that the operations of a concurrent module appear to behave atomically; and Concurrent Abstract Predicates (CAP) [12], an approach that offers support for reasoning about abstract disjoint resources and for specification of program modules.

With linearisability, each operation is given a sequential specification, and operations are asserted to behave atomically *with respect to each other*. Therefore, linearisability is a whole-module property: if we extend a module with an operation, we have to redo the linearisability proof to check that this new operation respects the atomicity of the others, and vice versa. Moreover, all operations are required to be atomic, so we can not specify non-atomic operations.

The CAP approach is based on concurrent separation logic, which provide concurrent reasoning based on the fundamental principle that resources are disjoint. Using CAP, it is possible to reason about shared resources as if they were disjoint, allowing for sophisticated concurrent implementations to be verified against simple specifications. However, in CAP, shared resources can only be accessed using *primitive atomic* operations, and operations provided by concurrent modules are rarely primitive atomic. Consequently, the abstract resources provided by a module are not easily shared and the nesting of modules is difficult.

Linearisability and CAP have complementary virtues and weaknesses. Linearisability gives strong, whole-module specifications based on abstract atomicity; CAP gives weaker, independent specifications based on abstract disjointness. Linearisability supports nested modules, but whole-module specifications make it difficult to extend modules; CAP supports the extension of modules, but the weak specifications make building up nested modules more difficult. Linearisability does not constrain the client, thus placing significant burden on the implementation; CAP constrains the client to use specific disjoint resources, enabling more flexibility in the implementation.

We propose a solution that combines the virtues of both approaches.

1.1 Contributions and Thesis Outline

Our main contribution is TaDA, a program logic for Time and Data Abstraction, which extends Concurrent Abstract Predicates [12] with rules for deriving and using atomic triples. We show how the logic supports vertical reasoning about modules by verifying implementations that are built on the

top of each other. We thus demonstrate that TaDA combines the benefits of abstract atomicity and abstract disjointness within a single program logic.

Moreover, we show how to extend TaDA to prove termination of non-blocking programs. We name this extension Total-TaDA and use it to prove several examples with increasing difficulty. We provide and prove relations between our specifications and lock-freedom and wait-freedom for modules and operations. We also present a new non-blocking property, called non-impedance, which enables us to describe that one operation cannot prevent the progress of another.

The last part of the thesis proposes a possible extension to the original logic that focusses on algorithms that exhibit helping and speculation with respect to an atomic specification. We propose a way to adapt TaDA to handle these algorithms and apply it to the ticket lock, which exhibits helping, a counter that requires speculation, and a Michael-Scott queue, which uses speculation.

The outline of the thesis is as follows:

- Chapter 2 provides the technical background that motivates the rest of the thesis. We start by introducing two concurrent modules — a counter and a ticket lock — and describing some of the challenges with reasoning about them. We give a brief introduction to Hoare reasoning, introduces the proof rules and how they can be used to reason about programs. We then survey a range of verification techniques for specifying concurrent modules, highlighting four key concepts in particular: auxiliary state, interference abstraction, resource ownership and atomicity.

- In chapter 3, we present an informal development of the TaDA logic using the counter and the ticket lock examples previously introduced. We show how our approach can be used to give abstract specifications, using atomic triples, which allow for modular reasoning and overcome the problems described throughout chapter 2.

- We formalise the programming language in chapter 4. We define the syntax and semantics of the TaDA logic and prove the soundness of the logic.

- In chapter 5, we apply the logic to a series of modules that are built on the top of each other. We show how the logic supports vertical reasoning about modules, by verifying an implementation of multiple-compare-and-set (MCAS) using the lock specification, and an implementation of a concurrent double-ended queue (deque) using the MCAS specification. We show that our approach is modular and that it can verify the operations of a module independently and provide abstract specifications.

- In chapter 6, we discuss related work and its advantages and disadvantages with respect to our work.

- In chapter 7, we present Total-TaDA, a program logic that extends TaDA with support for verifying the total correctness of non-blocking concurrent programs. We verify the total correctness of non-blocking algorithms, such as counters and Treiber's stack. We relate our specifications to the standard definitions of lock- and wait-freedom and propose a new non-blocking property we call non-impedance. We show that our extension is sound by adapting the semantics used for TaDA.

- In chapter 8, we propose a technique to reason about advanced concurrent algorithms that make use of helping and speculation. We apply the technique to the ticket lock, a novel two-step counter, and a Michael-Scott queue.
- Finally, in chapter 9 we summarise the contributions and raise several research questions.

Collaboration

Chapter 2 is heavily inspired by work done in collaboration with Dinsdale-Young and Gardner. Much of it was previously published in *Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper)* [9]. Chapter 4 and chapter 5 are based on joint work with Dinsdale-Young and Gardner and part of it was published in *TaDA: A Logic for Time and Data Abstraction* [8]. The termination work in chapter 7 was undertaken with Dinsdale-Young, Gardner and Sutherland, and published in *Modular Termination Verification for Non-blocking Concurrency* [10]. Further examples of modules verified using TaDA are available in *Abstract Specifications for Concurrent Maps* [64].

2 Background

This chapter introduces the terminology and notation used in the thesis and provides a systematic overview of a range of techniques for specifying and verifying concurrent modules that introduced the concepts that this work is built upon.

The specification and verification of a concurrent program are difficult problems. When concurrent threads work with shared data, the resulting behaviour can be complicated. In order to specify such modules, we require effective abstractions for capturing such complex behaviour. We will describe the main concepts related to specifying and verifying concurrent programs that have emerged over the last few decades. In particular, we restrict our exposition to those concepts on which this work is based: auxiliary state, interference abstraction, resource ownership and atomicity.

We highlight the challenges of specifying a concurrent module by using the counter module as an illustrative example in §2.1. We require a specification to be expressive enough for verifying the intended clients of the module. We also require the specification to be opaque, in that the implementation details do not leak into the specification.

Using the counter module, we present a range of historical verification techniques for concurrency:

- Owicki-Gries [49] introduces *auxiliary state* to abstract the internal state of threads (§2.4);
- rely/guarantee [34] introduces *interference abstraction* to abstract the interactions between different threads (§2.5);
- concurrent separation logic [47] introduces *resource ownership* to encode interference abstraction as auxiliary state (§2.6);
- linearisability [26] introduces *atomicity* as a way to abstract the effects of an operation (§2.7).

Later, we show in §3 how to combine these techniques to provide expressive ways for specifying concurrent modules.

2.1 Concurrent Modules

2.1.1 A Spin Counter Implementation

We consider the implementation of a concurrent counter shown in Figure 2.1. We make use of three *atomic* operations (operations that take effect at a single, discrete instant in time) that manipulate the heap. The operation $\mathtt{x} := [\mathbb{E}];$ reads the value of the heap position $\mathbb{E}$ and assigns it to the variable $\mathtt{x}$. The operation $[\mathbb{E}_1] := \mathbb{E}_2;$ stores the value $\mathbb{E}_2$ in the heap position $\mathbb{E}_1$. Finally, the compare-and-set (CAS) operation $\mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3);$ checks if the value at heap position $\mathbb{E}_1$ is equal to $\mathbb{E}_2$: if so, it replaces it with the value $\mathbb{E}_3$ and assigns 1 to $\mathtt{x}$; otherwise, $\mathtt{x}$ is assigned 0.

Here, the `read` operation returns the value of the counter. The `incr` operation increments the value of the counter and returns the old value, it is robust to to other threads incrementing the counter.

```
function makeCounter() {
  x := alloc(1);              // Allocate a single cell
  [x] := 0;                   // Initialise the value at x with 0
  return x;
}

function read(x) {
  v := [x];                   // Get value at address x
  return v;
}

function incr(x) {
  do {
    v := [x];                 // Get value at x
    b := CAS(x, v, v + 1);    // Compare value at x with v and
                              // set it to v + 1 if they are the same
  } while (b = 0);            // Retry if the CAS failed
  return v;
}

function wkIncr(x) {
  v := [x];                   // Get value at address x
  [x] := v + 1;               // Set value at x with v + 1
  return v;
}
```

Figure 2.1: A counter module.

This is done by using the compare-and-set to atomically update the value of the counter. The `wkincr` just increments the value of the counter assuming that no other thread is performing any increment concurrently.

A specification should describe how each operation affects the value of the counter. A specification should require the counter to exist as a precondition for each operation, since operations will not work unless the memory holding the counter is allocated. A specification should also describe the permitted interference from the context of concurrent operations. Intuitively, the `read` and `incr` operations are robust with respect to concurrent operations that change the value of the counter. By contrast, the potentially faster `wkIncr` requires that there is no concurrent operation that changes the value of the counter between the read and the increment of the value in order to behave as intended.

2.1.2 A Ticket Lock Client

Let us now consider the ticket lock [42] module that uses the counter module to provide synchronisation. The code for the lock is given in Figure 2.2. The lock uses two counters, the ticket counter `next` and the serving counter `owner`, which both initially have the value 0. A thread acquires the lock by calling the `acquire` operation. This operation increments the `next` counter to obtain a notional ticket. When the value of the `owner` counter agrees with this ticket, the thread has acquired the lock. It can then use whatever resources are protected by the lock, without interference from other threads. Control of these resources is relinquished by calling the `release` operation. This increments the `owner` counter, passing the lock on to the next waiting thread. Intuitively, the use of `incr` for the `acquire` operation

```
function makeLock() {
  next := makeCounter();    // Allocate ticket counter
  owner := makeCounter();   // Allocate serving counter
  x := alloc(2);            // Allocate two cells
  [x.next] := next;         // Initialise first cell with ticket counter address
  [x.owner] := owner;       // Initialise second cell with serving counter address
  return x;
}

function acquire(x) {
  next := [x.next];         // Get address of the ticket counter
  owner := [x.owner];       // Get address of the serving counter
  t := incr(next);          // Take a ticket
  do {
    v := read(owner);       // Get serving counter value
  } while (v ≠ t);          // Wait for serving value to match the ticket
}

function release(x) {
  owner := [x.owner];       // Get address of the serving counter
  wkIncr(owner);            // Increment the serving counter
}
```

where $\mathbb{E}.\mathtt{next} \stackrel{\text{def}}{=} \mathbb{E}$ $\qquad \mathbb{E}.\mathtt{owner} \stackrel{\text{def}}{=} \mathbb{E} + 1.$

Figure 2.2: A ticket lock implemented using the counter module.

is necessary, since it needs to be robust with respect to concurrent threads taking tickets. The use of `wkIncr` for the `release` operation is possible, since only the thread holding the lock should release it.

The challenge is to develop a concurrent specification of the counter module that would be strong enough to allow us to reason about the ticket lock. This mandates a precise description of how each operation affects the value of the counter, and a detailed account of interference to capture the fundamental distinction between `incr` and `wkIncr`.

The counter and its ticket lock client are realistic examples that illustrate some of the key difficulties in specifying and reasoning about concurrent modules. We are looking for a formal specification technique that would allow us to express such specifications and formally verify both their implementations as well as programs that use such specifications.

2.2 Proof Outlines

Before examining any techniques, we need to introduce some notation for formally proving properties of programs. Tony Hoare introduced Hoare triples [29] of the form $\left\{P\right\}\ \mathbb{C}\ \left\{Q\right\}$ with the main goal of specifying the behaviour of sequential programs. A triple associates the *predicates* P and Q with a program $\mathbb{C}$, where P is the precondition and Q is the postcondition of the program. The meaning of the triple is that if a program $\mathbb{C}$ is run in a state described by P then it will not fault, and if it terminates, the resulting state will be described by Q.

Hoare also introduced a formal system with a set of rules for rigorous reasoning about imperative programs, which included the following proof rules:

$$\textsc{Skip}\ \frac{}{\vdash \left\{P\right\} \texttt{skip;} \left\{P\right\}}$$

$$\textsc{Sequencing}\ \frac{\vdash \left\{P\right\} \mathbb{C}_1 \left\{R\right\} \qquad \vdash \left\{R\right\} \mathbb{C}_2 \left\{Q\right\}}{\vdash \left\{P\right\} \mathbb{C}_1\ \mathbb{C}_2 \left\{Q\right\}}$$

$$\textsc{Loop}\ \frac{\vdash \left\{P \wedge \mathbb{B}\right\} \mathbb{C} \left\{P\right\}}{\vdash \left\{P\right\} \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\} \left\{P \wedge \neg\mathbb{B}\right\}}$$

$$\textsc{Conditional}\ \frac{\vdash \left\{P \wedge \mathbb{B}\right\} \mathbb{C}_1 \left\{Q\right\} \qquad \vdash \left\{P \wedge \neg\mathbb{B}\right\} \mathbb{C}_2 \left\{Q\right\}}{\vdash \left\{P\right\} \texttt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \texttt{else}\ \{\mathbb{C}_2\} \left\{Q\right\}}$$

$$\textsc{Assignment}\ \frac{}{\vdash \left\{P[\mathbb{E}/\mathtt{x}]\right\} \mathtt{x} := \mathbb{E}; \left\{P\right\}}$$

$$\textsc{Consequence}\ \frac{P' \vdash P \qquad \vdash \left\{P\right\} \mathbb{C} \left\{Q\right\} \qquad Q \vdash Q'}{\vdash \left\{P'\right\} \mathbb{C} \left\{Q'\right\}}$$

The Skip rule asserts that the `skip;` does not change the state of the program.

The Sequencing rule allows us to compose programs that are executed sequentially.

The Loop rule states that there is a loop invariant, P, which is preserved by the loop body $\mathbb{C}$. After the loop is finished, the invariant P still holds, and moreover $\mathbb{B}$ must be caused the loop to end.

The Conditional rule states that if both branches of the conditional establish Q, then the postcondition of the whole satisfies Q.

The Assignment rule states that, after the assignment, any precondition that was true still holds for the variable. $P[\mathbb{E}/\mathtt{x}]$ denotes the assertion P in which each free occurrence of x has been replaced by the expression $\mathbb{E}$.

The Assignment rule states that, after the assignment, the variable `x` has value $\mathbb{E}$ where each occurrence of `x` has been replaced by v.

The Consequence rule allows to strengthen the precondition and weaken the postcondition.

Remark 2.1 (Conventions)**.** When doing proofs with Hoare triples, we generally do not mention the names of the rules, such as the rules that are used to reason about the constructs of the language. We implicitly use the rule of consequence and do not expand its steps. Moreover, when we explicitly apply rules we denote them wth a vertical bar, where the name of the rule is given sideways on the left of the bar.

In order to illustrate how this reasoning system works in practice, we will start by showing a program which manipulates the store. Given a program $\mathbb{C}$ with the following specification:

$$\vdash \left\{\mathtt{x} = v\right\} \mathbb{C} \left\{\mathtt{x} = 2\right\},$$

by the precondition, we know that , before the execution of the program, `x` contains some value v. During the execution, the program is allowed to change `x` to any value, as long as when the program terminates, the value of `x` is set to 2.

We can show that $\mathbb{C} \stackrel{\text{def}}{=} \texttt{if}\ (\mathtt{x} \neq 2)\ \{\mathtt{x} := 2;\}\ \texttt{else}\ \{\texttt{skip};\}$ satisfies the specification by constructing

the following derivation:

CONDITIONAL:

$\left\{ \mathtt{x} = v \right\}$

`if` $(\mathtt{x} \neq 2)$ {

  $\left\{ \mathtt{x} = v \land \mathtt{x} \neq 2 \right\}$

  CONSEQUENCE:

    $\left\{ \mathtt{x} = v \right\}$

    ASSIGNMENT:

      $\left\{ \mathtt{x} = v \right\}$

      $\mathtt{x} := 2;$

      $\left\{ \mathtt{x} = 2 \right\}$

    $\left\{ \mathtt{x} = 2 \right\}$

  $\left\{ \mathtt{x} = 2 \right\}$

} `else` {

  $\left\{ \mathtt{x} = v \land \mathtt{x} = 2 \right\}$

  CONSEQUENCE:

    $\left\{ \mathtt{x} = 2 \right\}$

    SKIP:

      $\left\{ \mathtt{x} = 2 \right\}$

      `skip;`

      $\left\{ \mathtt{x} = 2 \right\}$

    $\left\{ \mathtt{x} = 2 \right\}$

  $\left\{ \mathtt{x} = 2 \right\}$

}

$\left\{ \mathtt{x} = 2 \right\}$

The original proof system did not account for programs that manipulate the heap, such as the counter, or concurrency. We extend the first-order assertion language to heaps, using the syntax $x \mapsto v$ to assert that the heap, at address x, holds value v.

2.3 Sequential Specification

We can give a sequential specification for the counter module using Hoare triples:

$$\vdash \left\{ \mathsf{True} \right\} \; \mathtt{makeCounter}() \; \left\{ \mathsf{ret} \mapsto 0 \right\}$$
$$\vdash \left\{ \mathtt{x} \mapsto n \right\} \; \mathtt{read}(\mathtt{x}) \; \left\{ \mathtt{x} \mapsto n \land \mathsf{ret} = n \right\}$$
$$\vdash \left\{ \mathtt{x} \mapsto n \right\} \; \mathtt{incr}(\mathtt{x}) \; \left\{ \mathtt{x} \mapsto n + 1 \land \mathsf{ret} = n \right\}$$
$$\vdash \left\{ \mathtt{x} \mapsto n \right\} \; \mathtt{wkIncr}(\mathtt{x}) \; \left\{ \mathtt{x} \mapsto n + 1 \land \mathsf{ret} = n \right\}$$

With standard Hoare logic, we can use this specification to verify sequential clients that call the counter operations. However, this specification gives no information about the behaviour of the operations in a concurrent setting.

2.4 Auxiliary State

Owicki and Gries [49] developed the first tractable proof technique for concurrent programs, identifying the importance of reasoning about *interference* between threads and of using *auxiliary state*. With the Owicki-Gries method, each thread is given a sequential proof. When the threads are composed, we must check that they do not interfere with each other's proofs. This is achieved by extending standard Hoare logic with the Owicki-Gries rule for parallel composition:

$$\frac{\vdash_{\mathsf{OG}} \left\{P_1\right\} \mathbb{C}_1 \left\{Q_1\right\} \qquad \vdash_{\mathsf{OG}} \left\{P_2\right\} \mathbb{C}_2 \left\{Q_2\right\} \qquad \mathsf{non\text{-}interference}}{\vdash_{\mathsf{OG}} \left\{P_1 \wedge P_2\right\} \mathbb{C}_1 \parallel \mathbb{C}_2 \left\{Q_1 \wedge Q_2\right\}} \textsc{OG-Parallel}$$

The non-interference side condition constrains the proof derivations for $\mathbb{C}_1$ and $\mathbb{C}_2$. It requires that every intermediate assertion between atomic actions in the proof of $\mathbb{C}_1$ must be preserved by every atomic action in the proof of $\mathbb{C}_2$, and vice-versa.

An abstract specification for the counter needs to be robust with respect to the non-interference condition. However, in general, the condition will vary depending on the concurrent context. Let us assume that the client may invoke any of the counter operations concurrently, but will not directly interact with the state of the counter. That is, we will only consider interference caused by the counter operations themselves. To this end, we can use an *invariant* — an assertion that is preserved by each atomic action in the module. For the counter, the invariant $\exists n.\, \mathtt{x} \mapsto n$ asserts that the counter at $\mathtt{x}$ is allocated and has some value.

We can give the following specification for the counter module:

$$\begin{aligned}
\vdash_{\mathsf{OG}} &\left\{\mathsf{True}\right\} \mathtt{makeCounter}() \left\{\exists n.\, \mathsf{ret} \mapsto n\right\} \\
\vdash_{\mathsf{OG}} &\left\{\exists n.\, \mathtt{x} \mapsto n\right\} \mathtt{read}(\mathtt{x}) \left\{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\right\} \\
\vdash_{\mathsf{OG}} &\left\{\exists n.\, \mathtt{x} \mapsto n\right\} \mathtt{incr}(\mathtt{x}) \left\{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\right\} \\
\vdash_{\mathsf{OG}} &\left\{\exists n.\, \mathtt{x} \mapsto n\right\} \mathtt{wkIncr}(\mathtt{x}) \left\{\exists n, m.\, \mathtt{x} \mapsto n \wedge \mathsf{ret} = m\right\}
\end{aligned}$$

However, these specifications are too weak to specify clients such as the ticket lock. They lose all information about the value of the counter, and give no information about how the operations change this value. In fact, the `read` operation could change the value of the counter and still satisfy the specification! Unfortunately, assertions that describe the precise value of the counter are not invariant.

The Owicki-Gries method is able to provide stronger specifications by using *auxiliary state*, which records extra information about the execution history via auxiliary variables. The code is instrumented with *auxiliary code*, which updates the auxiliary variables. Since the auxiliary code only updates auxiliary variables, it has no effect on the program behaviour, and so can be erased — it is not required when the program is run.

By way of example, consider two threads that both increment a counter, as in Figure 2.3. The auxiliary variables `y` and `z`, with initial values 0, are used to record the contribution (*i.e.* the number of increments) of each thread. For each thread, the code of the `incr` operation is instrumented with code that updates the auxiliary variables when the `CAS` operations succeed. The auxiliary variables

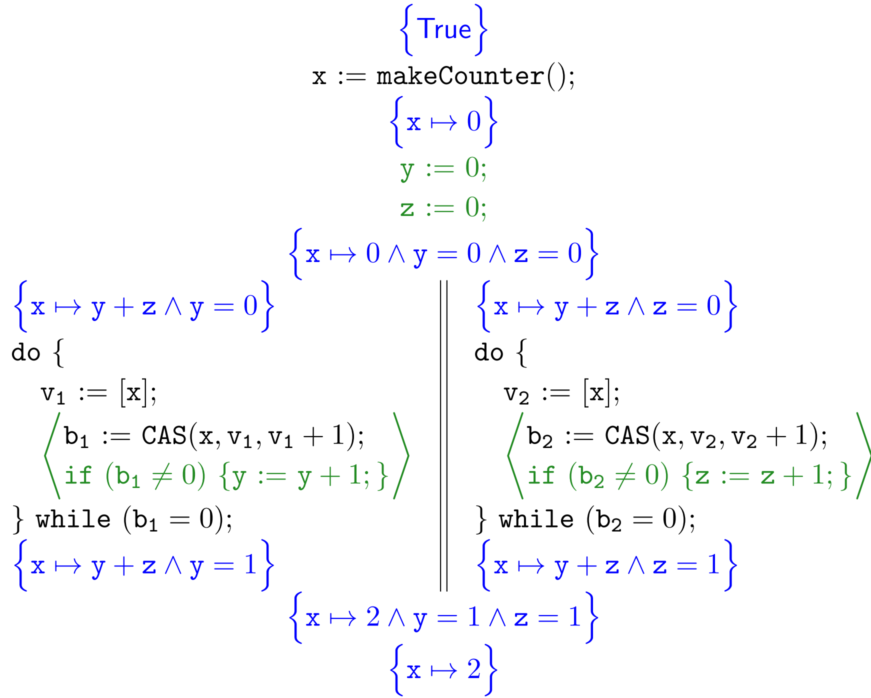


Figure 2.3: Reasoning about concurrent increments using auxiliary state.

must be updated at the same instant as the counter, so that the counter always holds the sum of the two contributions — our invariant. This is expressed in the angle brackets, $\langle _ \rangle$, which indicate that the `CAS` and auxiliary code should be executed in a single atomic step.

The resulting specification of the two-increment program is strong, with precise information about the initial and final value of the counter. However, it comes at the price of modularity. Firstly, each use of the `incr` operation requires the underlying implementation to be extended with auxiliary code to increment the appropriate auxiliary variable. A modular proof would not modify the module code for each use by the client. Secondly, the `incr` operations require different specifications depending on the client's use: in our example, the assertion $\mathtt{x} \mapsto \mathtt{y} + \mathtt{z}$ uses auxiliary variables `y` and `z`; with three threads, the specification requires three auxiliary variables. A modular proof would give a specification for the module that captures all use cases. Thirdly and more subtly, the Owicki-Gries method requires the global non-interference condition. To meet this, we made the implicit assumption that the client only interacts with the state of the counter through the counter operations. A modular proof would be explicit about such assumptions about the behaviour of the client.

Thesis

The concept of *auxiliary state*, introduced in the Owicki-Gries method, is important in specifying concurrent modules. Auxiliary state abstracts the internal state of threads. It is more convenient to reason using auxiliary variables than to consider the program counter and local variables of each thread in describing invariants. This abstraction is a step towards compositional reasoning. As we shall see, various subsequent approaches have taken a more modular approach to auxiliary state than that provided by auxiliary variables in the Owicki-Gries method.

$$
\begin{array}{c}
\left\{\mathsf{True}\right\} \\
\mathtt{x} := \mathtt{makeCounter}(); \\
\left\{\mathtt{x} \mapsto 0\right\} \\
\text{// Weaken assertion} \\
\left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 0\right\} \\
\begin{array}{l||l}
\left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 0\right\} & \left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 0\right\} \\
\mathtt{incr}(\mathtt{x}); & \mathtt{incr}(\mathtt{x}); \\
\left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 1\right\} & \left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 1\right\}
\end{array} \\
\left\{\exists n.\, \mathtt{x} \mapsto n \land n \geq 1\right\}
\end{array}
$$

Figure 2.4: Reasoning about concurrent increments using interference abstraction.

2.5 Interference Abstraction

Jones [34] introduced *interference abstraction*, providing the rely/guarantee method as a way to improve the compositionality of the Owicki-Gries approach. To avoid the global non-interference condition, specifications explicitly constrain the interference from the concurrent context, and describe the interference that a thread may cause. To this end, each specification incorporates two relations—the *rely* and *guarantee* relations—that abstract the interference between threads. The rely relation abstracts the actions of other threads; each assertion in the derivation must be *stable* under all of these actions. The guarantee relation abstracts the actions in the derivation; each atomic update by the thread must be described by the guarantee.

Rely/guarantee specifications have the form $R, G \vdash_{\mathsf{RG}} \left\{P\right\}\ \mathbb{C}\ \left\{Q\right\}$, where R and G are the rely and guarantee relations respectively. We denote the elements of the rely and guarantee relations in terms of actions $P \rightsquigarrow Q$ that describe the changes performed. When composing concurrent threads, the guarantee of each thread must be included in the rely of the other. The parallel composition rule is therefore adapted to:

$$
\textsc{RG-Parallel}\quad
\frac{R \cup G_2, G_1 \vdash_{\mathsf{RG}} \left\{P_1\right\}\ \mathbb{C}_1\ \left\{Q_1\right\} \qquad R \cup G_1, G_2 \vdash_{\mathsf{RG}} \left\{P_2\right\}\ \mathbb{C}_2\ \left\{Q_2\right\}}{R, G_1 \cup G_2 \vdash_{\mathsf{RG}} \left\{P_1 \land P_2\right\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \left\{Q_1 \land Q_2\right\}}
$$

The rely/guarantee specifications for the `read` and `incr` operations are:

$$
A, \emptyset \vdash_{\mathsf{RG}} \left\{\exists n.\, \mathtt{x} \mapsto n\right\}\ \mathtt{read}(\mathtt{x})\ \left\{\exists n.\, \mathtt{x} \mapsto n \land \mathsf{ret} \leq n\right\}
$$
$$
A, A \vdash_{\mathsf{RG}} \left\{\exists n.\, \mathtt{x} \mapsto n\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\exists n.\, \mathtt{x} \mapsto n + 1 \land \mathsf{ret} \leq n\right\}
$$

where $A = \{\mathtt{x} \mapsto n \rightsquigarrow \mathtt{x} \mapsto n + 1 \mid n \in \mathbb{N}\}$. The `read` specification has an empty guarantee relation indicating that nothing is changed by the read. It has the rely relation A, stating that other threads can only increment the counter, and that they can do so as many times as they like. The `incr` relation has the same rely relation. Its guarantee relation is also A, stating that the increment can increase the value of the counter. The guarantee must be defined for all n, as the context can change the counter value. This means that we cannot express that the `incr` operation only does a single increment.

The rely/guarantee specification for the `wkIncr` operation is subtle. Recall that, intuitively, the `wkIncr` operation is intended to be used when no other threads are concurrently updating the counter. As a first try, we can give a simple specification with a rely condition that enforces this constraint:

$$\emptyset, G \vdash_{\mathsf{RG}} \left\{ \mathtt{x} \mapsto n \right\} \ \mathtt{wkIncr(x)} \ \left\{ \mathtt{x} \mapsto n+1 \land \mathsf{ret} = n \right\}$$

where $G \triangleq \{\mathtt{x} \mapsto n \rightsquigarrow \mathtt{x} \mapsto n+1\}$. The rely relation is empty, so this specification cannot be used in a context where concurrent updates may occur. This means that the guarantee relation can be very precise, consisting of a single action. Effectively, the increment will appear as a single atomic operation.

Although this specification captures some of the intended behaviour of `wkIncr`, it is insufficient to reason about the ticket lock. With the ticket lock, it is possible for two invocations of the `wkIncr` operation to be executing concurrently. Only one thread can call `release` at any one time, because only one thread can have the lock. However, suppose one thread calling `release` has executed the body of `wkIncr`. Then, a second thread may correctly conclude that it now has the lock and release it, before the call of the first thread has returned. This results in a concurrent invocation of `wkIncr`. By ruling out all concurrent updates to the counter with an empty rely relation, the above specification does not allow this concurrent behaviour.

By changing the rely, we can allow such concurrent updates, but that would weaken the specification:

$$R, G \vdash_{\mathsf{RG}} \left\{ \mathtt{x} \mapsto n \right\} \ \mathtt{wkIncr(x)} \ \left\{ \exists n' \geq n+1.\, \mathtt{x} \mapsto n' \land \mathsf{ret} = n \right\}$$

where $R = \{\mathtt{x} \mapsto m \rightsquigarrow \mathtt{x} \mapsto m+1 \mid m > n\}$ and G is as before. Notice that the rely states that concurrent increments can only happen when the value of the counter is above n. Also notice that, in weakening the rely, we must weaken the postcondition to make it stable.

In summary, this specification is again too weak to reason about the ticket lock. It is possible to instrument the code with auxiliary variables, as with the Owicki-Gries method, but this would again lead to a loss of modularity.

Thesis

The concept of *interference abstraction*, introduced in the rely/guarantee method, is important in specifying concurrent modules. By abstracting the interactions between different threads, specifications can express constraints on their concurrent contexts. This abstraction leads to more compositional reasoning: since the interference is part of the specification, we do not need to examine proofs in order to justify parallel composition. While they may specify it differently, some form of interference abstraction is generally present in subsequent concurrency verification approaches.

2.6 Resource Ownership

In the Owicki-Gries and rely/guarantee approaches, auxiliary variables provide a mechanism for reasoning about which threads can do what and when. For instance, auxiliary variables can be used to reason about the contribution of individual threads to the counter, as we demonstrated in §2.4, or that one thread can increment a counter after another. O'Hearn introduced a style of reasoning based on

resource ownership, developing concurrent separation logic [47] which provides an alternative, more modular approach to such reasoning.

Concurrent separation logic is a Hoare logic, with assertions describing data (such as heap cells or counter objects) treated as resources. Each operation acts on specific resources, with the precondition conferring ownership of the resources it represents. When threads operate on disjoint resources, they do not interfere and so their effects can be simply combined. This principle is embodied in the disjoint parallel composition rule:

$$\textsc{Parallel}\quad \frac{\left\{P_1\right\} \mathbb{C}_1 \left\{Q_1\right\} \qquad \left\{P_2\right\} \mathbb{C}_2 \left\{Q_2\right\}}{\left\{P_1 * P_2\right\} \mathbb{C}_1 \parallel \mathbb{C}_2 \left\{Q_1 * Q_2\right\}}$$

where $P_1 * P_2$ describes the disjoint combination of the resources of P_1 and P_2.

Remark 2.2 (On disjoint resources)**.** The separating conjunction $*$ is based on separation logic [31, 51], which concurrent separation logic builds upon. Its key idea was to treat heap memory as a resource, which can be subdivided into disjoint resources. Heap operations, such as updating the value of a heap cell, require parts of the heap for their execution. These operations are *local* with respect to the specific resource they operate, as they do no affect other parts of the heap. This idea is expressed in the following rule:

$$\textsc{Frame}\quad \frac{\left\{P\right\} \mathbb{C} \left\{Q\right\}}{\left\{P * R\right\} \mathbb{C} \left\{Q * R\right\}}$$

The frame rule of separation logic allows us to reason about programs in a local way, i.e. we can focus our reasoning on the resource that the program manipulates, and any additional resource, which would not be affected by the program, can be added on. This is possible because $*$ enforces disjointness. □

We can think of ownership as embodying specialised notions of auxiliary state and interference abstraction. Ownership is a form of auxiliary state and an abstraction that we use for reasoning: the program does not explicitly record which threads own what resources. Ownership implements a simple interference abstraction: threads may update the resources that they own, and disjointness of ownership enforces that they cannot interfere with the resources of other threads.

In the original concurrent separation logic, it was only possible to reason about shared resource that had been transferred between threads through synchronisation. In [47], conditional critical regions provide the synchronisation mechanism. Subsequent approaches [63, 18, 15] added support for reasoning about fine-grained concurrency by incorporating various styles of rely/guarantee reasoning over shared resources. Building on this work, the concurrent abstract predicates (CAP) [12] approach introduces abstractions over these shared resources that may be split, effectively allowing concurrent manipulation at the abstract level. Let us illustrate this on the example of a counter.

We use the abstract predicate $\mathsf{Counter}(x, n)$ to denote the existence of a counter at memory location x with the value n. In the counter implementation we are reasoning with, the abstract predicate is instantiated by the implementation as $x \mapsto n$.

Treating the abstract predicate $\mathsf{Counter}(\mathtt{x}, n)$ as a resource, we could use the original sequential specification as a concurrent one. However, for multiple threads to use the counter, they would have to transfer the resource between each other using some form of synchronisation. Such a specification

effectively enforces sequential access to the counter. This is because the client has no mechanism for dividing the resource: in particular,

$$\mathsf{Counter}(\mathtt{x}, n) \implies \mathsf{Counter}(\mathtt{x}, n) * \mathsf{Counter}(\mathtt{x}, n)$$

does not hold.

Following Boyland [4], Bornat *et al.* [2] introduced *permission accounting* to separation logic. This allows shared resources to be divided by associating with them a fraction in the interval $(0, 1]$. Shared resources may be subdivided by splitting this fraction. For instance, we may associate fractions with our counter resource and declare the logical axiom:

$$\mathsf{Counter}(\mathtt{x}, n, \pi_1 + \pi_2) \iff \mathsf{Counter}(\mathtt{x}, n, \pi_1) * \mathsf{Counter}(\mathtt{x}, n, \pi_2)$$

for $\pi_1 + \pi_2 \leq 1$. We can now modify our counter specification to give concurrent read access:

$$\begin{array}{c}
\left\{\mathsf{Counter}(\mathtt{x}, n, \pi)\right\}\ \mathtt{read(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n, \pi) * \mathsf{ret} = n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, 1)\right\}\ \mathtt{incr(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n + 1, 1) * \mathsf{ret} = n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, 1)\right\}\ \mathtt{wkIncr(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n + 1, 1) * \mathsf{ret} = n\right\}
\end{array}$$

Notice that we require full permission (the 1) in order to perform either increment operation. This means that only concurrent reads are permitted; concurrent updates must be synchronised with all other concurrent accesses (both increments and reads). If only partial permission were necessary, then the specification for `read` would be incorrect, since it could no longer guarantee that the value being read matched the resource it had.

It is possible to specify concurrent increments by changing how we interpret the counter predicate $\mathsf{Counter}(\mathtt{x}, n, \pi)$. Now, the resource $\mathsf{Counter}(\mathtt{x}, n, \pi)$ no longer asserts that the value of the counter is n, except if $\pi = 1$. Instead, it asserts that the thread is contributing n to the value of the counter; other threads may also have contributions. We can split this counter resource by declaring the logical axiom:

$$\mathsf{Counter}(\mathtt{x}, n_1 + n_2, \pi_1 + \pi_2) \iff \mathsf{Counter}(\mathtt{x}, n_1, \pi_1) * \mathsf{Counter}(\mathtt{x}, n_2, \pi_2)$$

for $n_1, n_2 \in \mathbb{N}$ and $\pi_1, \pi_2 \in (0, 1]$. We then specify our counter operations as:

$$\begin{array}{c}
\left\{\mathsf{Counter}(\mathtt{x}, n, \pi)\right\}\ \mathtt{read(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n, \pi) * \mathsf{ret} \geq n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, 1)\right\}\ \mathtt{read(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n, 1) * \mathsf{ret} = n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, \pi)\right\}\ \mathtt{incr(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n + 1, \pi) * \mathsf{ret} \geq n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, 1)\right\}\ \mathtt{incr(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n + 1, 1) * \mathsf{ret} = n\right\} \\
\left\{\mathsf{Counter}(\mathtt{x}, n, 1)\right\}\ \mathtt{wkIncr(x)}\ \left\{\mathsf{Counter}(\mathtt{x}, n + 1, 1) * \mathsf{ret} = n\right\}
\end{array}$$

At last, we have a specification that allows concurrent reads and increments.

Figure 2.5 shows how it can be used to verify the example of two concurrent increments. Whereas in Figure 2.3 each thread was instrumented with different auxiliary code, here the code has not been

changed. Rather than each thread having an auxiliary variable to record its contribution to the counter, the contribution is recorded in auxiliary resources that are owned by the thread and encapsulated in the $\mathsf{Counter}(\mathtt{x}, n, \pi)$ predicate. This idea of subjective auxiliary state is at the core of subjective concurrent separation logic (SCSL) [37] (and the subsequent FCSL [44, 52]).

This specification still has some weaknesses. The `wkIncr` operation must still be synchronised with the other operations. Also, sequenced reads will never see decreasing values of the counter (since the contribution is not changed and only provides the lower bound). It is possible to describe a more elaborate permission system that allows `wkIncr` in the presence of reads, and to extend the predicate to record the last known value as a lower bound for reads. This would give us a more useful, if somewhat cumbersome, specification. However, it would still not handle the ticket lock.

While a ticket lock has been verified using CAP [12], the proof depends on the atomicity of the underlying counter operations in order to synchronise access to shared resources. The proof does not work with any of our abstract specifications, since they simply do not embody the necessary atomicity.

Thesis

The concept of *resource ownership*, developed by concurrent separation logic and its successors, is important in specifying concurrent modules. The idiom of ownership can be seen as a form of auxiliary state, which critically embodies a notion of disjointness and interference abstraction. Various approaches have explored the power of ownership for reasoning about concurrency [12, 37, 59, 44, 55, 54, 52, 35]. While it is an effective concept, and can be used to give elegant specifications, something more is required to provide the strong specifications we are seeking.

2.7 Atomicity

Atomicity is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of such operations is equivalent to a sequential interleaving of the operations, and a client can use such operations as if they were simple atomic operations. A well-known correctness condition for atomicity, which identifies when the operations of a concurrent module *appear* to behave atomically, is *linearisability* [26].

Using the linearisability approach, each operation is given a sequential specification. The operations are then proved to behave atomically *with respect to each other*. One way of seeing this is that there is an instant during the invocation of each operation at which that operation appears to take effect. This

$$\left\{\mathsf{Counter}(\mathtt{x}, 0, 1)\right\}$$
$$\left\{\mathsf{Counter}(\mathtt{x}, 0, 0.5) * \mathsf{Counter}(\mathtt{x}, 0, 0.5)\right\}$$
$$\begin{array}{l||l} \left\{\mathsf{Counter}(\mathtt{x}, 0, 0.5)\right\} & \left\{\mathsf{Counter}(\mathtt{x}, 0, 0.5)\right\} \\ \mathtt{incr}(\mathtt{x}) & \mathtt{incr}(\mathtt{x}) \\ \left\{\mathsf{Counter}(\mathtt{x}, 1, 0.5)\right\} & \left\{\mathsf{Counter}(\mathtt{x}, 1, 0.5)\right\} \end{array}$$
$$\left\{\mathsf{Counter}(\mathtt{x}, 1, 0.5) * \mathsf{Counter}(\mathtt{x}, 1, 0.5)\right\}$$
$$\left\{\mathsf{Counter}(\mathtt{x}, 2, 1)\right\}$$

Figure 2.5: Ownership-based reasoning for concurrent increments.

instant is referred to as the *linearisation point*. With linearisability, the interference of every operation is tolerated at all times by any of the other operations. Consequently, the interference abstraction is deemed to be the module boundary.

Given our sequential specification for the counter in §2.3, is our implementation linearisable? If we only consider the `read` and `incr` operations, then yes, it is. However, the addition of the `wkIncr` operation breaks linearisability. The problem with `wkIncr` is that, for instance, two concurrent calls can result in the counter only being incremented once. This is not consistent with atomic behaviour.

The essence of the problem is that we only envisage calling `wkIncr` in a concurrent context where there are no other increments. In such a case, it would appear to behave atomically. By itself, the sequential specification cannot express this constraint. We need an interference abstraction that constrains the concurrent context.

Alternatively, we could adapt the sequential specification to abstract the changes to the counter value performed by `wkIncr`, allowing it to be used concurrently with other increments, as follows:

$$\vdash \left\{\mathsf{True}\right\}\ \mathtt{makeCounter}()\ \left\{\mathsf{ret} \mapsto 0\right\}$$

$$\vdash \left\{\mathtt{x} \mapsto n\right\}\ \mathtt{read}(\mathtt{x})\ \left\{\mathtt{x} \mapsto n \land \mathsf{ret} = n\right\}$$

$$\vdash \left\{\mathtt{x} \mapsto n\right\}\ \mathtt{incr}(\mathtt{x})\ \left\{\mathtt{x} \mapsto n + 1 \land \mathsf{ret} = n\right\}$$

$$\vdash \left\{\mathtt{x} \mapsto n\right\}\ \mathtt{wkIncr}(\mathtt{x})\ \left\{\exists n.\, \mathtt{x} \mapsto n \land \mathsf{ret} \leq n\right\}$$

The new specification allows us to see the operations as if they were behaving atomically with respect to each other. However, the specification for `wkIncr` loses information about the state of the counter.

Linearisability is related to the notion of contextual refinement. With contextual refinement, the behaviour of program code is described by (more abstract) specification code.[1] Contextual refinement asserts that the specification code can be replaced by the program code in any context, without introducing new observable behaviours; we say that the program code contextually refines the specification code. Filipović *et al.* [19] have shown that, under certain assumptions about a programming language, linearisability implies contextual refinement for that language. For a linearisable module, each operation contextually refines the operation itself executed atomically. For instance, `incr(x)` contextually refines $\langle$`incr(x)`$\rangle$.

CaReSL [59] is a logic for proving contextual refinement of concurrent programs. CaReSL makes use of auxiliary state, interference abstraction and ownership in its proof technique. However, these concepts are not exposed in their specifications. This means that it is not obvious what a suitable specification of `wkIncr` in CaReSL should be.

Thesis

The concept of *atomicity*, put forward by linearisability, is important in specifying concurrent modules. Atomicity can be seen as a form of interference abstraction: it effectively guarantees that the only observable interference from an operation will occur at a single instant in its execution. This is a powerful abstraction, since a client need not consider intermediate states of an atomic operation (which, for non-atomic operations, might violate invariants) but only the overall transformation it performs.

[1] In general, the specification code need not be directly executable, although it does have a semantics.

3 Informal Development

In chapter 2, we have considered a number of proof methods for verifying concurrent programs: Owicki-Gries, rely/guarantee, concurrent separation logics and linearisability. For each method, we have identified a particularly valuable contribution towards specifying concurrent modules.

In this chapter we combine this contributions to produce specifications that are both expressive and modular. We propose a solution that combines the virtues of each of the approaches. Specifically, we introduce a new *atomic triple* judgement for specifying abstract atomicity in a program logic. The simplest form of atomic triple judgement is

$$\vdash \left\langle P \right\rangle \mathbb{C} \left\langle Q \right\rangle$$

where P and Q are assertions in the style of separation logic, and $\mathbb{C}$ is a program. This judgement is read as "$\mathbb{C}$ atomically updates P to Q". The program may actually take multiple steps, but each step before the atomic update from P to Q must preserve the assertion P. Before the atomic update occurs, the concurrent environment may also update the state, provided that the assertion P is preserved. As soon as the atomic update has happened, the environment can do whatever it likes; it is not constrained to preserve Q. At the same time, the program $\mathbb{C}$ may no longer have access to the resources in Q.

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

Atomic triples are our key contribution, as they allow us to overcome the limitations of the linearisability and CAP approaches. Atomic triples can be used to access shared resources concurrently, rather than relying on primitive atomic operations to do so. This makes it easier to build modules on top of each other. Atomic triples specify operations with respect to an abstraction, so they can be proved independently. This makes it possible to extend modules at a later date, and mix together both atomic and non-atomic operations, as well as operations working at different levels of abstraction. Atomic triples can specify clear constraints on how a client can use them. For instance, they can enforce that the unlock operation on a lock should not be called by two threads at the same time (§5.1). Furthermore, atomic triples can specify the transfer of resources between a client and a module. For instance, they can specify an operation that non-atomically stores the result of an atomic read into a buffer provided by a client (§5.2.3).

In order to illustrate our notion of atomicity for a program, we start by showing two simple programs

which manipulate the heap. First, let us consider a program $\mathbb{C}_1$ with the following specification:

$$\vdash \left\langle x \mapsto 0 \vee x \mapsto 1 \right\rangle \mathbb{C}_1 \left\langle x \mapsto 2 \right\rangle$$

By the precondition, we know that x can be only 0 or 1. The program is allowed to change x to 0 or 1 without this change being considered as part of the atomic update. This is because the abstraction boundary defined by the precondition allows for such interference to happen. The environment is required to maintain that condition until $\mathbb{C}_1$ assigns 2 to x. When $\mathbb{C}_1$ assigns 2 to x, that update is considered to be atomic and at that point, $\mathbb{C}_1$ no longer has access to the memory position x, as the environment is allowed to do anything with it.

Consider now a variation of the previous specification, featuring the TaDA-specific quantifier $\mathbb{V}$:

$$\vdash \mathbb{V}v \in \{0, 1\}. \left\langle x \mapsto v \right\rangle \mathbb{C}_2 \left\langle x \mapsto 2 \right\rangle$$

As in the previous example, the precondition here can either be $x \mapsto 0$ or $x \mapsto 1$, but the abstraction boundary introduced by the $\mathbb{V}$ quantifier is more precise. The environment is still allowed to change (multiple times) the value of x to any of those two values until the atomic update of $\mathbb{C}_2$ occurs. On the other hand, $\mathbb{C}_2$ is no longer allowed to change x, unless to perform the atomic update. The environment is the only one allowed to change the state for any v. At some point, $\mathbb{C}_2$ will perform its atomic update, writing 2 to x and at that point, as in the previous example, it will no longer have access to the memory position and the environment will be allowed to do anything with it.

We introduce TaDA by showing how to specify and verify the counter and the ticket lock modules shown in chapter 2.

3.1 Spin Counter

We consider the counter implementation from §2.1.1 and give for it a strong specification that allows us to prove the ticket lock. Using TaDA, we prove that the implementation satisfies this specification.

3.1.1 Atomic Specification

The counter operations are specified in terms of an abstract predicate [50] that represents the state of a counter: $\mathsf{Counter}(s, x, n)$ asserts the existence of a counter at address x, with value n. The first parameter s ranges over an *abstract type* (in this case, $\mathbb{T}_1$), which captures implementation-specific information about the counter[1]. This parameter serves a technical purpose that we shall address shortly. The predicate confers ownership of the counter: it is not possible to have more than one $\mathsf{Counter}(s, x, n)$ for the same value of x.

The specification for the `makeCounter` operation is a simple Hoare triple:

$$\vdash \left\{ \mathsf{True} \right\} \texttt{makeCounter}() \left\{ \exists s \in \mathbb{T}_1. \mathsf{Counter}(s, \mathsf{ret}, 0) \right\}$$

The operation creates a new counter, which is initially set to value 0, and returns its address. The specification says nothing about the granularity of the operation. In fact, the granularity is hardly

[1] To the client, the type is opaque; the implementation realises the type appropriately.

relevant, since no concurrent environment can meaningfully observe the effects of `makeCounter` until its return value is known—that is, once the operation has been completed.

Remark 3.1 (On the abstractly-typed parameters)**.** Many of the proofs of abstract atomicity conclude with a step that quantifies over some fixed parameters in the representation of the data-structure. This generally presumes a rule of the form:

$$\frac{\vdash \left\langle P(x) \right\rangle \ \mathbb{C} \ \left\langle Q(x) \right\rangle}{\vdash \left\langle \exists x.\, P(x) \right\rangle \ \mathbb{C} \ \left\langle \exists x.\, Q(x) \right\rangle}$$

However, such a rule is unsound in general: in the conclusion, we assume that the environment is able to change the value of x, while in the premiss the value cannot be changed. To avoid this unsoundness, we instead expose the parameter (x in this case). The client does not need to know any particular information about x, only that it should not be changed; hence, the type of x can be abstracted. □

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

$$\vdash \mathbb{V} n \in \mathbb{N}. \left\langle \mathsf{Counter}(s, \mathtt{x}, n) \right\rangle \ \mathtt{read(x)} \ \left\langle \mathsf{Counter}(s, \mathtt{x}, n) * \mathsf{ret} = n \right\rangle$$

Intuitively, this specification states that the `read` operation will read the state of the counter *atomically*, even in the presence of concurrent updates by the environment that may change the value of the counter, which are possible, as n is bound by $\mathbb{V}$. However, the environment must preserve the counter and cannot, for instance, deallocate it.

This atomicity means that the resources in the specification may be *shared*—that is, concurrently accessible by multiple threads. Sharing in this way is not possible with ordinary Hoare triples, since they make no guarantee that intermediate steps preserve invariants on the resources. The atomic triple, by contrast, makes a strong guarantee: as long as the concurrent environment guarantees that the (possibly) shared resource $\mathsf{Counter}(s, \mathtt{x}, n)$ is available for some n, the `read` operation will preserve $\mathsf{Counter}(s, \mathtt{x}, n)$ until it reads it; after reading, the operation no longer requires $\mathsf{Counter}(s, \mathtt{x}, n)$, and is consequently oblivious to subsequent transformations by the environment (such as another thread incrementing the counter).

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

Specifying `incr` is similar:

$$\vdash \mathbb{V} n \in \mathbb{N}. \left\langle \mathsf{Counter}(s, \mathtt{x}, n) \right\rangle \ \mathtt{incr(x)} \ \left\langle \mathsf{Counter}(s, \mathtt{x}, n+1) * \mathsf{ret} = n \right\rangle$$

The specification states that `incr` operation will increment the counter atomically and return its previous value. Note that when the operation returns, the counter might have been incremented by another thread if the counter is shared.

The last operation can be specified as follows:

$$\forall n \in \mathbb{N}. \vdash \left\langle \mathsf{Counter}(s, \mathtt{x}, n) \right\rangle \ \mathtt{wkincr(x)} \ \left\langle \mathsf{Counter}(s, \mathtt{x}, n+1) \right\rangle$$

The `wkincr` will atomically update the counter from n to $n+1$, as long the environment guarantees that the shared counter will not change the value before the atomic update. The specification assumes that the environment will not change the value of the counter, since n is not bound by $\mathbb{V}$. This means that if the counter is shared, other threads can concurrently only perform `read` operations until the counter has been incremented. Technically, it is possible for other `incr` or `wkincr` operations to occur between the update and the return of the operation.

3.1.2 Implementation

To verify this implementation against the atomic specification, we must give a concrete interpretation of the abstract predicate Counter. For this, we need to introduce the notion of a *shared region*. A shared region encapsulates some resource that is available to multiple threads[2]. In our example, this resource will be a heap cell that stores the value of the counter described by $x \mapsto n$. A shared region is associated with a protocol, which determines how its contents change over time. Following iCAP [54], the state of a shared region is abstracted, and protocols are expressed as transition systems over these abstract states. A thread may only change the abstract state of a region when it has the *guard* resource associated with the transition to be performed. An interpretation function associates each abstract state of a region with a concrete assertion. In summary, to specify a region we must supply the guards for the region, an abstract state transition system that is labelled by these guards, and a function interpreting abstract states as assertions.

In TaDA, guards consist of abstract resources taken from any separation algebra. This gives us more flexibility in specifying complex usage patterns for regions. For the counter, we need only a very simple guard separation algebra: there is a single, indivisible guard named Inc, as well as the empty guard $\mathbf{0}$. As a separation algebra, guard resources must have a partial composition operator that is associative and commutative. In this case, $\mathbf{0} \bullet x = x = x \bullet \mathbf{0}$, for $x \in \{\mathbf{0}, \text{Inc}\}$, and $\text{Inc} \bullet \text{Inc}$ is not allowed.

The transition system for the region will allow the counter to be incremented using the guard Inc. This is specified by the labelled transition system:

$$\text{Inc} \;:\; \forall n.\, n \rightsquigarrow n+1$$

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

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

$$I(\mathbf{Counter}_a(x, n)) \triangleq x \mapsto n.$$

With this interpretation, the heap cell that contains the value of the counter is always in the region.

[2] Similar constructs exist in other logics such as [63, 11, 18, 12].

We can now give an interpretation to the predicate $\mathsf{Counter}(s, x, n)$ and the abstract type $\mathbb{T}_1$:

$$\begin{aligned} \mathbb{T}_1 &\triangleq \mathsf{RId} \\ \mathsf{Counter}(a, x, n) &\triangleq \mathbf{Counter}_a(x, n) * [\textsc{Inc}]_a \end{aligned}$$

where RId is the set of region identifiers. The abstract predicate asserts that there exists a region with identifier a and is in state n. It also states that there is a guard $[\textsc{Inc}]_a$, which is used to update the region. Note that the first parameter of the $\mathsf{Counter}$ predicate fixes the region identifier.

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

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

This rule establishes that $\mathbb{C}$ atomically updates the region a, from some state $x \in X$ to some state $y \in Q(x)$. To do so, it requires the guard G for the region, which must permit the update according to the appropriate transition system $\mathcal{T}_{\mathbf{t}}(\mathrm{G})^*$, where $\mathbf{t}$ is the region type; this is established by the first premiss. In our case, the region type $\mathbf{t}$ will be $\mathsf{Counter}$, the guard G will be Inc, and the transition system will be $\mathcal{T}_{\mathbf{t}}(\textsc{Inc}) = \{(n, n+1) \mid x \in \mathbb{N}\}$. We use * to denote reflexive-transitive closure.

We use $\mathbf{t}_a(\vec{z}, x)$ to represent a region with region type $\mathbf{t}$, identifier a, parameters $\vec{z}$ and abstract state x. In our example, the region is parametrised with the address of the counter and the abstract state is a natural number. The second premiss introduces two new notations. The first, $a : x \in X \rightsquigarrow Q(x)$, is called the *atomicity context*. The atomicity context records the abstract atomic action that is to be performed. The second, $a \Rrightarrow -$, is the atomic tracking resource. The atomic tracking resource indicates whether the atomic update has occurred (the $a \Rrightarrow \blacklozenge$ indicates it has not) and, if so, the state of the shared region immediately before and after (the $a \Rrightarrow (x, y)$). The resource $a \Rrightarrow \blacklozenge$ also plays two special roles that are normally filled by guards. Firstly, it limits the interference on region a: the environment may only update the state so long as it remains in the set X, as specified by the atomicity context. Secondly, it confers permission for the thread to update the region from state $x \in X$ to any state $y \in Q(x)$; in doing so, the thread also updates $a \Rrightarrow \blacklozenge$ to $a \Rrightarrow (x, y)$. This permission is expressed by the UpdateRegion rule (see below), and ensures that the atomic update only happens once.

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

The second key proof rule is the UpdateRegion rule, which deals with using the atomicity tracking resource to update the region. A simplified version of this rule is as follows:

$$\frac{\vdash \forall x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) \right\rangle \;\mathbb{C}\; \left\langle \exists y \in Q(x).\, I(\mathbf{t}_a(\vec{z}, y)) * Q_1(x, y) \vee I(\mathbf{t}_a(\vec{z}, x)) * Q_2(x) \right\rangle}{a : x \in X \rightsquigarrow Q(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * a \Rrightarrow \blacklozenge\right\} \;\mathbb{C}\; \left\{\begin{array}{c}\exists x \in X.\, (\exists y \in Q(x).\, Q_1(x, y) * \\ a \Rrightarrow (x, y) \vee \mathbf{t}_a(\vec{z}, x) * Q_2(x) * a \Rrightarrow \blacklozenge)\end{array}\right\}}$$

Note that if $y = x$ in the postcondition, the abstract state of the region is not changed and we can either perform the atomic update or not.

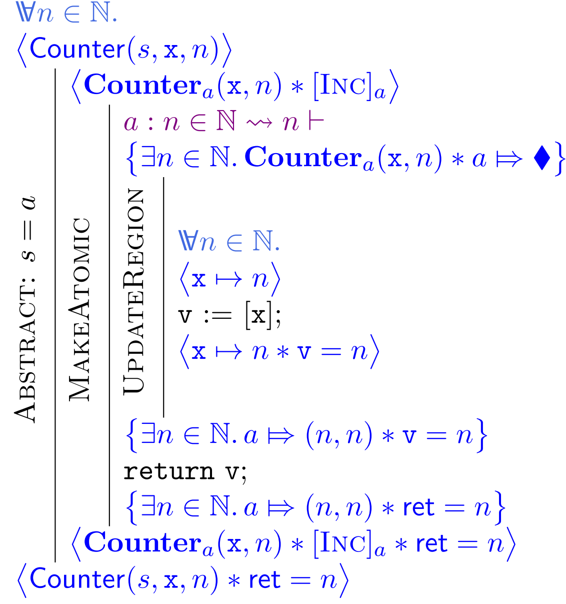


Figure 3.1: Proof outline for the `read` operation.

The proof of the `read` implementation is given in Figure 3.1. The proof first massages the specification into a form where we can apply the MakeAtomic rule. The atomicity context allows the region a to be in any state. The `UpdateRegion` rule performs the atomic action, leaving the region in the same state, and recording the state in the atomic tracking resource. The proof of the `incr` implementation is follows a similar style and is given in Figure 3.2. The main difference is that, when entering the loop, it first performs a read operation and stores the current value of the counter in `v`. The OpenRegion rule is used to open the region without changing its abstract state. We then use the UpdateRegion rule to conditionally perform the atomic action. If the atomic compare-and-set operation succeeds, we transition the region from state n to $n + 1$ and update the atomic tracking component. Finally, the proof of the `wkincr` implementation fixes the value of the region and follows the same style as the previous proofs. Given the fact that the environment is not allowed to change the abstract state of the region until the update, the proof does not need to existentially quantity the state of the region in order to make the assertions stable.

3.2 Ticket Lock

We define a lock module with the operations `acquire` and `release` and a constructor `makeLock`. We consider the ticket lock implementation from §2.1.2 and show how to use the atomic specification from the counter to prove the correctness of the ticket lock. We recall the implementation in Figure 3.4.

3.2.1 CAP Specification

We start by specifying the lock module using a specification based on ownership transfer from CAP [12]. The specification provides two abstract predicates: $\mathsf{IsLock}(x)$, which is a non-exclusive resource that

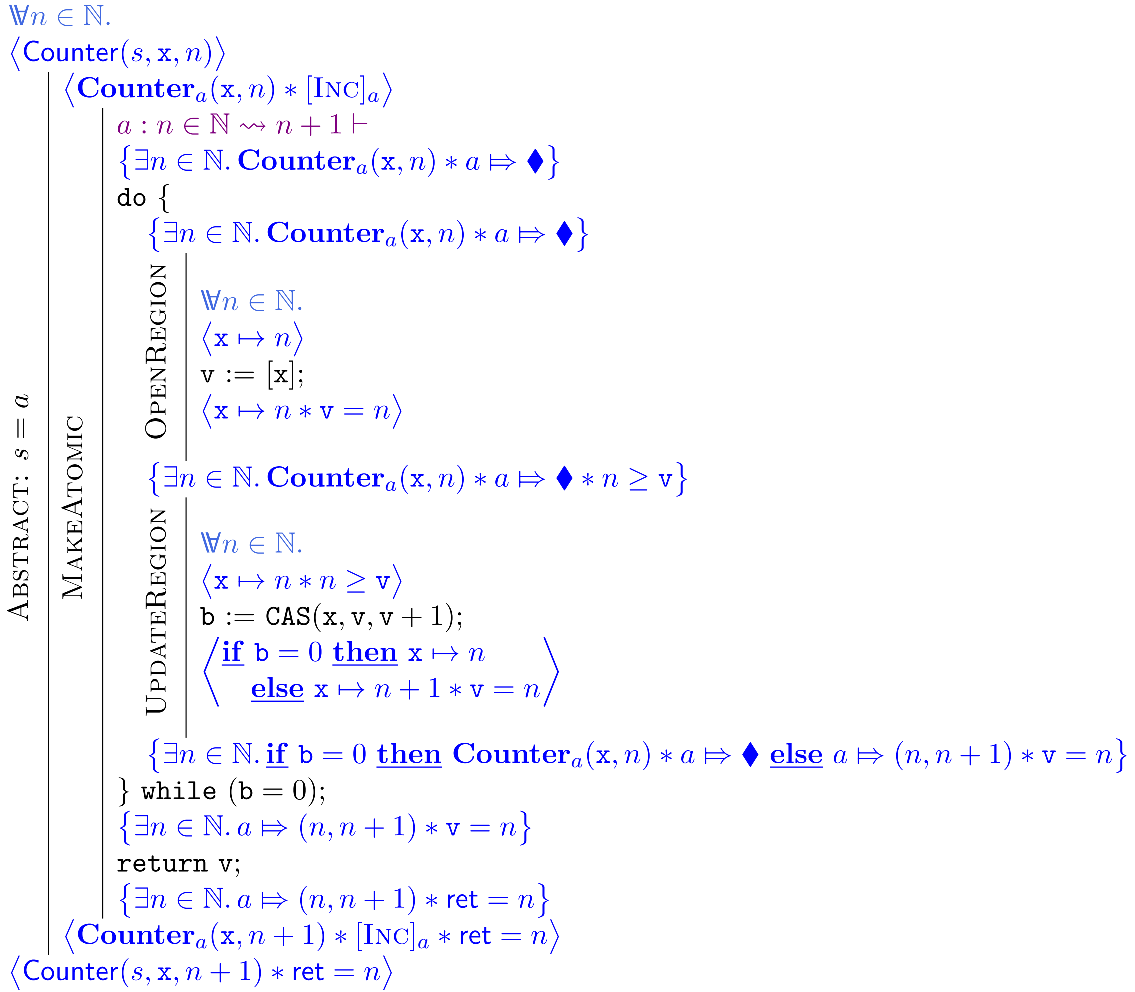


Figure 3.2: Proof outline for the `incr` operation.

allows a thread to compete for the lock; and $\mathsf{Locked}(x)$, which is an exclusive resource that represents that the thread has acquired the lock, and allows it to release the lock. The lock is specified as follows:

$$\vdash \left\{\mathsf{True}\right\} \ \mathtt{makeLock}() \ \left\{\mathsf{IsLock}(\mathsf{ret})\right\}$$
$$\vdash \left\{\mathsf{Locked}(\mathtt{x})\right\} \ \mathtt{release}(\mathtt{x}) \ \left\{\mathsf{True}\right\}$$
$$\vdash \left\{\mathsf{IsLock}(\mathtt{x})\right\} \ \mathtt{acquire}(\mathtt{x}) \ \left\{\mathsf{IsLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\right\}$$
$$\mathsf{IsLock}(x) \iff \mathsf{IsLock}(x) * \mathsf{IsLock}(x)$$
$$\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$$

When a thread acquires the lock, it gets holds of the $\mathsf{Locked}(x)$ that can be used to subsequently release the lock. The last two axioms allow us to duplicate the non-exclusive resource describing the existence of a lock and guarantee that two threads cannot hold the $\mathsf{Locked}(x)$ resource at the same time.

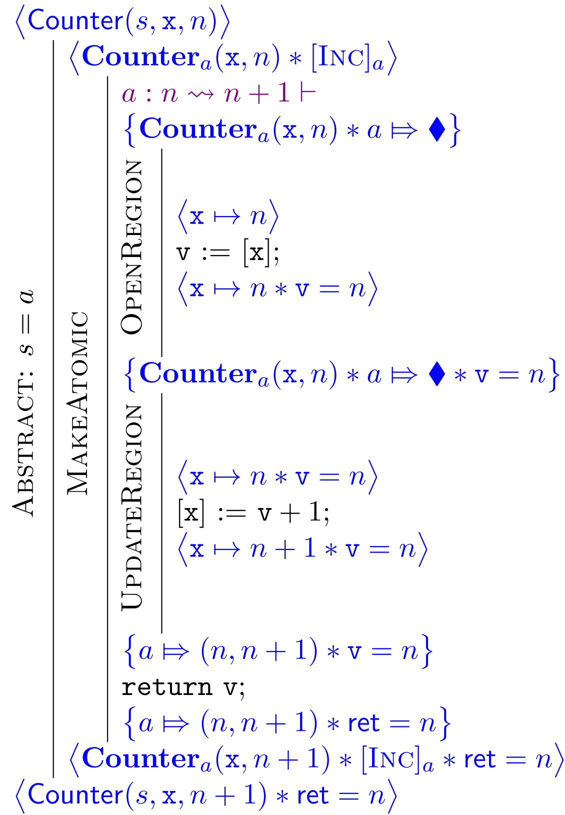


Figure 3.3: Proof outline for the `wkincr` operation.

```
function makeLock() {
  next := makeCounter();
  owner := makeCounter();
  x := alloc(2);
  [x.next] := next;
  [x.owner] := owner;
  return x;
}
```

```
function acquire(x) {
  next := [x.next];
  owner := [x.owner];
  t := incr(next);
  do {
    v := read(owner);
  } while (v ≠ t);
}
```

```
function release(x) {
  owner := [x.owner];
  wkIncr(owner);
}
```

where

$$\mathbb{E}.\mathtt{next} \stackrel{\text{def}}{=} \mathbb{E}$$
$$\mathbb{E}.\mathtt{owner} \stackrel{\text{def}}{=} \mathbb{E} + 1.$$

Figure 3.4: A ticket lock implementation using the counter module.

3.2.2 Implementation

To verify this implementation against the atomic specification, we must first give a concrete interpretation of the abstract predicates. Let us recall that the ticket lock comprises two counters: the first counter records the next available ticket, whereas the second counter records the ticket which currently holds the lock. The lock is considered unlocked when the two counters are equal. In order for a thread to acquire the lock, it must obtain a ticket by incrementing the first counter and then needs to wait until the second counter reaches the value of the obtained ticket. To release the lock, a thread simply increments the second counter.

To verify the implementation, we introduce a new region type, **TLock**. The abstract state of the region will be a natural number n, and there are an infinite amount of them. The abstract state n represents the ticket that currently holds the lock. We associate three guards with the region type.

We have the guard $\textsc{Pending}(n, m)$ that keeps track of how many tickets are being held by threads operating on the lock. We also have a unique guard $\textsc{Key}(v)$, for each $n \leq v < m$ that represent each ticket, and the empty guard $\mathbf{0}$.

We define the partial composition operator in the following way: $\mathbf{0} \bullet x = x = x \bullet \mathbf{0}$ for $x \in \{\mathbf{0}\} \uplus \{\textsc{Pending}(n) \mid n \in \mathbb{N}\} \uplus \{\textsc{Key}(n) \mid n \in \mathbb{N}\}$. $\textsc{Pending}(n_1, m_1) \bullet \textsc{Pending}(n_2, m_2)$ is not allowed and $\textsc{Key}(n) \bullet \textsc{Key}(m)$ is only allowed for $n \neq m$. Moreover, $\textsc{Pending}(n, m) \bullet \textsc{Key}(v)$ is allowed if and only if $n \leq v < m$. This ensures that there can only be tickets for a particular v if they correspond to the ones tracked by the guard $\textsc{Pending}(n, m)$. Finally, we define the composition to allow the creation and destruction of $\textsc{Key}(v)$ guards as follows:

$$\begin{aligned} \textsc{Pending}(n, m) &= \textsc{Pending}(n, m+1) \bullet \textsc{Key}(m), \text{ if } n \leq m \\ \textsc{Pending}(n, m) \bullet \textsc{Key}(n) &= \textsc{Pending}(n+1, m), \text{ if } n < m \end{aligned}$$

The first equality allows us to create new guards and the second allows us to remove guards as long as they have the minimum value.

For the guards, we have constructed an instance of the authoritative monoid of Iris [35]. An alternative approach would be to have only KEY guards, one for each natural number, and define PENDING in terms of those KEY elements, such as the ones used in [7, 12].

The labelled transition system is as follows:

$$\textsc{Key}(n) \ : \ n \rightsquigarrow n+1$$

It guarantees that a thread must hold the guard $\textsc{Key}(n)$ in order to perform the transition.

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

$$\begin{aligned} I(\mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n)) \triangleq \exists m.\, x.\mathtt{owner} \mapsto \mathit{owner} * x.\mathtt{next} \mapsto \mathit{next} * \mathsf{Counter}(s, \mathit{owner}, n) \\ * \mathsf{Counter}(t, \mathit{next}, m) * [\textsc{Pending}(n, m)]_a * n \leq m \end{aligned}$$

The region has five parameters, x is the address of the lock and allows us to retrieve both counter addresses, located at *owner* and *next* respectively. Moreover, we also have s and t, which correspond to the each counter abstract predicate.

We now define the interpretation of the predicates as follows:

$$\begin{aligned} \mathsf{IsLock}(x) &\triangleq \exists a, s, t, \mathit{owner}, \mathit{next}, n.\, \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) \\ \mathsf{Locked}(x) &\triangleq \exists a, s, t, \mathit{owner}, \mathit{next}, n.\, \mathbf{TLock}_a(x, s, t, \mathit{owner}, \mathit{next}, n) * [\textsc{Key}(n)]_a \end{aligned}$$

The abstract predicate $\mathsf{IsLock}(x)$ asserts there is a region with identifier a and the region is in some state n. $\mathsf{Locked}(x)$ asserts the same as the previous and additionally states that there is a guard $[\textsc{Key}(n)]_a$ which will be used to update the region. Note that by holding the $\mathsf{Locked}(x)$ exclusively, we guarantee that the region abstract state cannot by changed by the environment, as they do not hold the necessary guards to perform such update.

It remains to prove the specifications for the operations and the axioms. The key proof is the

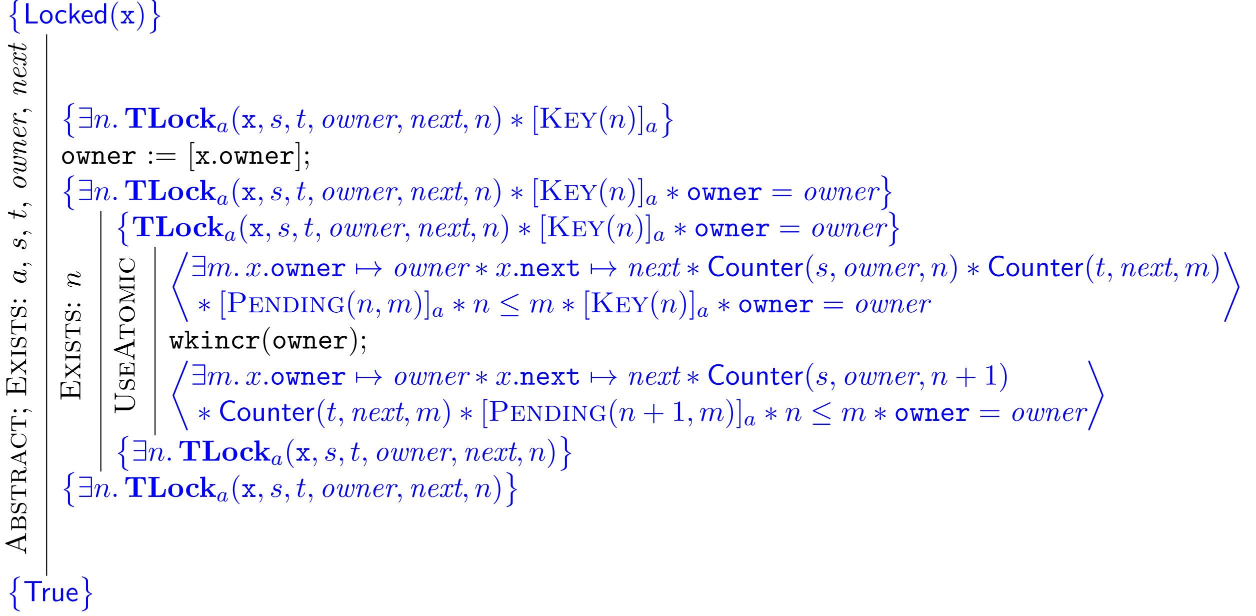


Figure 3.5: Proof outline for the `release` operation.

UseAtomic rule. A simplified version of the rule is as follows:

$$\frac{\forall x \in X.\,(x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \vdash \forall\!\!\!\forall x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \right\rangle \ \mathbb{C}\ \left\langle I(\mathbf{t}_a(\vec{z}, f(x))) * Q(x) \right\rangle}{\vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * [\mathrm{G}]_a\right\} \ \mathbb{C}\ \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, f(x)) * Q(x)\right\}}$$

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

The proofs of the `release` and `acquire` operations are given in Figure 3.5 and Figure 3.6. In the `release` proof, the interesting step is the use of the UseAtomic, where we first fix the region for some n by owning the guard $\textsc{Key}(n)$ which guarantees that the assertion is stable. We then open the region and combine the $\textsc{Key}(n)$ with the $\textsc{Pending}(n, m)$ when the counter is atomically incremented. After this happens, the guard $\textsc{Key}(n)$ is lost and as such we need to weaken to postcondition to allow an arbitrary n. In the end, we weaken the postcondition to True to satisfy the specification. It would also be possible to give a specification that has $\mathsf{IsLock}(x)$ as the postcondition instead.

The `acquire` proof uses the $\forall\!\!\!\forall$ quantifier in the premiss of the UseAtomic rule to account for the fact that, in the precondition, the lock could be in any state n. The first use of the UseAtomic rule increments the counter and retrieves a $\textsc{Key}(\mathtt{t})$ for the value read. After the read, because we own $\textsc{Key}(\mathtt{t})$, we can guarantee that the state of the region cannot be larger than `t`, i.e. that the environment does not have the necessary guards to perform such a transition. The loop then simply waits until the state of the region matches the ticket. When that happens, we know it cannot change as long as we own the guard $\textsc{Key}(\mathtt{t})$ and as such we can satisfy the $\mathsf{Locked}(x)$ predicate.

The axiom $\mathsf{IsLock}(x) \iff \mathsf{IsLock}(x) * \mathsf{IsLock}(x)$ follows from the duplicability of region assertions: *i.e.* $\mathbf{TLock}_a(x, \mathit{owner}, \mathit{next}, n) \equiv \mathbf{TLock}_a(x, \mathit{owner}, \mathit{next}, n) * \mathbf{TLock}_a(x, \mathit{owner}, \mathit{next}, n)$. Finally, the axiom $\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$ follows from the fact that $\textsc{Key}(n) \bullet \textsc{Key}(n)$ is undefined.

$$
\begin{array}{l}
\{\mathsf{IsLock}(\mathtt{x})\} \\
\quad \text{ABSTRACT; EXISTS: } a,\ s,\ t,\ \mathit{owner},\ \mathit{next} \\
\quad \{\exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n)\} \\
\quad \mathtt{next} := [\mathtt{x.next}]; \\
\quad \mathtt{owner} := [\mathtt{x.owner}]; \\
\quad \{\exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n) * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner}\} \\
\quad\quad \text{USEATOMIC} \\
\quad\quad \forall n. \\
\quad\quad \left\langle \begin{array}{l} \mathsf{Counter}(s, \mathit{owner}, n) * \mathsf{Counter}(t, \mathit{next}, m) * [\text{PENDING}(n, m)]_a \\ * n \leq m * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner} \end{array} \right\rangle \\
\quad\quad \mathtt{t} := \mathtt{incr}(\mathtt{x.next}); \\
\quad\quad \left\langle \begin{array}{l} \mathsf{Counter}(s, \mathit{owner}, n) * \mathsf{Counter}(t, \mathit{next}, m+1) * [\text{PENDING}(n, m+1)]_a \\ * n \leq m * [\text{KEY}(\mathtt{t})]_a * \mathtt{t} = m * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner} \end{array} \right\rangle \\
\quad \{\exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n) * [\text{KEY}(\mathtt{t})]_a * n \leq \mathtt{t} * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner}\} \\
\quad \mathtt{do}\ \{ \\
\quad\quad \{\exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n) * [\text{KEY}(\mathtt{t})]_a * n \leq \mathtt{t} * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner}\} \\
\quad\quad\quad \text{USEATOMIC} \\
\quad\quad\quad \forall n. \\
\quad\quad\quad \left\langle \begin{array}{l} \mathsf{Counter}(s, \mathit{owner}, n) * \mathsf{Counter}(t, \mathit{next}, m) * [\text{PENDING}(n, m)]_a \\ * n \leq m * [\text{KEY}(\mathtt{t})]_a * n \leq \mathtt{t} * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner} \end{array} \right\rangle \\
\quad\quad\quad \mathtt{v} := \mathtt{read}(\mathtt{x.owner}); \\
\quad\quad\quad \left\langle \begin{array}{l} \mathsf{Counter}(s, \mathit{owner}, n) * \mathsf{Counter}(t, \mathit{next}, m) * [\text{PENDING}(n, m)]_a \\ * n \leq m * [\text{KEY}(\mathtt{t})]_a * n \leq \mathtt{t} * n = \mathtt{v} * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner} \end{array} \right\rangle \\
\quad\quad \left\{ \begin{array}{l} \exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n) * [\text{KEY}(\mathtt{t})]_a * n \leq \mathtt{t} * n \geq \mathtt{v} \\ * \mathtt{next} = \mathit{next} * \mathtt{owner} = \mathit{owner} \end{array} \right\} \\
\quad \}\ \mathtt{while}\ (\mathtt{v} \neq \mathtt{t}); \\
\quad \{\exists n.\, \mathbf{TLock}_a(\mathtt{x}, s, t, \mathit{owner}, \mathit{next}, n) * [\text{KEY}(\mathtt{t})]_a * n = \mathtt{t}\} \\
\{\mathsf{Locked}(\mathtt{x})\}
\end{array}
$$

Figure 3.6: Proof outline for the `acquire` operation.

The counter specifications shown in this section are strong: a client can derive the abstract disjoint specifications from them. Moreover, they are strong enough to support synchronisation: the correctness of the ticket lock can be justified from the counter specifications. These approaches to specification are expressive enough to enforce obligations on both the client and the implementation. By contrast, CAP specifications tend to unduly restrict the client (e.g. a counter specification cannot be used for synchronisation), while linearisability specifications tend to unduly restrict the implementation (e.g. a counter cannot provide a `wkincr` operation).

4 TaDA Logic

We formalise the TaDA logic and prove the soundness of the proof system. In order to do so, we first present the programming language (§4.1) and give its operational semantics (§4.2). We then formalise the program logic in §4.4 and present a system of proof rules for reasoning about programs. Finally, we provide the semantics for the logic (§4.5) and show the soundness of the proof system in §4.6.

4.1 Programming Language

We consider an imperative programming language with dynamically allocated heap and mutable local variables. The operations that manipulate the heap are atomic. Local variables have local scope, in that they can only be accessed within a function. Moreover, the language supports the spawning of new threads using a fork command.

Definition 4.1 (Variable Names)**.** Assume a set of *variable names* Var, ranged over by $\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots$

Definition 4.2 (Expressions)**.** The set of integer-valued *expressions* Expr, ranged over by $\mathbb{E}, \mathbb{E}_1, \ldots,$ is defined by the grammar:

$$\begin{array}{rcl} \mathbb{E} & ::= & \mathtt{x} \\ & | & v \\ & | & \mathbb{E}_1 + \mathbb{E}_2 \\ & | & \mathbb{E}_1 - \mathbb{E}_2 \\ & | & \mathbb{E}_1 \times \mathbb{E}_2 \\ & | & \mathbb{E}_1 \div \mathbb{E}_2 \end{array}$$

where $v \in \mathbb{Z}$.

Expressions can be either variables or integer constants or the compositions of these two using standard binary arithmetic operations.

Definition 4.3 (Boolean Expressions)**.** The set of *boolean expressions* BExpr, ranged over by $\mathbb{B}, \mathbb{B}_1, \ldots,$ is defined by the grammar:

$$\begin{array}{rcl} \mathbb{B} & ::= & \mathtt{true} \\ & | & \mathtt{false} \\ & | & \mathbb{E}_1 = \mathbb{E}_2 \\ & | & \mathbb{E}_1 < \mathbb{E}_2 \\ & | & \mathbb{B}_1 \ \mathtt{and} \ \mathbb{B}_2 \\ & | & \mathbb{B}_1 \ \mathtt{or} \ \mathbb{B}_2 \\ & | & \mathtt{not} \ \mathbb{B} \end{array}$$

Boolean expressions can be either constants (`true` and `false`), or comparisons on expressions, or boolean combinations of boolean expressions.

Definition 4.4 (Function Names)**.** Assume a set of *function names* Fun, ranged over by $\mathtt{f}, \mathtt{g}, \ldots$

Definition 4.5 (Commands)**.** The set of *commands* Cmd, ranged over by $\mathbb{C}, \mathbb{C}_1, \ldots$, is defined by the following grammar:

$$
\begin{array}{rcll}
\mathbb{C} & ::= & \mathtt{skip}; & \text{Empty command} \\
& \mid & \mathbb{C}_1\ \mathbb{C}_2 & \text{Sequencing} \\
& \mid & \mathtt{while}\ (\mathbb{B})\ \{\mathbb{C}\} & \text{Loop} \\
& \mid & \mathtt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \mathtt{else}\ \{\mathbb{C}_2\} & \text{Conditional} \\
& \mid & \mathtt{x} := \mathtt{f}(\vec{\mathbb{E}}); & \text{Function call} \\
& \mid & \mathtt{fork}\ \mathtt{f}(\vec{\mathbb{E}}); & \text{Spawn thread} \\
& \mid & \mathtt{x} := \mathtt{alloc}(\mathbb{E}); & \text{Allocation} \\
& \mid & \mathtt{x} := \mathbb{E}; & \text{Assignment} \\
& \mid & \mathtt{x} := [\mathbb{E}]; & \text{Lookup} \\
& \mid & [\mathbb{E}_1] := \mathbb{E}_2; & \text{Mutation} \\
& \mid & \mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3); & \text{Compare-and-set}
\end{array}
$$

We identify commands up to associativity of sequencing: that is, $(\mathbb{C}_1\ \mathbb{C}_2)\ \mathbb{C}_3 = \mathbb{C}_1\ (\mathbb{C}_2\ \mathbb{C}_3)$.

Note that we can encode the command $\mathtt{do}\ \{\mathbb{C}\}\ \mathtt{while}\ (\mathbb{B})$ as $\mathbb{C}\ \mathtt{while}\ (\mathbb{B})\ \{\mathbb{C}\}$.

Definition 4.6 (Function Bodies)**.** The set of *function bodies* $\mathsf{FunBody}$, ranged over by $\mathbb{F}, \mathbb{F}_1, \ldots$, is defined as:

$$\mathbb{F} ::= \mathbb{C}\ \mathtt{return}\ \mathbb{E};$$

For simplicity, we restrict the return of a function to occur only at the end. When we are not interested on the return value we might omit it, in such cases consider it to be sugar syntax for a function with a return at the end with some expression.

Definition 4.7 (Programs)**.** The set of *programs* Prog, ranged over by $\mathbb{P}, \mathbb{P}_1, \ldots$, is defined as a finite sequence of function definitions:

$$\mathbb{P} ::= \epsilon \mid \mathbb{P}, \mathtt{function}\ \mathtt{f}(\vec{\mathtt{x}})\ \{\mathbb{F}\}$$

where function names are assumed to be pairwise distinct.

4.2 Operational Semantics

We present the operational semantics of the programming language as a Views-style [13] labelled transition system. Transitions are labelled by atomic actions. The labelled transition system splits the control-flow aspect of the execution from the heap-transforming aspect of the execution which is represented by the labels of each transition.

Definition 4.8 (Program Values)**.** Fix the set of *program values* $\mathsf{Val} \stackrel{\text{def}}{=} \mathbb{Z}$ to be the integers, ranged over by $v, v_1, \ldots$

Definition 4.9 (Variable Store)**.** A *variable store* $\mathsf{Store} \stackrel{\text{def}}{=} \mathsf{Var} \to \mathsf{Val}$ is a total function from program variables to program values, ranged over by $\sigma, \sigma_1, \ldots$

Definition 4.10 (Semantics of Expressions)**.** The *semantics of expressions* $\mathcal{E}[\![-]\!]_{-} : \mathsf{Expr} \times \mathsf{Store} \to \mathsf{Val}$, is defined with respect to the variable store by:

$$
\begin{aligned}
\mathcal{E}[\![\mathtt{x}]\!]_\sigma &\stackrel{\text{def}}{=} \sigma(\mathtt{x}) \\
\mathcal{E}[\![v]\!]_\sigma &\stackrel{\text{def}}{=} v \\
\mathcal{E}[\![\mathbb{E}_1 + \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma + \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \\
\mathcal{E}[\![\mathbb{E}_1 - \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma - \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \\
\mathcal{E}[\![\mathbb{E}_1 \times \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma \times \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \\
\mathcal{E}[\![\mathbb{E}_1 \div \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \begin{cases} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma \div \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma & \text{if } \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \neq 0 \\ 42 & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that the store is a total function and, as such, if a variable is not initialised, then its value will be arbitrary but consistent in the sense that multiple accesses of the same variable returns the same value.

Definition 4.11 (Semantics of Boolean Expressions)**.** The semantics of *boolean expressions* $\mathcal{B}[\![-]\!]_{-} : \mathsf{BExpr} \times \mathsf{Store} \to \mathsf{Bool}$, is defined with respect to the variable store by:

$$
\begin{aligned}
\mathcal{B}[\![\mathtt{true}]\!]_\sigma &\stackrel{\text{def}}{=} \mathsf{True} \\
\mathcal{B}[\![\mathtt{false}]\!]_\sigma &\stackrel{\text{def}}{=} \mathsf{False} \\
\mathcal{B}[\![\mathbb{E}_1 = \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma = \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \\
\mathcal{B}[\![\mathbb{E}_1 < \mathbb{E}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{E}[\![\mathbb{E}_1]\!]_\sigma < \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma \\
\mathcal{B}[\![\mathbb{B}_1 \ \mathtt{and}\ \mathbb{B}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{B}[\![\mathbb{B}_1]\!]_\sigma \wedge \mathcal{B}[\![\mathbb{B}_2]\!]_\sigma \\
\mathcal{B}[\![\mathbb{B}_1 \ \mathtt{or}\ \mathbb{B}_2]\!]_\sigma &\stackrel{\text{def}}{=} \mathcal{B}[\![\mathbb{B}_1]\!]_\sigma \vee \mathcal{B}[\![\mathbb{B}_2]\!]_\sigma \\
\mathcal{B}[\![\mathtt{not}\ \mathbb{B}]\!]_\sigma &\stackrel{\text{def}}{=} \neg\mathcal{B}[\![\mathbb{B}]\!]_\sigma
\end{aligned}
$$

The programming language allows function calls inside a function body. This requires the creation of a new store that is used to evaluate the body of the function being called. We model the stack of stores and the current execution point using continuations.

In order to define the operational semantics over commands, we extend the commands to include execution stacks that occur as a result of a function called.

Definition 4.12 (Extended Commands)**.** The set of extended commands ExtCmd, ranged over by $c, c_1, \ldots$, is defined by the following grammar:

$$
\begin{aligned}
c \quad ::= &\quad \mathtt{x} := (\sigma, c \ \mathtt{return}\ \mathbb{E};)\,; \\
| &\quad \mathbb{C} \\
| &\quad c\,\mathbb{C}
\end{aligned}
$$

The first construct represents the execution stack. It is used to keep track of the point to which the function call occurred along with each store for each of the successive function calls. The second construct represents the current command being executed. Finally, the third construct allows us to sequence extended commands with commands.

Definition 4.13 (Continuations)**.** The set of continuations Cont, ranged over by $k, k_1, \ldots$, is defined

by the following grammar:

$$k ::= c \texttt{ return } \mathbb{E};$$

Definition 4.14 (Threads)**.** A *thread* $\mathsf{Thread} \stackrel{\text{def}}{=} \mathsf{Store} \times \mathsf{Cont}$ is a pair of variable store and a continuation, ranged over by $t, t_1, \ldots$

Definition 4.15 (Labels of Atomic Commands)**.** The set of transition labels $\mathsf{AAction}$, ranged over by α, is described by the following grammar:

$$
\begin{array}{rcll}
\alpha & ::= & \mathsf{id} & \text{Unchanged} \\
& | & \mathsf{alloc}(v_1, v_2) & \text{Allocation} \\
& | & \mathsf{read}(v_1, v_2) & \text{Lookup} \\
& | & \mathsf{write}(v_1, v_2) & \text{Mutation} \\
& | & \mathsf{cas}(v_1, v_2, v_3, v_4) & \text{Compare-and-set} \\
& | & \mathsf{spawn}(\texttt{f}, \vec{v}) & \text{Fork thread} \\
& | & \lightning & \text{Fault}
\end{array}
$$

The labels of atomic commands describe all possible operations that can manipulate the heap. Note that the labels $\mathsf{alloc}(v_1, v_2)$, $\mathsf{read}(v_1, v_2)$ and $\mathsf{cas}(v_1, v_2, v_3, v_4)$ use the last argument to represent the returned values. In the label alloc, v_1 to represents the size of heap allocated and v_2 its address. In the label read, v_1 is the heap address being read and v_2 the value stored at that address. Finally, in the label cas, v_1 is the heap address that the compare-and-set is performed on, v_2 is the expected value, v_3 the new value and v_4 represents if the operation was successful or not.

Definition 4.16 (Function Environment)**.** A function environment $\mathsf{FEnv} \stackrel{\text{def}}{=} \mathsf{Fun} \rightharpoonup_{fin} (\mathsf{Var}^* \times \mathsf{FunBody})$ is a finite partial mapping from function names to parameters and bodies, ranged over by η. We denote a well-defined lookup as $\eta(\texttt{f}) = (\vec{\texttt{x}}, \mathbb{F})$. Moreover, we refer to the parameters as $vars(\eta(\texttt{f}))$ and to the body of the function as $code(\eta(\texttt{f}))$. A function environment for a given program $\mathbb{P}$ must coincide with its function definitions.

Definition 4.17 (Thread Operational Semantics)**.** The thread operational semantics is defined as a labelled transition relation of individual threads

$$- \xrightarrow{-}_{-} - : \mathsf{Thread} \times \mathsf{AAction} \times \mathsf{FEnv} \times \mathsf{Thread}$$

defined by the following rule:

$$\textsc{FunctionStep}\quad \frac{(\sigma_1, c_1) \xrightarrow{\alpha}_{\eta} (\sigma_2, c_2)}{(\sigma_1, c_1 \texttt{ return } \mathbb{E};) \xrightarrow{\alpha}_{\eta} (\sigma_2, c_2 \texttt{ return } \mathbb{E};)}$$

The FunctionStep reduces extended command in the function body until it is equal to `skip`;.

Definition 4.18 (Extend Commands Operational Semantics)**.** The extended commands operational semantics is defined as a labelled transition relation

$$(-,-) \xrightarrow{-}_{-} (-,-) : \mathsf{Store} \times \mathsf{ExtCmd} \times \mathsf{AAction} \times \mathsf{FEnv} \times \mathsf{Store} \times \mathsf{ExtCmd}$$

defined by the following rules:

Skip
$$\frac{}{(\sigma, \texttt{skip;}\ c) \xrightarrow{\textsf{id}}_\eta (\sigma, c)}$$

Sequencing
$$\frac{(\sigma_1, c_1) \xrightarrow{\alpha}_\eta (\sigma_2, c_2)}{(\sigma_1, c_1\ \mathbb{C}) \xrightarrow{\alpha}_\eta (\sigma_2, c_2\ \mathbb{C})}$$

LoopTrue
$$\frac{\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \textsf{True}}{(\sigma, \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}) \xrightarrow{\textsf{id}}_\eta (\sigma, \mathbb{C}\ \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\})}$$

LoopFalse
$$\frac{\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \textsf{False}}{(\sigma, \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}) \xrightarrow{\textsf{id}}_\eta (\sigma, \texttt{skip;})}$$

ConditionalTrue
$$\frac{\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \textsf{True}}{(\sigma, \texttt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \texttt{else}\ \{\mathbb{C}_2\}) \xrightarrow{\textsf{id}}_\eta (\sigma, \mathbb{C}_1)}$$

ConditionalFalse
$$\frac{\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \textsf{False}}{(\sigma, \texttt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \texttt{else}\ \{\mathbb{C}_2\}) \xrightarrow{\textsf{id}}_\eta (\sigma, \mathbb{C}_2)}$$

FunctionCall
$$\frac{\mathcal{E}[\![\vec{\mathbb{E}}]\!]_{\sigma_1} = \sigma_2(\mathit{vars}(\eta(\texttt{f})))}{\left(\sigma_1, \texttt{x} := \texttt{f}(\vec{\mathbb{E}});\right) \xrightarrow{\textsf{id}}_\eta (\sigma_1, \texttt{x} :=\ (\sigma_2, \mathit{code}(\eta(\texttt{f})))\,;)}$$

FunctionCallStep
$$\frac{(\sigma_1, c_1) \xrightarrow{\alpha}_\eta (\sigma_2, c_2)}{(\sigma_3, \texttt{x} :=\ (\sigma_1, c_1\ \texttt{return}\ \mathbb{E};)\,;) \xrightarrow{\alpha}_\eta (\sigma_3, \texttt{x} :=\ (\sigma_2, c_2\ \texttt{return}\ \mathbb{E};)\,;)}$$

Return
$$\frac{}{(\sigma_1, \texttt{x} :=\ (\sigma_2, \texttt{skip;}\ \texttt{return}\ \mathbb{E};)\,;) \xrightarrow{\textsf{id}}_\eta (\sigma_1[\texttt{x} \mapsto \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_2}], \texttt{skip;})}$$

Fork
$$\frac{}{\left(\sigma, \texttt{fork f}(\vec{\mathbb{E}});\right) \xrightarrow{\textsf{spawn}(\texttt{f}, \mathcal{E}[\![\vec{\mathbb{E}}]\!]_\sigma)}_\eta (\sigma, \texttt{skip;})}$$

Allocation
$$\frac{}{(\sigma, \texttt{x} := \texttt{alloc}(\mathbb{E});) \xrightarrow{\textsf{alloc}(\mathcal{E}[\![\mathbb{E}]\!]_\sigma, v)}_\eta (\sigma[\texttt{x} \mapsto v], \texttt{skip;})}$$

Assignment
$$\frac{}{(\sigma, \texttt{x} := \mathbb{E};) \xrightarrow{\textsf{id}}_\eta (\sigma[\texttt{x} \mapsto \mathcal{E}[\![\mathbb{E}]\!]_\sigma], \texttt{skip;})}$$

Lookup
$$\frac{}{(\sigma, \texttt{x} := [\mathbb{E}];) \xrightarrow{\textsf{read}(\mathcal{E}[\![\mathbb{E}]\!]_\sigma, v)}_\eta (\sigma[\texttt{x} \mapsto v], \texttt{skip;})}$$

Mutation
$$\frac{}{(\sigma, [\mathbb{E}_1] := \mathbb{E}_2;) \xrightarrow{\textsf{write}(\mathcal{E}[\![\mathbb{E}_1]\!]_\sigma, \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma)}_\eta (\sigma, \texttt{skip;})}$$

$$\textsc{CompareAndSet}\quad \frac{}{(\sigma, \mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3);) \xrightarrow{\mathsf{cas}(\mathcal{E}[\![\mathbb{E}_1]\!]_\sigma, \mathcal{E}[\![\mathbb{E}_2]\!]_\sigma, \mathcal{E}[\![\mathbb{E}_3]\!]_\sigma, v)}_\eta (\sigma[\mathtt{x} \mapsto v], \mathtt{skip};)}$$

The SKIP leaves the state unchanged and continues on the next command. The SEQUENCING performs a step of c_1, reducing it to c_2, when it is not equal to `skip;`. The LOOPTRUE reduces to a sequence containing the body of the loop, followed by the loop itself. This corresponds to executing once the loop body if the boolean expression holds. The LOOPFALSE reduces to `skip;` if the boolean expression does not hold, stopping the loop. The CONDITIONALTRUE and CONDITIONALFALSE simply reduce the conditional to one of its branches, without affecting the state, depending on whether the boolean expression holds. The FUNCTIONCALL creates a new store to execute the function body. The FUNCTIONCALLSTEP executes steps in a function previously called. Note that the function previously called may contain further function calls. The RETURN terminates a function previously called and destroys its store. The SPAWN spawns a new thread, which has no effect for the current thread. Finally, the remaining cases perform basic operations on the state of the program.

Definition 4.19 (Thread Identifiers)**.** Assume a set of *thread identifiers* TId.

Definition 4.20 (Thread Pools)**.** A thread pool $T \in \mathsf{ThreadPool} \stackrel{\text{def}}{=} \mathsf{TId} \rightharpoonup_{\mathit{fin}} \mathsf{Thread}$ is a finite partial function from thread identifiers to threads.

Definition 4.21 (Heaps)**.** A *heap* $\mathsf{Heap} \stackrel{\text{def}}{=} \mathsf{Addr} \rightharpoonup_{\mathit{fin}} \mathsf{Val}$ is a finite partial function from addresses to values, ranged over by $h, h_1, \ldots$ Heaps form a partial commutative monoid $(\mathsf{Heap}, \uplus, \emptyset)$, where $\uplus$ is the disjoint union of partial functions, and $\emptyset$ is the partial function with the empty domain. The structure of the partial commutative monoid induces a resource order $\leq$, defined as follows:

$$h_1 \leq h_2 \stackrel{\text{def}}{\iff} \exists h_3.\, h_1 \uplus h_3 = h_2.$$

Note that $\leq$ is a partial order.

Definition 4.22 (Program States)**.** A *program state* is a pair of a heap and a thread pool. Moreover, program states are enriched with an exceptional faulting state, denoted $\lightning$, to represent the result of an invalid memory access:

$$\mathsf{State} \stackrel{\text{def}}{=} (\mathsf{Heap} \uplus \{\lightning\}) \times \mathsf{ThreadPool}$$

Definition 4.23 (Thread Pool Operational Semantics)**.** The labelled transition relation of a thread pool

$$- \xrightarrow{-}_{-} - : \mathsf{ThreadPool} \times \mathsf{AAction} \times \mathsf{FEnv} \times \mathsf{ThreadPool}$$

is defined by:

$$\textsc{ThreadReturn}\quad \frac{}{T \parallel (\sigma, \mathtt{skip;\ return}\ \mathbb{E};) \xrightarrow{\mathsf{id}}_\eta T}$$

$$\textsc{ThreadSpawn}\quad \frac{t \xrightarrow{\mathsf{spawn}(\mathtt{f}, \vec{v})}_\eta t' \qquad \sigma(\mathit{vars}(\eta(\mathtt{f}))) = \vec{v}}{T \parallel t \xrightarrow{\mathsf{id}}_\eta T \parallel t' \parallel (\sigma, \mathit{code}(\eta(\mathtt{f})))}$$

$$\frac{t \xrightarrow{\alpha}_\eta t' \qquad \alpha \notin \{\mathsf{spawn}(\mathtt{f}, \vec{v}) \mid \mathtt{f} \in \mathsf{Fun}, \vec{v} \in \mathsf{Val}^*\}}{T \parallel t \xrightarrow{\alpha}_\eta T \parallel t'}\ \textsc{ThreadStep}$$

The ThreadReturn reduces the number of threads in the thread pool when the thread has finished its execution. A thread being executed can perform a fork operation spawning a new thread, while the other operations have no effect on the number of threads in the thread pool. The ThreadSpawn creates a new thread when an existing thread performs a fork operation. The newly created thread has a store σ with the parameters of the function initialised by the values passed by the fork operation. The ThreadStep updates a thread in the thread pool, by performing a non-spawn action as defined by the thread operational semantics.

Definition 4.24 (Addresses)**.** Fix the set of *addresses* $\mathsf{Addr} \stackrel{\text{def}}{=} \mathbb{Z}^+$ to be the set of positive integers, where $\mathsf{Addr} \subseteq \mathsf{Val}$.

Definition 4.25 (Interpretation of Atomic Commands)**.** The Interpretation of atomic action labels is given as a function

$$[\![-]\!] : \mathsf{AAction} \to \mathsf{Heap} \to \mathcal{P}(\mathsf{Heap} \uplus \{\lightning\})$$

that associates each atomic command with a non-deterministic state transformer and is defined as follows:

$$
\begin{aligned}
[\![\mathsf{id}]\!] &\stackrel{\text{def}}{=} \lambda h.\, \{h\} \\
[\![\mathsf{alloc}(v_1, v_2)]\!] &\stackrel{\text{def}}{=} \lambda h. \begin{cases}
\{h[\vec{v_3} \mapsto \vec{v_4}] \mid \vec{v_4} \in \mathsf{Val}^{v_1}\} & \text{if } v_1 > 0 \text{ and } v_2 \in \mathsf{Addr} \text{ and} \\
& \ \{v_2, \dots, v_2 + v_1 - 1\} \cap \mathrm{dom}(h) = \emptyset \\
& \ \text{and } \vec{v_3} = (v_2, \dots, v_2 + v_1 - 1) \\
\emptyset & \text{if } v_1 > 0 \text{ and } v_2 \in \mathsf{Addr} \text{ and} \\
& \ \{v_2, \dots, v_2 + v_1 - 1\} \cap \mathrm{dom}(h) \neq \emptyset \\
& \ \text{or } v_1 > 0 \text{ and } v_2 \notin \mathsf{Addr} \\
\{\lightning\} & \text{otherwise.}
\end{cases} \\
[\![\mathsf{read}(v_1, v_2)]\!] &\stackrel{\text{def}}{=} \lambda h. \begin{cases}
\{h\} & \text{if } v_1 \in \mathrm{dom}(h) \text{ and } h(v_1) = v_2 \\
\emptyset & \text{if } v_1 \in \mathrm{dom}(h) \text{ and } h(v_1) \neq v_2 \\
\{\lightning\} & \text{otherwise.}
\end{cases} \\
[\![\mathsf{write}(v_1, v_2)]\!] &\stackrel{\text{def}}{=} \lambda h. \begin{cases}
\{h[v_1 \mapsto v_2]\} & \text{if } v_1 \in \mathrm{dom}(h) \\
\{\lightning\} & \text{otherwise.}
\end{cases} \\
[\![\mathsf{cas}(v_1, v_2, v_3, v_4)]\!] &\stackrel{\text{def}}{=} \lambda h. \begin{cases}
\{h[v_1 \mapsto v_3]\} & \text{if } v_1 \in \mathrm{dom}(h) \text{ and } h(v_1) = v_2 \text{ and } v_4 \neq 0 \\
\{h\} & \text{if } v_1 \in \mathrm{dom}(h) \text{ and } h(v_1) \neq v_2 \text{ and } v_4 = 0 \\
\{\lightning\} & v_1 \notin \mathrm{dom}(h) \\
\emptyset & \text{otherwise.}
\end{cases} \\
[\![\mathsf{spawn}(\mathtt{f}, \vec{v})]\!] &\stackrel{\text{def}}{=} \lambda h.\, \{h\} \\
[\![\lightning]\!] &\stackrel{\text{def}}{=} \lambda h.\, \{\lightning\}
\end{aligned}
$$

For a heap h, the set of states $[\![\alpha]\!](h)$ is the set of possible outcomes of running the atomic command α. If the set is empty, then the command diverges. We lift non-deterministic state transformers to heaps or the faulting state by letting $[\![\alpha]\!](\lightning) = \{\lightning\}$.

Definition 4.26 (Program Operational Semantics)**.** The single step transition relation of a program

$$- \xrightarrow{-}_{-} - : \mathsf{State} \times \mathsf{AAction} \times \mathsf{FEnv} \times \mathsf{State}$$

is defined by the following rule:

$$\textsc{ProgramStep}\quad \frac{T \xrightarrow{\alpha}_{\eta} T' \qquad h' \in [\![\alpha]\!](h)}{(h, T) \rightarrow_{\eta} (h', T')}$$

The program operational semantics combines the execution of the thread pool and the updates performed by each thread on the heap, as described by the atomic commands α.

4.3 Assertion Language

We now present an assertion language for defining predicates. As well as standard separation logic connectives, the logic includes regions, guards, abstract predicates, atomicity tracking resources and view shifts.

Definition 4.27 (Logical Expressions)**.** Assume a set of *logic expressions* LExpr, ranged over by $e, e_1, \ldots$.

Definition 4.28 (Assertions)**.** The set of *assertions* Assn, ranged over by $P, Q, R, P_1, \ldots$ are defined

inductively as follows:

$$
\begin{array}{rcll}
P, Q, R & ::= & \mathsf{False} & \text{Falsehood} \\
& \mid & \mathsf{True} & \text{Truthfulness} \\
& \mid & P * Q & \text{Separating conjunction} \\
& \mid & x \mapsto y & \text{Heap cell} \\
& \mid & P \wedge Q & \text{Conjunction} \\
& \mid & P \vee Q & \text{Disjunction} \\
& \mid & \exists x.\, P & \text{Existential quantification} \\
& \mid & \forall x.\, P & \text{Universal quantification} \\
& \mid & P \implies Q & \text{Implication (material)} \\
& \mid & \mathbf{t}_a^\lambda(\vec{z}, x) & \text{Shared region } a\text{, of type } \mathbf{t}\text{, with level } \lambda\text{,} \\
& & & \text{parametrised by } \vec{z} \text{ and with abstract state } x \\
& \mid & I(\mathbf{t}_a^\lambda(\vec{z}, x)) & \text{Interpretation of a shared region } a\text{, of type } \mathbf{t}\text{, with level } \lambda\text{,} \\
& & & \text{parametrised by } \vec{z} \text{ and with abstract state } x \\
& \mid & [\mathrm{G}(\vec{z})]_a & \text{Guard G for region } a\text{, parametrised by } \vec{z} \\
& \mid & a \Mapsto \blacklozenge & \text{Atomicity tracking resource for region } a\text{,} \\
& & & \text{allowing atomic update} \\
& \mid & a \Mapsto \lozenge & \text{Atomicity tracking resource for region } a\text{,} \\
& & & \text{not allowing atomic update} \\
& \mid & a \Mapsto (x, y) & \text{Atomicity tracking resource for region } a\text{,} \\
& & & \text{witnessing an atomic update from } x \text{ to } y \\
& \mid & \mathsf{a}(\vec{z}) & \text{Abstract predicate } \mathsf{a} \text{ parametrised by } \vec{z}
\end{array}
$$

where

- $x, y \in \mathsf{LExpr}$ ranges over logical expressions and $\vec{z}$ denotes vectors of logical expressions.
- $a \in \mathsf{RId}$ ranges over region identifiers,
- $\lambda \in \mathsf{Level}$ ranges over levels,
- $\mathsf{a} \in \mathsf{APName}$ ranges over abstract predicates names.

4.4 Program Logic

The TaDA proof system is presented here for the programming language in §4.1.

Definition 4.29 (Function Specification Context)**.** Function specification contexts assign specifications to function symbols. They are generated by the following grammar, where $\vec{z}$ is a sequence of logical variables:

$$
\begin{array}{rcl}
\Gamma \in \mathsf{FunctionCtxt} & ::= & \epsilon \\
& \mid & \Gamma, \left(\lambda; \mathcal{A} \vdash \begin{array}{c} \mathbb{V} x \in X. \left\langle P_p(\vec{z}) \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathtt{f}(\vec{z}) \\ \exists\!\!\exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \end{array} \right)
\end{array}
$$

Definition 4.30 (Atomic Judgements)**.** The generalised form of atomic judgements in TaDA is:

$$\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

Here, P_p is the private precondition, $P(x)$ is the public precondition, $Q_p(x, y)$ is the private postcondition, and $Q(x, y)$ is the public postcondition. The private precondition is independent of x, since the environment can change x. The two parts of the postcondition are linked by y, which is chosen arbitrarily by the implementation when the atomic operation appears to take effect.

Formally, we only consider a single variable bound by $\mathbb{V}$ or $\exists\!\!\exists$. However, it is often useful to consider multiple such variables. We treat this as syntactic sugar which can be interpreted using Cartesian product and projection functions. We also drop the binder altogether when the variable ranges over the set $\mathbf{1}$.

We define $\Gamma; \lambda; \mathcal{A} \vdash \left\{P\right\} \; \mathbb{C} \; \left\{Q\right\}$ as sugar for $\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in \mathbf{1}. \left\langle P \,\middle|\, \mathsf{True} \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in \mathbf{1}. \left\langle Q \,\middle|\, \mathsf{True} \right\rangle$.

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

Definition 4.31 (Proof System)**.** The derivation rules for atomic judgements of the form

$$\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

are given in Figure 4.1, Figure 4.2 and Figure 4.3.

The proof rules in Figure 4.1 are related to the programming language.

The Skip rule asserts that the `skip;` does not change the state of the program. An alternative would be specify the pre- and postcondition as True.

The Sequencing rule allows us to compose programs that are executed sequentially.

The Loop rule states that there is a loop invariant, P_p, which is preserved by the loop body $\mathbb{C}$. After the loop is finished, the invariant P_p still holds, and moreover $\mathbb{B}$ must be caused the loop to end.

The Conditional rule states that if both branches of the conditional establish Q_p, then the postcondition of the whole satisfies Q_p.

The FunctionCall rule and Fork rule establish the conditions for the functions to be called. Essentially, in order to call a function, we have to have enough resources. In the case of the fork operation, the ownership of P_p is transferred to the newly created thread.

The Allocation rule allocates memory and returns the address on `x`.

The Assignment rule states that, after the assignment, the variable `x` has value $\mathbb{E}$ where each occurrence of `x` has been replaced by v. The Mutation rule and the CompareAndSet rule are similar, but manipulate the heap instead.

The main proof rules to handle abstract atomicity are shown in Figure 4.2. The first three rules allow us to access the content of a shared region by using an atomic command. With all of the rules, the update to the shared region must be atomic, so its interpretation is in the public part in the premiss. The region is in the public part in the conclusion also, but may be moved by applying atomicity weakening. The last rule is used to prove that a command is atomic.

Skip

$$\frac{}{\Gamma; \lambda; \mathcal{A} \vdash \{P_p\}\ \texttt{skip};\ \{P_p\}}$$

Sequencing

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \{P_p\}\ \mathbb{C}_1\ \{R_p\} \qquad \Gamma; \lambda; \mathcal{A} \vdash \{R_p\}\ \mathbb{C}_2\ \{Q_p\}}{\Gamma; \lambda; \mathcal{A} \vdash \{P_p\}\ \mathbb{C}_1\ \mathbb{C}_2\ \{Q_p\}}$$

Loop

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \{P_p * \mathbb{B}\}\ \mathbb{C}\ \{P_p\}}{\Gamma; \lambda; \mathcal{A} \vdash \{P_p\}\ \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}\ \{P_p * \neg\mathbb{B}\}}$$

Conditional

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \{P_p * \mathbb{B}\}\ \mathbb{C}_1\ \{Q_p\} \qquad \Gamma; \lambda; \mathcal{A} \vdash \{P_p * \neg\mathbb{B}\}\ \mathbb{C}_2\ \{Q_p\}}{\Gamma; \lambda; \mathcal{A} \vdash \{P_p\}\ \texttt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \texttt{else}\ \{\mathbb{C}_2\}\ \{Q_p\}}$$

FunctionCall

$$\frac{\left(\lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p(\vec{z}) \mid P(x, \vec{z}) \rangle\ \texttt{f}(\vec{z})\quad \exists\!\!\exists y \in Y. \langle Q_p(x, y, \vec{z}, \mathsf{ret}) \mid Q(x, y, \vec{z}, \mathsf{ret}) \rangle\right) \in \Gamma}{\Gamma; \lambda; \mathcal{A} \vdash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p(\vec{z}) * \vec{z} = \vec{\mathbb{E}} \,\middle|\, P(x, \vec{z}) \right\rangle \\ \texttt{x} := \texttt{f}(\vec{\mathbb{E}}); \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \langle Q_p(x, y, \vec{z}, r) * \texttt{x} = r \mid Q(x, y, \vec{z}, r) \rangle \end{array}}$$

Fork

$$\frac{\left(\lambda; \emptyset \vdash \mathbb{V}x \in \mathbf{1}. \langle P_p(\vec{z}) \mid \mathsf{True} \rangle\ \texttt{f}(\vec{z})\quad \exists\!\!\exists y \in \mathbf{1}. \langle Q_p(\vec{z}, \mathsf{ret}) \mid \mathsf{True} \rangle\right) \in \Gamma}{\Gamma; \lambda; \mathcal{A} \vdash \left\{P_p(\vec{z}) * \vec{z} = \vec{\mathbb{E}}\right\}\ \texttt{fork f}(\vec{\mathbb{E}});\ \left\{\mathsf{True}\right\}}$$

Allocation

$$\frac{}{\Gamma; \lambda; \mathcal{A} \vdash \{\mathbb{E} = v * v > 0\}\ \texttt{x} := \texttt{alloc}(\mathbb{E});\ \{\texttt{x} \mapsto _ * \ldots * (\texttt{x} + v - 1) \mapsto _\}}$$

Assignment

$$\frac{}{\Gamma; \lambda; \mathcal{A} \vdash \{\texttt{x} = v\}\ \texttt{x} := \mathbb{E};\ \{\texttt{x} = \mathbb{E}[v/\texttt{x}]\}}$$

Lookup

$$\frac{}{\Gamma; 0; \mathcal{A} \vdash \mathbb{V}v \in \mathsf{Val}. \langle z = \mathbb{E} \mid z \mapsto v \rangle\ \texttt{x} := [\mathbb{E}];\quad \exists\!\!\exists y \in \mathbf{1}. \langle \texttt{x} = v \mid z \mapsto v \rangle}$$

Mutation

$$\frac{}{\Gamma; 0; \mathcal{A} \vdash \mathbb{V}v \in \mathsf{Val}. \langle z_1 = \mathbb{E}_1 * z_2 = \mathbb{E}_2 \mid z_1 \mapsto v \rangle\ [\mathbb{E}_1] := \mathbb{E}_2;\quad \exists\!\!\exists y \in \mathbf{1}. \langle \mathsf{True} \mid z_1 \mapsto z_2 \rangle}$$

CompareAndSet

$$\frac{}{\Gamma; 0; \mathcal{A} \vdash \begin{array}{c} \mathbb{V}v_1 \in \mathsf{Val}. \langle z = \mathbb{E}_1 * v_2 = \mathbb{E}_2 * v_3 = \mathbb{E}_3 \mid z \mapsto v_1 \rangle \\ \texttt{x} := \texttt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3); \\ \exists\!\!\exists y \in \mathbf{1}. \langle (v_1 = v_2 * \texttt{x} \neq 0) \vee (v_1 \neq v_2 * \texttt{x} = 0) \mid (v_1 = v_2 * z \mapsto v_3) \vee (v_1 \neq v_2 * z \mapsto v_1) \rangle \end{array}}$$

Function

$$\frac{\begin{array}{c} \mathit{code}(\eta(\texttt{f})) = \mathbb{C}\ \texttt{return}\ \mathbb{E}; \qquad \mathit{vars}(\eta(\texttt{f})) = \vec{\texttt{x}} \\ \Gamma; \lambda; \mathcal{A} \vdash \begin{array}{c} \mathbb{V}x \in X. \langle P_p(\vec{z}) * \vec{\texttt{x}} = \vec{z} \mid P(x, \vec{z}) \rangle \\ \mathbb{C} \\ \exists\!\!\exists (y, \mathsf{ret}) \in Y \times \mathsf{Val}. \langle Q_p(x, y, \vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathbb{E} \mid Q(x, y, \vec{z}, \mathsf{ret}) \rangle \end{array} \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p(\vec{z}) \mid P(x, \vec{z}) \rangle\ \texttt{f}(\vec{z})\quad \exists\!\!\exists y \in Y. \langle Q_p(x, y, \vec{z}, \mathsf{ret}) \mid Q(x, y, \vec{z}, \mathsf{ret}) \rangle}$$

Figure 4.1: Programming language proof rules of the TaDA logic.

OpenRegion

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) \right\rangle}{\Gamma; \lambda + 1; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) \right\rangle}$$

UseAtomic

$$\frac{\begin{array}{c} a \notin \mathcal{A} \qquad \forall x \in X. \, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) \right\rangle \end{array}}{\Gamma; \lambda + 1; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) \right\rangle}$$

UpdateRegion

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \; \mathbb{C} \quad \exists\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y, w) \,\middle|\, \begin{array}{r} I(\mathbf{t}_a^\lambda(\vec{z}, w)) * Q_1(x, y, w) \\ \vee\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q_2(x, y) \end{array} \right\rangle}{\begin{array}{l} \Gamma; \lambda + 1; a : x \in X \rightsquigarrow W, \mathcal{A} \vdash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Mapsto \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y, w) \,\middle|\, \begin{array}{r} \mathbf{t}_a^\lambda(\vec{z}, w) * Q_1(x, y, w) * a \Mapsto (x, w) \\ \vee\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x, y) * a \Mapsto \blacklozenge \end{array} \right\rangle \end{array} \end{array}}$$

MakeAtomic

$$\frac{\begin{array}{c} a \notin \mathcal{A} \qquad \{(x, y) \mid x \in X, y \in Y\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \\ \Gamma; \lambda'; a : x \in X \rightsquigarrow Y, \mathcal{A} \vdash \left\{ P_p * \exists x \in X. \, \mathbf{t}_a^\lambda(\vec{z}, x) * a \Mapsto \blacklozenge \right\} \; \mathbb{C} \; \left\{ \exists x \in X, y \in Y. \, Q_p(x, y) * a \Mapsto (x, y) \right\} \end{array}}{\Gamma; \lambda'; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \right\rangle \; \mathbb{C} \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, y) * [\mathrm{G}]_a \right\rangle}$$

Figure 4.2: Atomicity proof rules of the TaDA logic.

The OpenRegion rule allows us to access the contents of a shared region without updating its abstract state. The command may change the concrete state of the region, so long as the abstract state is preserved.

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

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

update that is performed for use further down the proof tree.

It is necessary for the update region rule to account for both the case where the update occurs and where it does not. One might expect that the case with no update could be dealt with by the open region rule, and the results combined using a disjunction rule. However, a general disjunction rule is not sound for atomic triples. (If we have $\langle P_1 \rangle \, \mathbb{C} \, \langle Q \rangle$ and $\langle P_2 \rangle \, \mathbb{C} \, \langle Q \rangle$, we may not have $\langle P_1 \vee P_2 \rangle \, \mathbb{C} \, \langle Q \rangle$ since $\mathbb{C}$ might rely on the environment not changing between P_1 and P_2.) The proof of the atomic specification for the spin lock uses the conditional nature of the update region rule.

We revisit the MakeAtomic rule. As before, a guard in the conclusion must permit the update in accordance with the transition system for the region. This is replaced in the premiss by the atomicity context and atomicity tracking resource, which tracks the occurrence of the update. One difference is the inclusion of the private state, which is effectively preserved between the premiss and the conclusion. A second difference is the ∃∃-binding of the resulting state of the atomic update. This allows the private state to reflect the result of the update.

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

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

The last set of rules is given in Figure 4.3. In order to define the Frame rule, we introduce two auxiliary functions.

Definition 4.32 (Program Variables Sets)**.** The set of program variables in an assertion is denoted by pvars(P).

Definition 4.33 (Modified Sets)**.** The set of modified variables, $\mathrm{mods}(-) : \mathsf{Cmd} \to \mathcal{P}(\mathsf{Var})$, is defined

FRAME

$$\frac{\mathrm{mods}(\mathbb{C}) \cap \mathrm{pvars}(R') = \emptyset \qquad \mathrm{mods}(\mathbb{C}) \cap \mathrm{pvars}(R(x)) = \emptyset \qquad \Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle R' * P_p \mid R(x) * P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle R' * Q_p(x,y) \mid R(x) * Q(x,y) \rangle}$$

SUBSTITUTION

$$\frac{f : X' \to X \qquad g : Y' \to Y \qquad \Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y' \in Y'. \langle Q_p(x, g(y')) \mid Q(x, g(y')) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x' \in X'. \langle P_p \mid P(f(x')) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(f(x'), y) \mid Q(f(x'), y) \rangle}$$

CONSEQUENCE

$$\frac{\lambda; \mathcal{A} \vdash P_p \preceq P'_p \qquad \forall x \in X, y \in Y. \lambda; \mathcal{A} \vdash Q'_p(x,y) \preceq Q_p(x,y) \qquad \forall x \in X, y \in Y. \lambda; \mathcal{A} \vdash Q'(x,y) \preceq Q(x,y) \qquad \Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P'_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q'_p(x,y) \mid Q'(x,y) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}$$

AEXISTS

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}z \in \mathbf{1}. \langle P_p \mid \exists x \in X. P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists x \in X, y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}$$

AWEAKENING1

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P' * P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q'(x,y) * Q(x,y) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p * P' \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) * Q'(x,y) \mid Q(x,y) \rangle}$$

AWEAKENING2

$$\frac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}{\forall x \in X. \Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}z \in \mathbf{1}. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}$$

AWEAKENING3

$$\frac{\lambda' \leq \lambda \qquad \Gamma; \lambda'; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle\ \mathbb{C}\ \exists\!\!\exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle}$$

Figure 4.3: Auxiliary proof rules of the TaDA logic.

for a given command as follows:

$$\begin{aligned}
\mathrm{mods}(\texttt{skip};) &= \emptyset \\
\mathrm{mods}(\mathbb{C}_1\ \mathbb{C}_2) &= \mathrm{mods}(\mathbb{C}_1) \cup \mathrm{mods}(\mathbb{C}_2) \\
\mathrm{mods}(\texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}) &= \mathrm{mods}(\mathbb{C}) \\
\mathrm{mods}(\texttt{if}\ (\mathbb{B})\ \{\mathbb{C}_1\}\ \texttt{else}\ \{\mathbb{C}_2\}) &= \mathrm{mods}(\mathbb{C}_1) \cup \mathrm{mods}(\mathbb{C}_2) \\
\mathrm{mods}(\texttt{x} := \texttt{f}(\vec{\mathbb{E}});) &= \{\texttt{x}\} \\
\mathrm{mods}(\texttt{fork f}(\vec{\mathbb{E}});) &= \emptyset \\
\mathrm{mods}(\texttt{x} := \texttt{alloc}(\mathbb{E});) &= \{\texttt{x}\} \\
\mathrm{mods}(\texttt{x} := \mathbb{E};) &= \{\texttt{x}\} \\
\mathrm{mods}(\texttt{x} := [\mathbb{E}];) &= \{\texttt{x}\} \\
\mathrm{mods}([\mathbb{E}_1] := \mathbb{E}_2;) &= \emptyset \\
\mathrm{mods}(\texttt{x} := \texttt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3);) &= \{\texttt{x}\}
\end{aligned}$$

The functions are going to be used to guarantee that a command that modifies program variables does not invalidate the frame.

The Frame rule, as in separation logic, allows us to add the same resources to the pre- and postcondition, which are untouched by the command. Our frame rule separately adds to both the private and public parts. Note that the frame for the public part may be parametrised by the $\mathbb{V}$-bound variable x, additionally we require the frame to be stable as the pre- and postcondition assertions.

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

The Consequence rule allows to strengthen the precondition and weaken the postcondition, using view shifts which we define later in Definition 4.60.

The AExists allows us to eliminate existential quantifiers in the public state in the premiss of the conclusion.

The AWeakening1 rule allows us to convert private state from the conclusion into public state in the premiss. The first effectively implements a non-atomic specification with an atomic specification, dividing the assertions between the public and private parts. This allows us to forget about the (abstract) atomicity of an operation.

The AWeakening2 rule allow us to widen the interference from the conclusion to the premiss. This rule combined with AWeakening1 allows us make a implement a fully non-atomic specification with an atomic specification that potentially supports more interference.

The AWeakening3 rule allow us to implement a specification with a command that requires a lower level.

Previously in chapter 3 we have introduced simplified versions of the MakeAtomic, UpdateRegion and UseAtomic rules. We will show that they can be derived from the existing rules.

We start with the MakeAtomic rule, recall the simplified rule as follows:

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

Take $\mathcal{A} = \emptyset$, then $a \notin \mathcal{A}$ holds trivially. Take $P_p = \mathsf{True}$, $Q_p = \mathsf{True}$, and $Y = Q(x)$ and we have established that the simplified version follows from the MakeAtomic rule.

The simplified version of the UpdateRegion rule is as follows (renaming y as w):

$$\frac{\vdash \mathbb{V} x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) \right\rangle \mathbb{C} \left\langle \exists w \in Q(x).\, I(\mathbf{t}_a(\vec{z}, w)) * Q_1(x, w) \vee I(\mathbf{t}_a(\vec{z}, x)) * Q_2(x) \right\rangle}{a : x \in X \rightsquigarrow Q(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * a \Rightarrow \blacklozenge\right\} \mathbb{C} \left\{\begin{array}{c} \exists x \in X.\, (\exists w \in Q(x).\, Q_1(x, w) * \\ a \Rightarrow (x, w) \vee \mathbf{t}_a(\vec{z}, x) * Q_2(x) * a \Rightarrow \blacklozenge) \end{array}\right\}}$$

Starting from the UpdateRegion rule, take $\mathcal{A} = \emptyset$, $P_p = \mathsf{True}$, $Q_p = \mathsf{True}$, $Y = \mathbf{1}$, $W = Q(x)$, we have:

$$\Gamma; \lambda + 1; a : x \in X \rightsquigarrow Q(x) \vdash \begin{array}{c} \mathbb{V} x \in X. \left\langle \mathsf{True} \,\middle|\, \mathbf{t}^{\lambda}_a(\vec{z}, x) * P(x) * a \Rightarrow \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists w \in Q(x). \left\langle \mathsf{True} \,\middle|\, \begin{array}{c} \mathbf{t}^{\lambda}_a(\vec{z}, w) * Q_1(x, w) * a \Rightarrow (x, w) \\ \vee\, \mathbf{t}^{\lambda}_a(\vec{z}, x) * Q_2(x) * a \Rightarrow \blacklozenge \end{array} \right\rangle \end{array}$$

We apply the CONSEQUENCE rule to weaken the postcondition, forgetting about the region in the case there was an atomic update.

$$\Gamma; \lambda+1; a : x \in X \rightsquigarrow Q(x) \vdash \begin{array}{c} \mathbb{V}x \in X. \left\langle \mathsf{True} \middle| \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rightarrow \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists w \in Q(x). \left\langle \mathsf{True} \middle| \begin{array}{c} Q_1(x, w) * a \Rightarrow (x, w) \\ \vee\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x) * a \Rightarrow \blacklozenge \end{array} \right\rangle \end{array}$$

We apply the AEXISTS rule to weaken the abstract state of the region x and similarly the CONSEQUENCE rule for the postcondition we have:

$$\Gamma; \lambda+1; a : x \in X \rightsquigarrow Q(x) \vdash \begin{array}{c} \mathbb{V}z \in \mathbf{1}. \left\langle \mathsf{True} \middle| \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rightarrow \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists y \in Q(x). \left\langle \mathsf{True} \middle| \begin{array}{c} \exists x \in X.\, Q_1(x, y) * a \Rightarrow (x, y) \\ \vee\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x) * a \Rightarrow \blacklozenge \end{array} \right\rangle \end{array}$$

Finally, we apply the CONSEQUENCE rule to the postcondition to move the existencial quantification of $w \in Q(x)$ and AWEAKENING1 to move the whole public state to the private state, essentially converting the atomic triple into a normal Hoare triple.

Finally, the simplified USEATOMIC rule is as follows:

$$\frac{\forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad \vdash \mathbb{V}x \in X. \left\langle I(\mathbf{t}_a(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \right\rangle \, \mathbb{C} \, \left\langle I(\mathbf{t}_a(\vec{z}, f(x))) * Q(x) \right\rangle}{\vdash \left\{ \exists x \in X.\, \mathbf{t}_a(\vec{z}, x) * P(x) * [\mathrm{G}]_a \right\} \, \mathbb{C} \, \left\{ \exists x \in X.\, \mathbf{t}_a(\vec{z}, f(x)) * Q(x) \right\}}$$

Starting from the USEATOMIC rule, take $\mathcal{A} = \emptyset$, $P_p = \mathsf{True}$, $Q_p = \mathsf{True}$ and $Y = \mathbf{1}$ we have:

$$\Gamma; \lambda+1; \emptyset \vdash \mathbb{V}x \in X. \left\langle \mathsf{True} \middle| \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \right\rangle \, \mathbb{C} \quad \exists y \in \mathbf{1}. \left\langle \mathsf{True} \middle| \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x) \right\rangle$$

We apply the AEXISTS rule to weaken the abstract state of the region x and similarly the CONSEQUENCE rule for the postcondition we have:

$$\Gamma; \lambda+1; \emptyset \vdash \begin{array}{c} \mathbb{V}z \in \mathbf{1}. \left\langle \mathsf{True} \middle| \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \right\rangle \\ \mathbb{C} \\ \exists y \in \mathbf{1}. \left\langle \mathsf{True} \middle| \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x) \right\rangle \end{array}$$

Finally, we apply the AWEAKENING1 to move the whole public state to the private state, essentially converting the atomic triple into a normal Hoare triple.

4.5 Model

We now present the semantic model of the TaDA program logic, which we will use to define the semantics for the assertions and judgements in the logic.

Definition 4.34 (Variable Names)**.** Assume a set of *logic variable names* LVar, ranged over by $x, y, \ldots$.

Definition 4.35 (Guards and Guard Algebras)**.** Assume a set Guard that will contain all guards that we might wish to use, ranged over by $g, g_1, \ldots$ A *guard algebra* $\zeta = (\mathsf{G}, \bullet, \mathbf{0}, \mathbf{1})$ consists of:

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

where

$$g_1 \leq g_2 \overset{\text{def}}{\iff} \exists g_3.\, g_1 \bullet g_3 = g_2.$$

Let GAlg denote the set of all guard algebras, ranged ranged over by ζ and G_ζ denote the carrier set for guard algebra ζ.

Note that a guard algebra is a separation algebra (in the sense of [13]) with a single unit, $\mathbf{0}$.

Definition 4.36 (Abstract States and Transition Systems)**.** 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* $\mathcal{T} : \mathsf{G}_\zeta \to_{mon} \mathcal{P}(\mathsf{AState} \times \mathsf{AState})$ is a mapping from guards to relations between abstract states. Let $\mathcal{T}(g)^*$ denote the reflexive-transitive closure of the relation $\mathcal{T}(g)$ for $g \in \mathsf{G}_\zeta$. Let ASTS_ζ denote the set of all ζ-labelled transition systems.

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.

Definition 4.37 (Abstract Region Types)**.** 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.

Definition 4.38 (Abstract Predicates)**.** 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.

Definition 4.39 (Levels)**.** Assume a level $\lambda \in \mathsf{Level} \overset{\text{def}}{=} \mathbb{N}$. Levels are ordered by the usual well-founded ordering on natural numbers.

Remark 4.1 (On levels)**.** It would be possible to take the levels from a more general well-founded order. This might be useful if we need some kind of unbounded nesting of regions.

Definition 4.40 (Region Identifiers)**.** Assume a countably infinite set of region identifiers, RId.

Definition 4.41 (Region Assignments)**.** A region assignment $r \in \mathsf{RAss} \overset{\text{def}}{=} \mathsf{RId} \rightharpoonup_{\mathit{fin}} \mathsf{Level} \times \mathsf{RTName} \times \mathsf{Val}^*$ 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 $t \in \mathsf{ARType}$.

Definition 4.42 (Guard Assignments)**.** Given a region assignment r, a guard assignment

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

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

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

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

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

Definition 4.43 (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: $\rho_1 \leq \rho_2 \stackrel{\text{def}}{\iff} \forall a \in \mathrm{dom}(\rho_1).\, \rho_1(a) = \rho_2(a)$.

Definition 4.44 (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 $(\mathsf{World}, \bullet, \mathsf{emp})$ where

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

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

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

Definition 4.45 (World Predicates)**.** A world predicate $P \in \mathsf{WPred} \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathsf{World})$ is a set of worlds that is upwards closed with respect to the world ordering. That is, if $w \in P$ and $w \leq w'$ then $w' \in P$. This is also sometimes called intuitionistic interpretation.

The composition operator on worlds is lifted to world predicates:

$$P_1 * P_2 \stackrel{\text{def}}{=} \{w \mid \exists w_1 \in P_1, w_2 \in P_2.\, w = w_1 \bullet w_2\}$$

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

Remark 4.2 (Intuitionistic Interpretation)**.** An alternative approach would be to define world predicates using a classical interpretation. This would require changes in the proof rules, such as the MakeAtomic, to make sure that they do not forget regions.

Definition 4.46 (Worlds with Atomic Tracking)**.** The atomic tracking separation algebra is defined to be $((\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\}, \bullet, (\mathsf{AState} \times \mathsf{AState}) \cup \{\lozenge\})$, where $\bullet$ is defined by

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

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

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

By lifting $\bullet$ to maps, the maps form a separation algebra. Consequently, by combining the operators of its components, $\mathsf{AWorld}_{\mathcal{R}}$ is also an ordered separation algebra. We consider that $\mathsf{World} = \mathsf{AWorld}_{\emptyset}$. As with worlds, we consider predicates over worlds with atomic tracking $P \in \mathsf{AWPred}_{\mathcal{R}} \stackrel{\text{def}}{=} \mathcal{P}^{\uparrow}(\mathsf{AWorld}_{\mathcal{R}})$ to be upwards-closed sets. These predicates similarly have a $*$ operator.

Definition 4.47 (Atomicity Context)**.** An atomicity context $\mathcal{A} \in \mathsf{AContext} \stackrel{\text{def}}{=} \mathsf{RId} \rightharpoonup_{\mathit{fin}} \mathsf{AState} \rightharpoonup \mathcal{P}(\mathsf{AState})$ is a (finite) partial mapping from region identifiers to partial, non-deterministic abstract state transformers. In the context of proving that an operation is abstractly atomic, the atomicity context records the abstract operation to be performed. This has implications in terms of both how the thread performing the operation and the environment can update the region mentioned in the context.

Definition 4.48 (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:

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

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

Interference by the environment is abstracted by the rely relation.

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

The second rule expresses that having the ♦ 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.

Definition 4.49 (Stable Predicates)**.** Given an atomicity context $\mathcal{A} \in \mathsf{AContext}$, the stable predicates are those which are closed under the associated rely relation:

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

We call the stable predicates *views* (as in [13]) and denote the set of views (in atomicity context $\mathcal{A}$) by $\mathsf{View}_{\mathcal{A}}$. Note that if $\mathcal{A}'$ is an extension of $\mathcal{A}$ then $\mathsf{View}_{\mathcal{A}} \subseteq \mathsf{View}'_{\mathcal{A}}$. 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

$$\mathcal{A} \vDash P \text{ stable} \wedge \mathcal{A} \vDash Q \text{ stable} \implies \mathcal{A} \vDash P * Q \text{ stable}$$

Definition 4.50 (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 data structure 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. Here, we have purposely avoided having region interpretations directly referring to region interpretations. Impredicative CAP [54] 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.

Definition 4.51 (Abstract Predicate Interpretation)**.** An abstract predicate interpretation $\iota \in \mathsf{APInterp} \overset{\text{def}}{=} \mathsf{APName} \times \mathsf{Val}^* \to \mathsf{View}$ associates a view with each abstract predicate.

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

Definition 4.52 (Region Collapse)**.** Given a level $\lambda \in \mathsf{Level}$, the region collapse of a world $\varphi \in \mathsf{AWorld}_{\mathcal{R}'}$ is a set of worlds given by:

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

where r_φ and ρ_φ are projections of region assignments and region states respectively, from φ. This operation is lifted to predicates in a straightforward manner: $P{\downarrow_\lambda} \overset{\text{def}}{=} \bigcup_{\varphi \in P} \varphi{\downarrow_\lambda}$.

Definition 4.53 (Abstract Predicate Collapse)**.** The one-step abstract predicate collapse of a world is a set of worlds given by:

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

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

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

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

where b_φ is a projection of abstract predicate bags from φ.

Remark 4.3 (On the interpretation of abstract predicates)**.** 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 unfolding. If a predicate cannot be made fully concrete by finite unfolding, then its semantics will be False.

Definition 4.54 (Reification)**.** The reification operation on worlds collapses the regions and the abstract predicates, and then considers only the heap portion:

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

This operation is lifted to predicates in the usual manner.

Definition 4.55 (Guarantee Relation)**.** Given a level $\lambda \in \mathsf{Level}$, and atomicity context $\mathcal{A} \in \mathsf{AContext}$, the guarantee relation $\mathrm{G}_{\lambda;\mathcal{A}} \subseteq \mathsf{AWorld}_{\mathcal{R}'} \times \mathsf{AWorld}_{\mathcal{R}'}$ is defined as:

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

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.

Remark 4.4 (On reification)**.** 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!

4.5.1 Semantic Assertions

Logical variables in assertions are interpreted over the set of values Val and we assume that $\mathsf{RId} \subseteq \mathsf{Val}$, so that variables may range over region identifiers. In the semantics of assertions, free variables are evaluated using a *variable interpretation*, which maps variables to values.

Definition 4.56 (Variable Interpretations)**.** The set of variable Interpretations is $\mathsf{Interp} \stackrel{\text{def}}{=} \mathsf{LVar} \to \mathsf{Val}$, and is ranged over by $i, i_1, \ldots$

Definition 4.57 (Logical Expressions Semantics)**.** We assume an appropriate semantics for logical expressions

$$[\![-]\!]_{-} : \mathsf{LExpr} \times \mathsf{Interp} \to \mathsf{Val}$$

Definition 4.58 (Assertion Semantics)**.** The semantics of assertions

$$[\![-]\!]_{(-,-,-)} : \mathsf{Assn} \times \mathsf{Interp} \times \mathsf{RInterp} \times \mathsf{APInterp} \to \mathsf{View}_{\mathcal{A}}$$

is inductively defined as follows:

$$\begin{aligned}
[\![\mathsf{False}]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \emptyset \\
[\![\mathsf{True}]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \mathsf{AWorld} \\
[\![P \wedge Q]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \left\{\varphi \mid \varphi \in [\![P]\!]_{(i,I,\iota)} \text{ and } \varphi \in [\![Q]\!]_{(i,I,\iota)}\right\} \\
[\![P \vee Q]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \left\{\varphi \mid \varphi \in [\![P]\!]_{(i,I,\iota)} \text{ or } \varphi \in [\![Q]\!]_{(i,I,\iota)}\right\} \\
[\![\exists x.\, P]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \bigcup_{v \in \mathsf{Val}} [\![P]\!]_{(i[x \mapsto v],I,\iota)} \\
[\![\forall x.\, P]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \bigcap_{v \in \mathsf{Val}} [\![P]\!]_{(i[x \mapsto v],I,\iota)} \\
[\![P * Q]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \left\{\varphi \mid \text{there exists } \varphi_1 \in [\![P]\!]_{(i,I,\iota)}, \varphi_2 \in [\![Q]\!]_{(i,I,\iota)} \text{ s.t. } \varphi = \varphi_1 \bullet \varphi_2\right\} \\
[\![x \mapsto y]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid h_\varphi([\![x]\!]_i) = [\![y]\!]_i\} \\
[\![P \implies Q]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \left\{\varphi \mid \varphi \in [\![P]\!]_{(i,I,\iota)} \implies \varphi \in [\![Q]\!]_{(i,I,\iota)}\right\} \\
[\![\mathbf{t}_a^\lambda(\vec{z}, x)]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid r_\varphi([\![a]\!]_i) = ([\![\lambda]\!]_i, \mathbf{t}, [\![\vec{z}]\!]_i) \text{ and } \rho_\varphi([\![a]\!]_i) = [\![x]\!]_i\} \\
[\![I(\mathbf{t}_a^\lambda(\vec{z}, x))]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} I(\mathbf{t}_a^{[\![\lambda]\!]_i}([\![\vec{z}]\!]_i, [\![x]\!]_i)) \\
[\![[\mathrm{G}(\vec{z})]_a]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid \mathrm{G}([\![\vec{z}]\!]_i) \leq \gamma_\varphi([\![a]\!]_i)\} \\
[\![a \Mapsto \blacklozenge]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid d_\varphi([\![a]\!]_i) = \blacklozenge\} \\
[\![a \Mapsto \lozenge]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid d_\varphi([\![a]\!]_i) = \lozenge \text{ or } d_\varphi([\![a]\!]_i) = \blacklozenge\} \\
[\![a \Mapsto (x, y)]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \{\varphi \in \mathsf{AWorld} \mid d_\varphi([\![a]\!]_i) = ([\![x]\!]_i, [\![y]\!]_i)\} \\
[\![\mathsf{a}(\vec{z})]\!]_{(i,I,\iota)} &\stackrel{\text{def}}{=} \iota(\mathsf{a}([\![\vec{z}]\!]_i))
\end{aligned}$$

The False assertion is not satisfied by any world. The True assertion is satisfied by any world. Conjunction and disjunction have the usual semantics. The existencial quantifier is satisfied by all worlds in which the predicate P holds for some assignment of x. The universal quantifier is satisfied by all worlds in which the predicate P holds for all assignments of x. The $P * Q$ assertion is interpreted for worlds with atomicity tracking in a similar style as explained in Definition 4.45. The $x \mapsto y$ assertion is satisfied by any world that contains a heap cell with address x and value y. The $P \implies Q$ is satisfied by the usual first order logic semantics. The $\mathbf{t}_a^\lambda(\vec{z}, x)$ assertion is satisfied by any world that contains

a region with identifier a such that it's level, type, parameters and abstract state match the region assertion. The semantics of region interpretation is described in Definition 4.52. The $[\mathrm{G}(\vec{z})]_a$ assertion is satisfied by any world that contains guards for the region with identifier a that are at least G. The assertions relating to the atomicity tracking component are described in Definition 4.46. Finally, the abstract predicate semantics is described in Definition 4.53.

4.5.2 Semantic Judgements

In the Views Framework [13], 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.

Definition 4.59 (Primitive Atomic Satisfaction Judgement)**.** The primitive atomic satisfaction judgement $\lambda; \mathcal{A} \vDash \langle P \rangle\ \alpha\ \langle Q \rangle$, where $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, $\alpha \in \mathsf{AAction}$ and $P, Q \in \mathsf{View}_{\mathcal{A}}$, is defined as:

$$\lambda; \mathcal{A} \vDash \langle P \rangle\ \alpha\ \langle Q \rangle \overset{\text{def}}{\iff} \forall R \in \mathsf{View}_{\mathcal{A}}.\, \forall \varphi \in P * R.\, \forall h \in \lfloor \varphi \rfloor_{\lambda}.\, \forall h' \in [\![\alpha]\!](h).\quad \exists \varphi'.\, \varphi\ \mathrm{G}_{\lambda;\mathcal{A}}\ \varphi' \wedge h' \in \lfloor \varphi' \rfloor_{\lambda} \wedge \varphi' \in Q * R.$$

The primitive atomic satisfaction judgement incorporates two assertions: P, the precondition; and Q, the post condition which the atomic update will satisfy. Note that the action α changes the worlds according to the guarantee and level.

Definition 4.60 (View Shift)**.** We define view shifts as:

$$\lambda; \mathcal{A} \vDash P \preceq Q \overset{\text{def}}{\iff} \lambda; \mathcal{A} \vDash \langle P \rangle\ \mathsf{id}\ \langle Q \rangle$$

View shifts allows us to change views, such as allocating regions or weakening the type of a variable to a supertype, as long as the underlying program state remains the same.

For constructing a new region we can use the following view shift:

$$\textsc{RegionCreation}\ \frac{\mathrm{G} \in \mathcal{G}_{\mathbf{t}} \qquad \forall a.\, \lambda'; \mathcal{A} \vDash P * [\mathrm{G}]_a \preceq I(\mathbf{t}_a^{\lambda}(\vec{z}, x)) * Q(a)}{\lambda'; \mathcal{A} \vDash P \preceq \exists a.\, \mathbf{t}_a^{\lambda}(\vec{z}, x) * Q(a)}$$

The premiss states that if we pick a guard G, from the guard algebra for the region type we want to create, and combined with predicate P it implies the interpretation of the region for abstract state x. The predicate $Q(a)$ may contain guards from the newly created region. Then we can create a new region, which includes creating G as well.

View shifts can also be used to repartition where no concrete state changes. We use view shifts in the rule of consequence to move resources between regions. Finally, the view shift includes material implication and as such it subsumes the traditional one.

Definition 4.61 (Semantic Judgement)**.** The semantic judgement

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V} x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle\ c\quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle$$

where

- $\eta \in \mathsf{FEnv}$ is a function environment;
- $\lambda \in \mathsf{Level}$ 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;
- $P_p \in \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ 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;
- $P \in X \to \mathsf{View}_{\mathcal{A}}$ is the public part of the precondition, which may correspond to resources from some opened shared regions, and is parametrised by $x \in X$ that tracks the precondition at the atomic update and by the valuation of program variables;
- $c \in \mathsf{ExtCmd}$ is the program under consideration;
- $Q_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ is the private part of the postcondition, which is parametrised by $x \in X$ that tracks the precondition at the atomic update, by $y \in Y$ that tracks the postcondition at the atomic update, and by the valuation of program variables;
- $Q \in X \times Y \to \mathsf{View}_{\mathcal{A}}$ is the public part of the postcondition, which is similarly parametrised by $x \in X$ and $y \in Y$, and by the valuation of program variables;

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

- For all $\sigma, \sigma_1 \in \mathsf{Store}$, $c_1 \in \mathsf{ExtCmd}$, $\alpha \in \mathsf{AAction}$ with $(\sigma, c) \xrightarrow{\alpha}_{\eta} (\sigma_1, c_1)$, for all $x \in X$, there exists $P'_p \in \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$, $P''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ such that

$$\eta; \lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) * P(x) \right\rangle\ \alpha\ \left\langle P'_p(\sigma_1) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * Q(x, y) \right\rangle \tag{4.1}$$

$$\eta; \lambda; \mathcal{A} \vDash \forall x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle\ c_1\ \ \exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle, \tag{4.2}$$

$$\text{and for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\}\ c_1\ \left\{ Q_p(x, y) \right\}. \tag{4.3}$$

- For all $\sigma, \sigma_1 \in \mathsf{Store}$, $c_1 \in \mathsf{ExtCmd}$, $\mathtt{f}$, $\vec{v}$ with $(\sigma, c) \xrightarrow{\mathsf{spawn}(\mathtt{f}, \vec{v})}_{\eta} (\sigma_1, c_1)$, for all $x \in X$, there exist $P'_p \in \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$, $P''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ and $P_{\mathtt{f}} \in \mathsf{Store} \to \mathsf{View}$ such that for all $\sigma_{\mathtt{f}} \in \mathsf{Store}$ with $vars(\eta(\mathtt{f})) = \vec{\mathtt{x}}$, $\sigma_{\mathtt{f}}(\vec{\mathtt{x}}) = \vec{v}$ and $code(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E};$,

$$\lambda; \mathcal{A} \vDash P_p(\sigma) * P(x) \preceq P'_p(\sigma_1) * P_{\mathtt{f}}(\sigma_{\mathtt{f}}) * P(x) \vee \exists y \in Y.\, P''_p(\sigma_1) * P_{\mathtt{f}}(\sigma_{\mathtt{f}}) * Q(x, y), \tag{4.4}$$

$$\eta; \lambda; \mathcal{A} \vDash \forall x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle\ c_1\ \ \exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle, \tag{4.5}$$

$$\text{for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\}\ c_1\ \left\{ Q_p(x, y) \right\}, \tag{4.6}$$

$$\text{and } \eta; \lambda; \emptyset \vDash \begin{array}{c} \forall x \in \mathbf{1}. \left\langle \lambda \sigma_{\mathtt{f}}.\, P_{\mathtt{f}}(\vec{z}) * \sigma_{\mathtt{f}}(\vec{\mathtt{x}}) = \vec{z} \,\middle|\, \mathsf{True} \right\rangle \\ \mathbb{C} \\ \exists (y, \mathsf{ret}) \in \mathbf{1} \times \mathsf{Val}. \left\langle \lambda \sigma_{\mathtt{f}}.\, Q_{\mathtt{f}}(\vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}} \,\middle|\, \mathsf{True} \right\rangle \end{array}. \tag{4.7}$$

- If $c = \texttt{skip}$; then, for all $\sigma \in \mathsf{Store}$, $x \in X$, there exists $y \in Y$ such that

$$\lambda; \mathcal{A} \vDash P_p(\sigma) * P(x) \preceq Q_p(x, y, \sigma) * Q(x, y). \tag{4.8}$$

Here, we adopt the syntax $\eta; \lambda; \mathcal{A} \vDash \left\{ P_p \right\}\ c\ \left\{ Q_p \right\}$ as shorthand for

$$\eta; \lambda; \mathcal{A} \vDash \forall x \in \mathbf{1}. \left\langle P_p \,\middle|\, \mathsf{True} \right\rangle\ c\ \ \exists y \in \mathbf{1}. \left\langle Q_p \,\middle|\, \mathsf{True} \right\rangle.$$

The semantic judgement breaks down into three mutually-exclusive cases: two progressing and one terminating. The first case covers normal progress, where the thread performs some atomic action (possibly id) described by (4.1). The action may or may not perform the atomic update: the two new private views express the outcome of each case. In the case where the atomic update is not performed, the continuation (4.2) takes up this obligation. In the case where the atomic update is performed, the continuation (4.3) 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 (4.4), 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 (4.6) and the newly-forked thread (4.7). 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) enforced by (4.8).

Definition 4.62 (Function Environment and Function Context Agreement)**.** The function environment and function context agreement, given by

$$- \vDash - : \mathsf{FEnv} \times \mathsf{FunctionCtxt} \to \mathsf{Bool}$$

is defined inductively as follows:

$$\eta \vDash \epsilon \overset{\text{def}}{\iff} \mathsf{True}$$

$$\eta \vDash \Gamma, \left(\lambda; \mathcal{A} \vdash \begin{array}{c} \forall x \in X. \left\langle P_p(\vec{z}) \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathtt{f}(\vec{z}) \\ \exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \end{array} \right)$$

$$\overset{\text{def}}{\iff} \eta \vDash \Gamma \text{ and } \eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \forall x \in X. \left\langle \lambda\sigma.\, P_p(\vec{z}) * \sigma(\vec{\mathtt{x}}) = \vec{z} \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathbb{C} \\ \exists (y, \mathsf{ret}) \in Y \times \mathsf{Val}. \left\langle \lambda\sigma.\, Q_p(x, y, \vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathcal{E}[\![\mathbb{E}]\!]_\sigma \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \end{array}$$

where $\mathit{vars}(\eta(\mathtt{f})) = \vec{\mathtt{x}}$ and $\mathit{code}(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E};$

Definition 4.63 (Semantic Judgement with Function Context)**.**

$$\Gamma; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \middle| P(x) \right\rangle \; c \; \exists y \in Y. \left\langle Q_p(x,y) \middle| Q(x,y) \right\rangle$$

$$\overset{\text{def}}{\iff} \left(\forall \eta \in \mathsf{FEnv}.\, \eta \vDash \Gamma \implies \eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \middle| P(x) \right\rangle \\ c \\ \exists y \in Y. \left\langle Q_p(x,y) \middle| Q(x,y) \right\rangle \end{array} \right)$$

4.6 Soundness

We prove that the TaDA program logic is sound with respect to the operational semantics.

Theorem 4.1 (Soundness of Commands)**.** If

$$\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}\vec{x} \in X. \left\langle P_p \middle| P(\vec{x}) \right\rangle \; \mathbb{C} \; \exists \vec{y} \in Y. \left\langle Q_p(\vec{x},\vec{y}) \middle| Q(\vec{x},\vec{y}) \right\rangle$$

is provable in the logic, then

$$\Gamma; \lambda; \mathcal{A} \vDash \mathbb{V}\vec{x} \in X. \left\langle P_p \middle| P(\vec{x}) \right\rangle \; \mathbb{C} \; \exists \vec{y} \in Y. \left\langle Q_p(\vec{x},\vec{y}) \middle| Q(\vec{x},\vec{y}) \right\rangle$$

holds semantically.

To prove the Soundness of Commands theorem, we show that the proof rules hold semantically.

Lemma 4.2 (Skip Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p \right\} \; \texttt{skip;} \; \left\{ P_p \right\} \tag{4.9}$$

Proof. To establish (4.9), it is sufficient to establish the sub-condition (4.8):

$$\lambda; \mathcal{A} \vDash P_p(\sigma) \preceq P_p(\sigma) \tag{4.10}$$

The (4.10) sub-condition follows from the reflexivity of view shift. □

Lemma 4.3 (Sequencing Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, suppose that

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p \right\} \; \mathbb{C}_1 \; \left\{ R_p \right\} \tag{4.11}$$

and

$$\eta; \lambda; \mathcal{A} \vDash \left\{ R_p \right\} \; \mathbb{C}_2 \; \left\{ Q_p \right\} \tag{4.12}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p \right\} \; \mathbb{C}_1 \; \mathbb{C}_2 \; \left\{ Q_p \right\} \tag{4.13}$$

Proof. The proof is by coinduction. We have two cases two consider, when $\mathbb{C}_1 = \texttt{skip;}$ or otherwise.

Consider the first case where $\mathbb{C}_1 = \texttt{skip;}$. The only possible reduction is $(\sigma, \texttt{skip;}\ \mathbb{C}_2) \xrightarrow{\mathsf{id}}_\eta (\sigma, \mathbb{C}_2)$, and so the first case of the semantic judgement applies. We have to establish the following sub-conditions,

by finding a suitable P'_p and P''_p:

$$\lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) \right\rangle \ \mathsf{id} \ \left\langle P'_p(\sigma) \vee \exists y \in \mathbf{1}.\, P''_p(x, y, \sigma) \right\rangle \tag{4.14}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ \mathbb{C}_2 \ \left\{ Q_p \right\} \tag{4.15}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ \mathbb{C}_2 \ \left\{ Q_p \right\} \tag{4.16}$$

Take $P'_p = \lambda\sigma.\, R_p$ and $P''_p = \lambda\sigma.\, \mathsf{False}$, the (4.16) sub-condition becomes trivial, while the (4.15) follows from the assumption (4.12). The (4.14) sub-condition follows the assumption (4.11) and the definition of semantic judgment case (4.8) when $c = \texttt{skip};$.

Consider the second case where $\mathbb{C}_1$ is not a skip. It must be the case that $(\sigma, \mathbb{C}_1\ \mathbb{C}_2) \xrightarrow{\alpha}_\eta (\sigma_1, c_3\ \mathbb{C}_2)$, where $(\sigma, \mathbb{C}_1) \xrightarrow{\alpha}_\eta (\sigma_1, c_3)$ for some α and c_3. We consider cases on whether α is a spawn action.

If $\alpha \notin \{\mathsf{spawn}(\mathtt{f}, \vec{v}) \mid \mathtt{f} \in \mathsf{Fun}, \vec{v} \in \mathsf{Val}^*\}$, the first condition of the semantic judgement is the only one that may apply. We have to establish the following sub-conditions to show (4.13), by finding a suitable P'_p and P''_p:

$$\lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) \right\rangle \ \alpha \ \left\langle P'_p(\sigma) \vee \exists y \in \mathbf{1}.\, P''_p(x, y, \sigma_1) \right\rangle \tag{4.17}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ c_3\ \mathbb{C}_2 \ \left\{ Q_p \right\} \tag{4.18}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ c_3\ \mathbb{C}_2 \ \left\{ Q_p \right\} \tag{4.19}$$

From the assumption (4.11) and the definition of semantic judgement we have that:

$$\lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) \right\rangle \ \alpha \ \left\langle P'_p(\sigma) \vee \exists y \in \mathbf{1}.\, P''_p(x, y, \sigma_1) \right\rangle \tag{4.20}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ c_3 \ \left\{ R_p \right\} \tag{4.21}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ c_3 \ \left\{ R_p \right\} \tag{4.22}$$

The (4.17) sub-condition follows from (4.20). To establish (4.18), we have from (4.21)

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ c_3 \ \left\{ R_p \right\}$$

and from (4.12)

$$\eta; \lambda; \mathcal{A} \vDash \left\{ R_p \right\} \ \mathbb{C}_2 \ \left\{ Q_p \right\}$$

By the coinductive hypothesis, we have

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ c_3\ \mathbb{C}_2 \ \left\{ Q_p \right\}$$

To establish (4.19), we have from (4.22) for all $y \in \mathbf{1}$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ c_3 \ \left\{ R_p \right\}$$

and from (4.12)

$$\eta; \lambda; \mathcal{A} \vDash \left\{ R_p \right\} \ \mathbb{C}_2 \ \left\{ Q_p \right\}$$

By the coinductive hypothesis, we have

$$\eta;\lambda;\mathcal{A}\vDash\left\{P''_p(x,y)\right\}\ c_3\ \mathbb{C}_2\ \left\{Q_p\right\}$$

If $\alpha = \mathsf{spawn}(\mathtt{f},\vec{v})$, where $\mathtt{f}\in\mathsf{Fun}$ and $\vec{v}\in\mathsf{Val}^*$, the second condition of the semantic judgement is the only one that may apply. We have to establish the following sub-conditions to show (4.13), where $\sigma_{\mathtt{f}}\in\mathsf{Store}$ with $\mathit{vars}(\eta(\mathtt{f}))=\vec{\mathtt{x}}$, $\sigma_{\mathtt{f}}(\vec{\mathtt{x}})=\vec{v}$ and $\mathit{code}(\eta(\mathtt{f}))=\mathbb{C}\ \mathtt{return}\ \mathbb{E};$, by finding a suitable P'_p, $P_{\mathtt{f}}$ and P''_p:

$$\lambda;\mathcal{A}\vDash P_p(\sigma)\preceq P'_p(\sigma_1)*P_{\mathtt{f}}(\sigma_{\mathtt{f}})\vee\exists y\in\mathbf{1}.\,P''_p(\sigma_1)*P_{\mathtt{f}}(\sigma_{\mathtt{f}}) \tag{4.23}$$

$$\eta;\lambda;\mathcal{A}\vDash\left\{P'_p\right\}\ c_3\ \mathbb{C}_2\ \left\{Q_p\right\} \tag{4.24}$$

$$\text{for all } y\in\mathbf{1},\ \eta;\lambda;\mathcal{A}\vDash\left\{P''_p(x,y)\right\}\ c_3\ \mathbb{C}_2\ \left\{Q_p\right\} \tag{4.25}$$

$$\text{and } \eta;\lambda;\emptyset\vDash\begin{array}{c}\forall x\in\mathbf{1}.\left\langle\lambda\sigma_{\mathtt{f}}.\,P_{\mathtt{f}}(\vec{z})*\sigma_{\mathtt{f}}(\vec{\mathtt{x}})=\vec{z}\ \middle|\ \lambda\sigma_{\mathtt{f}}.\,\mathsf{True}\right\rangle\\ \mathbb{C}\\ \exists(y,\mathsf{ret})\in\mathbf{1}\times\mathsf{Val}.\left\langle\lambda\sigma_{\mathtt{f}}.\,Q_{\mathtt{f}}(\vec{z},\mathsf{ret})*\mathsf{ret}=\mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}}\ \middle|\ \lambda\sigma_{\mathtt{f}}.\,\mathsf{True}\right\rangle\end{array}. \tag{4.26}$$

From the assumption (4.11) and the definition of semantic judgement we have that:

$$\lambda;\mathcal{A}\vDash P_p(\sigma)\preceq P'_p(\sigma_1)*P_{\mathtt{f}}(\sigma_{\mathtt{f}})\vee\exists y\in\mathbf{1}.\,P''_p(\sigma_1)*P_{\mathtt{f}}(\sigma_{\mathtt{f}}) \tag{4.27}$$

$$\eta;\lambda;\mathcal{A}\vDash\left\{P'_p\right\}\ c_3\ \left\{R_p\right\} \tag{4.28}$$

$$\text{for all } y\in\mathbf{1},\ \eta;\lambda;\mathcal{A}\vDash\left\{P''_p(x,y)\right\}\ c_3\ \left\{R_p\right\} \tag{4.29}$$

$$\text{and } \eta;\lambda;\emptyset\vDash\begin{array}{c}\forall x\in\mathbf{1}.\left\langle\lambda\sigma_{\mathtt{f}}.\,P_{\mathtt{f}}(\vec{z})*\sigma_{\mathtt{f}}(\vec{\mathtt{x}})=\vec{z}\ \middle|\ \lambda\sigma_{\mathtt{f}}.\,\mathsf{True}\right\rangle\\ \mathbb{C}\\ \exists(y,\mathsf{ret})\in\mathbf{1}\times\mathsf{Val}.\left\langle\lambda\sigma_{\mathtt{f}}.\,Q_{\mathtt{f}}(\vec{z},\mathsf{ret})*\mathsf{ret}=\mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}}\ \middle|\ \lambda\sigma_{\mathtt{f}}.\,\mathsf{True}\right\rangle\end{array}. \tag{4.30}$$

The (4.23) sub-condition follows from (4.27) and the (4.26) sub-condition follows from (4.30). To establish (4.24), we have from (4.28)

$$\eta;\lambda;\mathcal{A}\vDash\left\{P'_p\right\}\ c_3\ \left\{R_p\right\}$$

and from (4.12)

$$\eta;\lambda;\mathcal{A}\vDash\left\{R_p\right\}\ \mathbb{C}_2\ \left\{Q_p\right\}$$

By the coinductive hypothesis, we have

$$\eta;\lambda;\mathcal{A}\vDash\left\{P'_p\right\}\ c_3\ \mathbb{C}_2\ \left\{Q_p\right\}$$

Finally, to establish (4.25), we have from (4.29) for all $y\in\mathbf{1}$

$$\eta;\lambda;\mathcal{A}\vDash\left\{P''_p(x,y)\right\}\ c_3\ \left\{R_p\right\}$$

and from (4.12)

$$\eta; \lambda; \mathcal{A} \vDash \left\{ R_p \right\} \mathbb{C}_2 \left\{ Q_p \right\}$$

By the coinductive hypothesis, we have

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p''(x, y) \right\} c_3 \ \mathbb{C}_2 \left\{ Q_p \right\}$$

□

Lemma 4.4 (Loop Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, suppose that

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p * \mathbb{B} \right\} \mathbb{C} \left\{ P_p \right\} \tag{4.31}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p \right\} \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.32}$$

Proof. Since $\texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\}$ has two possible reductions, both with transition id, to show (4.32), it is sufficient to establish:

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p * \mathbb{B} \right\} \mathbb{C} \ \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.33}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p * \neg\mathbb{B} \right\} \texttt{skip;} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.34}$$

This is sufficient since the first condition of the semantic judgement is the only one that may apply. For reduction $(\sigma, \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\}) \xrightarrow{\mathsf{id}}_\eta (\sigma, \mathbb{C} \ \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\})$ (which requires $\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \mathsf{True}$), we have to establish the following sub-conditions, by finding a suitable P_p' and P_p'':

$$\lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) \right\rangle \mathsf{id} \left\langle P_p'(\sigma) \vee \exists y \in \mathbf{1}.\, P_p''(x, y, \sigma) \right\rangle \tag{4.35}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p' \right\} \mathbb{C} \ \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.36}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P_p''(x, y) \right\} \mathbb{C} \ \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.37}$$

Take $P_p' = P_p * \mathbb{B}$ and $P_p'' = \lambda\sigma.\, \mathsf{False}$. The (4.35) and (4.37) sub-conditions become trivial, while the (4.36) reduces to (4.33). For the reduction $(\sigma, \texttt{while} \ (\mathbb{B}) \ \{\mathbb{C}\}) \xrightarrow{\mathsf{id}}_\eta (\sigma, \texttt{skip;})$ (which requires $\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \mathsf{False}$), we have to establish the following sub-conditions, by finding a suitable P_p' and P_p'':

$$\lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) \right\rangle \mathsf{id} \left\langle P_p'(\sigma) \vee \exists y \in \mathbf{1}.\, P_p''(x, y, \sigma) \right\rangle \tag{4.38}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p' \right\} \texttt{skip;} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.39}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P_p''(x, y) \right\} \texttt{skip;} \left\{ P_p * \neg\mathbb{B} \right\} \tag{4.40}$$

Take $P_p' = P_p * \neg\mathbb{B}$ and $P_p'' = \lambda\sigma.\, \mathsf{False}$. Similarly, the (4.38) and (4.40) sub-conditions are trivial and the (4.39) reduces to (4.34).

To establish (4.33), we have from (4.31)

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P_p * \mathbb{B} \right\} \mathbb{C} \left\{ P_p \right\}$$

By the coinductive hypothesis, we have

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p\right\} \texttt{while } (\mathbb{B}) \ \{\mathbb{C}\} \left\{P_p * \neg\mathbb{B}\right\}$$

Now (4.33) follows from the above by the SEQUENCING Rule Lemma.

It is trivial to establish (4.34) by applying the SKIP Rule Lemma. □

Lemma 4.5 (CONDITIONAL Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p * \mathbb{B}\right\} \mathbb{C}_1 \left\{Q_p\right\} \tag{4.41}$$

and

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p * \neg\mathbb{B}\right\} \mathbb{C}_2 \left\{Q_p\right\} \tag{4.42}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p\right\} \texttt{if } (\mathbb{B}) \ \{\mathbb{C}_1\} \texttt{ else } \{\mathbb{C}_2\} \left\{Q_p\right\} \tag{4.43}$$

Proof. Since $\texttt{if } (\mathbb{B}) \ \{\mathbb{C}_1\} \texttt{ else } \{\mathbb{C}_2\}$ has two possible reductions, both with transition id, the first condition of the semantic judgement is the only one that may apply. For reduction

$$(\sigma, \texttt{if } (\mathbb{B}) \ \{\mathbb{C}_1\} \texttt{ else } \{\mathbb{C}_2\}) \xrightarrow{\mathsf{id}}_\eta (\sigma, \mathbb{C}_1)$$

(which requires $\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \mathsf{True}$), we have to establish the following sub-conditions, by finding a suitable P_p' and P_p'':

$$\eta; \lambda; \mathcal{A} \vDash \left\langle P_p(\sigma)\right\rangle \mathsf{id} \left\langle P_p'(\sigma) \vee \exists y \in \mathbf{1}.\, P_p''(x, y, \sigma)\right\rangle \tag{4.44}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p'\right\} \mathbb{C}_1 \left\{Q_p\right\} \tag{4.45}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{P_p''(x, y)\right\} \mathbb{C}_1 \left\{Q_p\right\} \tag{4.46}$$

Take $P_p' = P_p * \mathbb{B}$ and $P_p'' = \lambda\sigma.\, \mathsf{False}$, the (4.44) and (4.46) sub-conditions become trivial, while the (4.45) follows from (4.41). For the reduction

$$(\sigma, \texttt{if } (\mathbb{B}) \ \{\mathbb{C}_1\} \texttt{ else } \{\mathbb{C}_2\}) \xrightarrow{\mathsf{id}}_\eta (\sigma, \mathbb{C}_2)$$

(which requires $\mathcal{B}[\![\mathbb{B}]\!]_\sigma = \mathsf{False}$), we have to establish the following sub-conditions, by finding a suitable P_p' and P_p'':

$$\eta; \lambda; \mathcal{A} \vDash \left\langle P_p(\sigma)\right\rangle \mathsf{id} \left\langle P_p'(\sigma) \vee \exists y \in \mathbf{1}.\, P_p''(x, y, \sigma)\right\rangle \tag{4.47}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{P_p'\right\} \mathbb{C}_2 \left\{Q_p\right\} \tag{4.48}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{P_p''(x, y)\right\} \mathbb{C}_2 \left\{Q_p\right\} \tag{4.49}$$

Take $P_p' = P_p * \neg\mathbb{B}$ and $P_p'' = \lambda\sigma.\, \mathsf{False}$. Similarly, the (4.47) and (4.49) sub-conditions are trivial and the (4.48) follows from (4.42). □

Lemma 4.6 (Stack Frame)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, suppose that

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \\ c \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle \lambda\sigma.\, Q_p(x, y, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_\sigma \,\middle|\, Q(x, y, r) \right\rangle \end{array} \tag{4.50}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \\ \mathtt{x} := (\sigma, c\ \mathtt{return}\ \mathbb{E};)\,; \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle Q_p(x, y, r) * \mathtt{x} = r \,\middle|\, Q(x, y, r) \right\rangle \end{array} \tag{4.51}$$

Proof. The proof is by coinduction. We have three cases to consider, one for each condition of the semantic judgment.

Consider the case where $c = \mathtt{skip};$. To establish (4.51) we have to show that:

$$\lambda; \mathcal{A} \vDash P_p * P(x) \preceq Q_p(x, y, r) * \mathtt{x} = R * Q(x, y, r) \tag{4.52}$$

From (4.50) and the third condition of the semantic judgement we have that:

$$\lambda; \mathcal{A} \vDash P_p * P(x) \preceq Q_p(x, y, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_\sigma * Q(x, y, r) \tag{4.53}$$

We are perform a view shift, corresponding to the id action

$$(\sigma_1, \mathtt{x} := (\sigma, \mathtt{skip;\ return}\ \mathbb{E};)\,;) \xrightarrow{\mathsf{id}}_\eta (\sigma_1[\mathtt{x} \mapsto \mathcal{E}[\![\mathbb{E}]\!]_\sigma], \mathtt{skip};) \tag{4.54}$$

To establish (4.52), we have $r = \mathcal{E}[\![\mathbb{E}]\!]_\sigma$ from (4.53) and $\mathtt{x} = \mathcal{E}[\![\mathbb{E}]\!]_\sigma$ from (4.54).

Consider the case where c is not skip and performs an action $\alpha \notin \{\mathsf{spawn}(\mathtt{f}, \vec{v}) \mid \mathtt{f} \in \mathsf{Fun}, \vec{v} \in \mathsf{Val}^*\}$, the first condition of the semantic judgement is the only one that may apply. For reduction $(\sigma_1, \mathtt{x} := (\sigma, c\ \mathtt{return}\ \mathbb{E};)\,;) \xrightarrow{\alpha}_\eta (\sigma_1, \mathtt{x} := (\sigma_2, c_2\ \mathtt{return}\ \mathbb{E};)\,;)$, where $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_2, c_2)$, we have to establish the following sub-conditions to show (4.51), by finding a suitable P_p' and P_p'':

$$\lambda; \mathcal{A} \vDash \left\langle P_p * P(x) \right\rangle\ \alpha\ \left\langle P_p' * P(x) \vee \exists (y, r) \in Y \times \mathsf{Val}.\, P_p''(x, y, r) * \mathtt{x} = R * Q(x, y, r) \right\rangle \tag{4.55}$$

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \\ \mathtt{x} := (\sigma_2, c_2\ \mathtt{return}\ \mathbb{E};)\,; \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle P_p'(x, y, r) \,\middle|\, Q(x, y, r) \right\rangle \end{array} \tag{4.56}$$

and for all $(y, r) \in Y \times \mathsf{Val}$, $\eta; \lambda; \mathcal{A} \vDash \left\{ P_p''(x, y, r) \right\}\ \mathtt{x} := (\sigma', c_2\ \mathtt{return}\ \mathbb{E};)\,;\ \left\{ Q_p(x, y, r) * \mathtt{x} = r \right\}$
(4.57)

We have from (4.50) and the first condition of the semantic judgement, for $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_2, c_2)$, we

have that:

$$\lambda; \mathcal{A} \vDash \left\langle P_p * P(x) \right\rangle \ \alpha \ \left\langle P'_p * P(x) \vee \exists (y, r) \in Y \times \mathsf{Val}.\, P''_p(x, y, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_\sigma * Q(x, y, r) \right\rangle \tag{4.58}$$

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \forall\!\!\forall x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \\ c_2 \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle P'_p(x, y, r) \,\middle|\, Q(x, y, r) \right\rangle \end{array} \tag{4.59}$$

$$\text{and for all } (y, r) \in Y \times \mathsf{Val},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y, r) \right\} \ c_2 \ \left\{ \lambda\sigma.\, Q_p(x, y, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_\sigma \right\} \tag{4.60}$$

Take P'_p and P''_p from (4.58), then we have established (4.55) sub-condition. We have (4.58), by the coinductive hypothesis, we have (4.56) sub-condition. Similarly, we have (4.60), by the coinductive hypothesis, we have (4.57) sub-condition.

Finally, the case where c performs a fork action follows the same reasoning. □

Lemma 4.7 (FunctionCall Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, $\Gamma \in \mathsf{FunctionCtxt}$, suppose that $\eta \vDash \Gamma$ and

$$\left(\lambda; \mathcal{A} \vdash \forall\!\!\forall x \in X. \left\langle P_p(\vec{z}) \,\middle|\, P(x, \vec{z}) \right\rangle \ \mathtt{f}(\vec{z}) \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \right) \in \Gamma \tag{4.61}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \forall\!\!\forall x \in X. \left\langle P_p(\vec{z}) * \vec{z} = \vec{\mathbb{E}} \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathtt{x} := \mathtt{f}(\vec{\mathbb{E}}); \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle Q_p(x, y, \vec{z}, r) * \mathtt{x} = r \,\middle|\, Q(x, y, \vec{z}, r) \right\rangle \end{array} \tag{4.62}$$

Proof. Since $\mathtt{x} := \mathtt{f}(\vec{\mathbb{E}});$ has only one possible reduction with transition id, the first condition of the semantic judgement is the only one that may apply. For reduction

$$\left(\sigma, \mathtt{x} := \mathtt{f}(\vec{\mathbb{E}});\right) \xrightarrow{\mathsf{id}}_\eta \left(\sigma, \mathtt{x} := \ (\sigma_\mathtt{f}, \mathit{code}(\eta(\mathtt{f})))\,;\right),$$

where $\mathcal{E}[\![\vec{\mathbb{E}}]\!]_\sigma = \sigma_\mathtt{f}(\mathit{vars}(\eta(\mathtt{f})))$, to show (4.62), we have to establish the following sub-conditions, where $\sigma_\mathtt{f} \in \mathsf{Store}$ with $\mathit{vars}(\eta(\mathtt{f})) = \vec{\mathtt{x}_\mathtt{f}}$, $\sigma_\mathtt{f}(\vec{\mathtt{x}}) = \vec{z}$ and $\mathit{code}(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E}_\mathtt{f};$, by finding a suitable P'_p and P''_p:

$$\eta; \lambda; \mathcal{A} \vDash \left\langle P_p(\vec{z}, \sigma) * P(x, \vec{z}) \right\rangle \ \mathsf{id} \ \left\langle P'_p(\vec{z}, \sigma) * P(x, \vec{z}) \vee \exists (y, r) \in Y \times \mathsf{Val}.\, P''_p(x, y, \vec{z}, \sigma) * Q(x, y, \vec{z}) \right\rangle \tag{4.63}$$

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \forall\!\!\forall x \in X. \left\langle P'_p(\vec{z}) \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathtt{x} := \ (\sigma_\mathtt{f}, \mathbb{C}\ \mathtt{return}\ \mathbb{E}_\mathtt{f};)\,; \\ \exists\!\!\exists (y, r) \in Y \times \mathsf{Val}. \left\langle Q_p(x, y, \vec{z}) * \mathtt{x} = r \,\middle|\, Q(x, y, \vec{z}) \right\rangle \end{array} \tag{4.64}$$

$$\text{and for all } (y, r) \in Y \times \mathsf{Val},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y, \vec{z}) \right\} \ \mathtt{x} := \ (\sigma_\mathtt{f}, \mathbb{C}\ \mathtt{return}\ \mathbb{E}_\mathtt{f};)\,; \ \left\{ Q_p(x, y, \vec{z}) * \mathtt{x} = r \right\} \tag{4.65}$$

Take $P'_p = \lambda\sigma.\, P_p$ and $P''_p = \lambda\sigma.\, \mathsf{False}$. Recall that P_p does not depend on the store σ. The (4.63) sub-condition follows from the reflexivity of view shift and (4.65) sub-condition is trivial.

To establish (4.64), we have from (4.61)

$$\left(\lambda; \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_p(\vec{z}) \,\middle|\, P(x, \vec{z}) \right\rangle \ \mathtt{f}(\vec{z}) \quad \exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle\right) \in \Gamma$$

By definition (4.62), we have that

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle \lambda\sigma_{\mathtt{f}}. P_p(\vec{z}) * \sigma_{\mathtt{f}}(\vec{\mathtt{x}_\mathtt{f}}) = \vec{z} \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathbb{C} \\ \exists (y, r) \in Y \times \mathsf{Val}. \left\langle \lambda\sigma_{\mathtt{f}}. Q_p(x, y, \vec{z}, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}} \,\middle|\, Q(x, y, \vec{z}, r) \right\rangle \end{array} \tag{4.66}$$

where $vars(\eta(\mathtt{f})) = \vec{\mathtt{x}_\mathtt{f}}$ and $code(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E}_{\mathtt{f}};$.

The (4.64) sub-condition follows from Lemma (4.6), given we have (4.66). □

Lemma 4.8 (Fork Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, $\Gamma \in \mathsf{FunctionCtxt}$, suppose that $\eta \vDash \Gamma$ and

$$\left(\lambda; \emptyset \vdash \mathbb{V}x \in \mathbf{1}. \left\langle P_p(\vec{z}) \,\middle|\, \mathsf{True} \right\rangle \ \mathtt{f}(\vec{z}) \quad \exists y \in \mathbf{1}. \left\langle Q_p(\vec{z}, \mathsf{ret}) \,\middle|\, \mathsf{True} \right\rangle\right) \in \Gamma \tag{4.67}$$

then

$$\Gamma; \lambda; \mathcal{A} \vDash \left\{ P_p(\vec{z}) * \vec{z} = \vec{\mathbb{E}} \right\} \ \mathtt{fork}\ \mathtt{f}(\vec{\mathbb{E}});\ \left\{ \mathsf{True} \right\} \tag{4.68}$$

Proof. Since $\mathtt{fork}\ \mathtt{f}(\vec{\mathbb{E}});$ has only one possible reduction with transition $\alpha = \mathsf{spawn}(\mathtt{f}, \vec{v})$, where $\mathtt{f} \in \mathsf{Fun}$ and $\vec{v} \in \mathsf{Val}^*$, the second condition of the semantic judgement is the only one that may apply. For reduction $\left(\sigma, \mathtt{fork}\ \mathtt{f}(\vec{\mathbb{E}});\right) \xrightarrow{\mathsf{spawn}(\mathtt{f},\vec{v})}_{\eta} (\sigma, \mathtt{skip};)$, where $\vec{v} = \mathcal{E}[\![\vec{\mathbb{E}}]\!]_{\sigma}$, to show (4.68), we have to establish the following sub-conditions, where $\sigma_{\mathtt{f}} \in \mathsf{Store}$ with $vars(\eta(\mathtt{f})) = \vec{\mathtt{x}}$, $\sigma_{\mathtt{f}}(\vec{\mathtt{x}}) = \vec{v}$ and $code(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E};$, by finding a suitable P'_p, $P_{\mathtt{f}}$ and P''_p and $Q_{\mathtt{f}}$:

$$\lambda; \mathcal{A} \vDash P_p(\vec{z}, \sigma) \preceq P'_p(\sigma_1) * P_{\mathtt{f}}(\sigma_{\mathtt{f}}) \vee \exists y \in \mathbf{1}. P''_p(\sigma_1) * P_{\mathtt{f}}(\sigma_{\mathtt{f}}) \tag{4.69}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\} \ \mathtt{skip};\ \left\{ \mathsf{True} \right\} \tag{4.70}$$

$$\text{for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ \mathtt{skip};\ \left\{ \mathsf{True} \right\} \tag{4.71}$$

$$\text{and } \eta; \lambda; \emptyset \vDash \begin{array}{c} \mathbb{V}x \in \mathbf{1}. \left\langle \lambda\sigma_{\mathtt{f}}. P_{\mathtt{f}}(\vec{z}) * \sigma_{\mathtt{f}}(\vec{\mathtt{x}}) = \vec{z} \,\middle|\, \mathsf{True} \right\rangle \\ \mathbb{C} \\ \exists (y, r) \in \mathbf{1} \times \mathsf{Val}. \left\langle \lambda\sigma_{\mathtt{f}}. Q_{\mathtt{f}}(\vec{z}, r) * r = \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}} \,\middle|\, \mathsf{True} \right\rangle \end{array}. \tag{4.72}$$

Take $P'_p = \lambda\sigma. \mathsf{True}$, $P_{\mathtt{f}} = \lambda\sigma_{\mathtt{f}}. P_p(\vec{z})$, $P''_p = \lambda\sigma. \mathsf{True}$ and $Q_{\mathtt{f}} = \lambda\sigma_{\mathtt{f}}. Q_p(\vec{z}, r)$, the (4.70) sub-condition follows from the reflexivity of view shift. The (4.70) and (4.71) sub-conditions follow directly from the Skip Rule Lemma.

To establish (4.72), we have from (4.67)

$$\left(\lambda; \emptyset \vdash \mathbb{V}x \in \mathbf{1}. \left\langle P_p(\vec{z}) \,\middle|\, \mathsf{True} \right\rangle \ \mathtt{f}(\vec{z}) \quad \exists y \in \mathbf{1}. \left\langle Q_p(\vec{z}, \mathsf{ret}) \,\middle|\, \mathsf{True} \right\rangle\right) \in \Gamma$$

By definition (4.62), we have that

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \forall\!\!\!\forall x \in X. \left\langle \lambda\sigma_{\mathtt{f}}.\, P_p(\vec{z}) * \sigma_{\mathtt{f}}(\vec{\mathtt{x}}) = \vec{z} \,\middle|\, \mathsf{True} \right\rangle \\ \mathbb{C} \\ \exists\!\!\!\exists (y, \mathsf{ret}) \in \mathbf{1} \times \mathsf{Val}. \left\langle \lambda\sigma_{\mathtt{f}}.\, Q_p(\vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_{\mathtt{f}}} \,\middle|\, \mathsf{True} \right\rangle \end{array} \tag{4.73}$$

where $vars(\eta(\mathtt{f})) = \vec{\mathtt{x}}$ and $code(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E};$. The (4.72) sub-condition follows directly from (4.73). □

Lemma 4.9 (Allocation Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \left\{ \mathbb{E} = w * w > 0 \right\}\ \mathtt{x} := \mathtt{alloc}(\mathbb{E});\ \left\{ \mathtt{x} \mapsto _ * \ldots * (\mathtt{x} + w - 1) \mapsto _ \right\} \tag{4.74}$$

Proof. Since $\mathtt{x} := \mathtt{alloc}(\mathbb{E});$ has only one possible reduction

$$(\sigma, \mathtt{x} := \mathtt{alloc}(\mathbb{E});) \xrightarrow{\mathsf{alloc}(\mathcal{E}[\![\mathbb{E}]\!]_\sigma, v)}_\eta (\sigma[\mathtt{x} \mapsto v], \mathtt{skip};)$$

To show (4.74), we have to establish the following sub-conditions, by finding a suitable P'_p and P''_p:

$$\eta; \lambda; \mathcal{A} \vDash \left\langle \mathbb{E} = w * w > 0 \right\rangle\ \mathsf{alloc}(\mathcal{E}[\![\mathbb{E}]\!]_\sigma, v)\ \left\langle P'_p(\sigma[\mathtt{x} \mapsto v]) \vee \exists y \in \mathbf{1}.\, P''_p(x, y, \sigma[\mathtt{x} \mapsto v]) \right\rangle \tag{4.75}$$

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P'_p \right\}\ \mathtt{skip};\ \left\{ \mathtt{x} \mapsto _ * \ldots * (\mathtt{x} + w - 1) \mapsto _ \right\} \tag{4.76}$$

$$\text{and for all } y \in \mathbf{1},\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\}\ \mathtt{skip};\ \left\{ \mathtt{x} \mapsto _ * \ldots * (\mathtt{x} + w - 1) \mapsto _ \right\} \tag{4.77}$$

Take $P'_p = \lambda\sigma.\, \mathtt{x} \mapsto _ * \ldots * (\mathtt{x} + w - 1) \mapsto _$ and $P''_p = \lambda\sigma.\, \mathsf{False}$, To establish (4.75), fix $R \in \mathsf{View}_{\mathcal{A}}$, $\varphi \in \mathbb{E} = w * w > 0 * R$, $h \in \lfloor \varphi \rfloor_\lambda$, $h' \in [\![\mathsf{alloc}(\mathcal{E}[\![\mathbb{E}]\!]_\sigma, v)]\!](h)$. Let

$$\varphi' = (r_\varphi, h', b_\varphi, \gamma_\varphi, \rho_\varphi, d_\varphi).$$

By the interpretation of atomic commands, given that $w > 0$, it must be that

$$h' = \emptyset[v \mapsto k_0, \ldots, v + w - 1 \mapsto k_{w-1}] \uplus h$$

and

$$\varphi' \in \mathtt{x} \mapsto _ * \ldots * (\mathtt{x} + w - 1) \mapsto _ * R$$

Finally, the (4.77) becomes trivial and (4.76) follows from the Skip Rule Lemma. □

Lemma 4.10 (Assignment Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \left\{ \mathtt{x} = v \right\}\ \mathtt{x} := \mathbb{E};\ \left\{ \mathtt{x} = \mathbb{E}[v/\mathtt{x}] \right\} \tag{4.78}$$

Proof. The proof follows directly from the operational semantics and is similar to that for the Allocation rule. □

Lemma 4.11 (Lookup Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; 0; \mathcal{A} \vDash \mathbb{V}v \in \mathsf{Val}.\left\langle z = \mathbb{E} \,\middle|\, z \mapsto v \right\rangle \; \mathtt{x} := [\mathbb{E}]; \quad \exists y \in \mathbf{1}.\left\langle \mathtt{x} = v \,\middle|\, z \mapsto v \right\rangle \tag{4.79}$$

Proof. The proof follows directly from the operational semantics and is similar to that for the Allocation rule. □

Lemma 4.12 (Mutation Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; 0; \mathcal{A} \vDash \mathbb{V}v \in \mathsf{Val}.\left\langle z_1 = \mathbb{E}_1 * z_2 = \mathbb{E}_2 \,\middle|\, z_1 \mapsto v \right\rangle \; [\mathbb{E}_1] := \mathbb{E}_2; \quad \exists y \in \mathbf{1}.\left\langle \mathsf{True} \,\middle|\, z_1 \mapsto z_2 \right\rangle$$

Proof. The proof follows directly from the operational semantics and is similar to that for the Allocation rule. □

Lemma 4.13 (CompareAndSet Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; 0; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}v_1 \in \mathsf{Val}.\left\langle z = \mathbb{E}_1 * v_2 = \mathbb{E}_2 * v_3 = \mathbb{E}_3 \,\middle|\, z \mapsto v_1 \right\rangle \\ \mathtt{x} := \mathtt{CAS}(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3); \\ \exists y \in \mathbf{1}.\left\langle (v_1 = v_2 * \mathtt{x} \neq 0) \vee (v_1 \neq v_2 * \mathtt{x} = 0) \,\middle|\, (v_1 = v_2 * z \mapsto v_3) \vee (v_1 \neq v_2 * z \mapsto v_1) \right\rangle \end{array}$$

Proof. The proof follows directly from the operational semantics and is similar to that for the Allocation rule. □

Lemma 4.14 (OpenRegion Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X.\left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \; c \quad \exists y \in Y.\left\langle Q_p(x, y) \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) \right\rangle \tag{4.80}$$

then

$$\eta; \lambda + 1; \mathcal{A} \vDash \mathbb{V}x \in X.\left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \right\rangle \; c \quad \exists y \in Y.\left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) \right\rangle \tag{4.81}$$

Proof. Consider the case where c is not skip and performs an action $\alpha \notin \{\mathsf{spawn}(\mathtt{f}, \vec{v}) \mid \mathtt{f} \in \mathsf{Fun}, \vec{v} \in \mathsf{Val}^*\}$, the first condition of the semantic judgement is the only one that may apply. For reduction $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_1, c_1)$, to show (4.81), we have to establish the following sub-conditions, by finding a suitable P'_p and P''_p:

$$\eta; \lambda + 1; \mathcal{A} \vDash \left\langle P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \right\rangle \; \alpha \; \left\langle \begin{array}{c} P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \vee \\ \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) \end{array} \right\rangle \tag{4.82}$$

$$\eta; \lambda + 1; \mathcal{A} \vDash \mathbb{V}x \in X.\left\langle P'_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \right\rangle \; c_1 \quad \exists y \in Y.\left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) \right\rangle \tag{4.83}$$

$$\text{and for all } y \in Y,\ \eta; \lambda + 1; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \; c_1 \; \left\{ Q_p(x, y) \right\} \tag{4.84}$$

By (4.80), there are P'_p and P''_p with:

$$\eta; \lambda; \mathcal{A} \vDash \left\langle P_p(\sigma) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \ \alpha \ \left\langle \begin{array}{c} P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \vee \\ \exists y \in Y.\, P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) \end{array} \right\rangle \tag{4.85}$$

$$\eta; \lambda; \mathcal{A} \vDash \forall x \in X. \left\langle P'_p \middle| I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \ c_1 \ \ \exists y \in Y. \left\langle Q_p(x, y) \middle| I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) \right\rangle \tag{4.86}$$

$$\text{and for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ c_1 \ \left\{ Q_p(x, y) \right\} \tag{4.87}$$

To establish (4.82), fix $x \in X$, $R \in \mathsf{View}_\mathcal{A}$, $\varphi \in P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * R$, $h \in \lfloor \varphi \rfloor_{\lambda+1}$, $h' \in [\![\alpha]\!](h)$. There will be some $\overline{\varphi} \in P_p(\sigma) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * R$, with $r_\varphi = r_{\overline{\varphi}}$ and $\rho_\varphi = \rho_{\overline{\varphi}}$, and $\lfloor \overline{\varphi} \rfloor_\lambda = \lfloor \varphi \rfloor_{\lambda+1}$, and so $h \in \lfloor \overline{\varphi} \rfloor_\lambda$. The $\overline{\varphi}$ corresponds to φ with all regions at level λ collapsed. (Recall that r_φ and ρ_φ are projections of region states and region assignments from φ.)

By (4.85), there is some $\overline{\varphi'}$ with $\overline{\varphi} \ \mathrm{G}_{\lambda;\mathcal{A}} \ \overline{\varphi'}$, $h' \in \lfloor \overline{\varphi'} \rfloor_\lambda$ and

$$\overline{\varphi'} \in \left(\ P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) \ \right) * R$$

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

- $\overline{\varphi'} \in P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * R$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in P'_p(\sigma_1) * P(x) * R$. Let

$$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}, d_{\varphi''}).$$

  Hence, by the guarantee, $\varphi' \in P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * R$, and by construction $\lfloor \varphi' \rfloor_{\lambda+1} = \lfloor \overline{\varphi'} \rfloor_\lambda$ and $\varphi \ \mathrm{G}_{\lambda+1;\mathcal{A}} \ \varphi'$.

- $\overline{\varphi'} \in P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q(x, y) * R$ for some $y \in Y$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in P''_p(x, y, \sigma_1) * Q(x, y) * R$. Let

$$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}, d_{\varphi''}).$$

  Hence, by the guarantee, $\varphi' \in P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) * R$, and by construction $\lfloor \varphi' \rfloor_{\lambda+1} = \lfloor \overline{\varphi'} \rfloor_\lambda$ and $\varphi \ \mathrm{G}_{\lambda+1;\mathcal{A}} \ \varphi'$.

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

$$\varphi' \in \Big(\; P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * Q(x, y) \; \Big) * R.$$

So we have established (4.82).

The (4.86) sub-condition follows from (4.86) and the coinductive hypothesis. The (4.84) sub-condition follows from (4.87) and the AWeakening3 Rule Lemma.

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

Lemma 4.15 (UseAtomic Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, $a \notin \mathcal{A}$ and $\forall x \in X.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathrm{G})^*$:

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \Big\langle P_p \;\Big|\; I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \Big\rangle \; c \quad \exists y \in Y. \Big\langle Q_p(x, y) \;\Big|\; I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) \Big\rangle \tag{4.88}$$

then

$$\eta; \lambda + 1; \mathcal{A} \vDash \mathbb{V}x \in X. \Big\langle P_p \;\Big|\; \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \Big\rangle \; c \quad \exists y \in Y. \Big\langle Q_p(x, y) \;\Big|\; \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) \Big\rangle \tag{4.89}$$

Proof. Consider the case where c is not skip and performs an action $\alpha \notin \{\mathsf{spawn}(\mathtt{f}, \vec{v}) \mid \mathtt{f} \in \mathsf{Fun}, \vec{v} \in \mathsf{Val}^*\}$, the first condition of the semantic judgement is the only one that may apply. For reduction $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_1, c_1)$, to show (4.89), we have to establish the following sub-conditions, by finding a suitable P'_p and P''_p:

$$\eta; \lambda + 1; \mathcal{A} \vDash \Big\langle P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \Big\rangle \; \alpha \; \Big\langle \begin{array}{c} P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \vee \\ \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) \end{array} \Big\rangle \tag{4.90}$$

$$\eta; \lambda + 1; \mathcal{A} \vDash \mathbb{V}x \in X. \Big\langle P'_p \;\Big|\; \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a \Big\rangle \; c_1 \quad \exists y \in Y. \Big\langle Q_p(x, y) \;\Big|\; \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) \Big\rangle \tag{4.91}$$

$$\text{and for all } y \in Y,\ \eta; \lambda + 1; \mathcal{A} \vDash \Big\{ P''_p(x, y) \Big\} \; c_1 \; \Big\{ Q_p(x, y) \Big\} \tag{4.92}$$

By (4.88), there are P'_p and P''_p with:

$$\eta; \lambda; \mathcal{A} \vDash \Big\langle P_p(\sigma) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \Big\rangle \; \alpha \; \Big\langle \begin{array}{c} P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \vee \\ \exists y \in Y.\, P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) \end{array} \Big\rangle \tag{4.93}$$

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \Big\langle P'_p \;\Big|\; I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \Big\rangle \; c_1 \quad \exists y \in Y. \Big\langle Q_p(x, y) \;\Big|\; I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) \Big\rangle \tag{4.94}$$

$$\text{and for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash \Big\{ P''_p(x, y) \Big\} \; c_1 \; \Big\{ Q_p(x, y) \Big\} \tag{4.95}$$

To establish (4.90), fix $x \in X$, $R \in \mathsf{View}_{\mathcal{A}}$, $\varphi \in P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a * R$, $h \in \lfloor \varphi \rfloor_{\lambda+1}$, $h' \in [\![\alpha]\!](h)$. There will be some $\overline{\varphi} \in P_p(\sigma) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a * R$, with $r_\varphi = r_{\overline{\varphi}}$ and $\rho_\varphi = \rho_{\overline{\varphi}}$,

and $\lfloor\overline{\varphi}\rfloor_\lambda = \lfloor\varphi\rfloor_{\lambda+1}$, and so $h \in \lfloor\overline{\varphi}\rfloor_\lambda$. By (4.93), there is some $\overline{\varphi'}$ with $\overline{\varphi}\ \mathrm{G}_{\lambda;\mathcal{A}}\ \overline{\varphi'}$, $h' \in \lfloor\overline{\varphi'}\rfloor_\lambda$ and

$$\overline{\varphi'} \in \Big(P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) \Big) * R$$

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

- $\overline{\varphi'} \in P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * [\mathrm{G}]_a * R$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in P'_p(\sigma_1) * P(x) * [\mathrm{G}]_a * R$. Let

$$\varphi' = (r_{\varphi''}, h_{\varphi''}, b_{\varphi''}, \gamma_{\varphi''}, \rho_{\varphi''}, d_{\varphi''}).$$

  Hence, by the guarantee, $\varphi' \in P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * [\mathrm{G}]_a * R$, and by construction $\lfloor\varphi'\rfloor_{\lambda+1} = \lfloor\overline{\varphi'}\rfloor_\lambda$ and $\varphi\ \mathrm{G}_{\lambda+1;\mathcal{A}}\ \varphi'$.

- $\overline{\varphi'} \in P''_p(x, y, \sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, f(x))) * Q(x, y) * R$ for some $y \in Y$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in P''_p(x, y, \sigma_1) * Q(x, y) * R$. Let

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

  Hence, by the guarantee, $\varphi' \in P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) * R$, and by construction $\lfloor\varphi'\rfloor_{\lambda+1} = \lfloor\overline{\varphi'}\rfloor_\lambda$ and $\varphi\ \mathrm{G}_{\lambda+1;\mathcal{A}}\ \varphi'$.

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

$$\varphi' \in \Big(P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, f(x)) * Q(x, y) \Big) * R.$$

So we have established (4.90).

The (4.94) sub-condition follows from (4.94) and the coinductive hypothesis. The (4.92) sub-condition follows from (4.95) and the AWeakening3 Rule Lemma.

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

Lemma 4.16 (UpdateRegion Rule)**.** Suppose that $a \notin \mathcal{A}$ and

$$\eta; \lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \\ \mathbb{C} \\ \exists\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y, w) \,\middle|\, \begin{array}{c} I(\mathbf{t}_a^\lambda(\vec{z}, w)) * Q_1(x, y, w) \\ \vee I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q_2(x, y) \end{array} \right\rangle \end{array} \tag{4.96}$$

Then

$$\eta; \lambda + 1; \mathcal{A}' \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Mapsto \blacklozenge \right\rangle \\ \mathbb{C} \\ \exists\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y, w) \,\middle|\, \begin{array}{c} \mathbf{t}_a^\lambda(\vec{z}, w) * Q_1(x, y, w) * a \Mapsto (x, w) \\ \vee \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x, y) * a \Mapsto \blacklozenge \end{array} \right\rangle \end{array} \tag{4.97}$$

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

Proof. Suppose that $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_1, c_1)$ with $\alpha \in \mathsf{AAction}$.

Fix $x \in X$. From our assumption (4.96), there are P_p' and $\overline{P_p''}$ with

$$\lambda; \mathcal{A} \vDash \begin{array}{c} \left\langle P_p(\sigma) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \\ \alpha \\ \left\langle \begin{array}{c} P_p'(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \vee \\ \exists (y, w) \in Y \times W. \overline{P_p''}(x, y, \sigma_1) * \left(I(\mathbf{t}_a^\lambda(\vec{z}, w)) * Q_1(x, y, w) \vee I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q_2(x, y) \right) \end{array} \right\rangle \end{array} \tag{4.98}$$

$$\lambda; \mathcal{A} \vDash \begin{array}{c} \mathbb{V}x \in X. \left\langle P_p' \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) \right\rangle \\ c_1 \\ \exists\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y) \,\middle|\, I(\mathbf{t}_a^\lambda(\vec{z}, w)) * Q_1(x, y, w) \vee I(\mathbf{t}_a^\lambda(\vec{z}, x)) * Q_2(x, y) \right\rangle \end{array} \tag{4.99}$$

$$\forall (y, w) \in Y \times W. \quad \lambda; \mathcal{A} \vDash \left\{ \overline{P_p''}(x, y) \right\} \; c_1 \; \left\{ Q_p(x, y) \right\} \tag{4.100}$$

We will show that these P_p' and $P_p''(x, y, w) = \overline{P_p''}(x, y, w) * a \Mapsto (x, w)$ work to establish our goal. Fix $R \in \mathsf{View}_{\mathcal{A}'}$, $\varphi \in P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Mapsto \blacklozenge * R$, $h \in \lfloor \varphi \rfloor_{\lambda+1}$, $h' \in [\![\alpha]\!](h)$.

Let $\overline{R} \in \mathsf{View}_{\mathcal{A}}$ be such that

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

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

There will be some $\overline{\varphi} \in P_p * I(\mathbf{t}_a^\lambda(\vec{z}, x)) * P(x) * \overline{R}$ with $r_\varphi = r_{\overline{\varphi}}$ and $\rho_\varphi = \rho_{\overline{\varphi}}$, and $\lfloor \overline{\varphi} \rfloor_\lambda = \lfloor \varphi \rfloor_{\lambda+1}$,

and so $h \in \lfloor\overline{\varphi}\rfloor_\lambda$. By (4.98), there is some $\overline{\varphi'}$ with $\overline{\varphi}\ \mathrm{G}_{\lambda;\mathcal{A}}\ \overline{\varphi'}$, $h' \in \lfloor\overline{\varphi'}\rfloor_\lambda$ and

$$\overline{\varphi'} \in \left(\begin{array}{c} P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z},x)) * P(x) \lor \\ \exists (y,w) \in Y \times W.\, \overline{P''_p}(x,y,w,\sigma_1) * \Big(I(\mathbf{t}_a^\lambda(\vec{z},w)) * Q_1(x,y,w) \lor I(\mathbf{t}_a^\lambda(\vec{z},x)) * Q_2(x,y) \Big) \end{array} \right) * \overline{R}$$

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

- $\overline{\varphi'} \in P'_p(\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z},x)) * P(x) * \overline{R}$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in P'_p(\sigma_1) * P(x) * R$. Let

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

  Hence, by the guarantee, $\varphi' \in P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z},x) * P(x) * R$, and by construction $\lfloor\varphi'\rfloor_{\lambda+1} = \lfloor\overline{\varphi'}\rfloor_\lambda$. Also $\varphi\ \mathrm{G}_{\lambda+1;\mathcal{A}'}\ \varphi'$.

- $\overline{\varphi'} \in \overline{P''_p}(x,y,w,\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z},w)) * Q_1(x,y,w) * \overline{R}$ for some $(y,w) \in Y \times W$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in \overline{P''_p}(x,y,w,\sigma_1) * Q_1(x,y,w) * R$. Let

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

  Hence, by the guarantee, $\varphi' \in P''_p(x,y,w,\sigma_1) * \mathbf{t}_a^\lambda(\vec{z},w) * Q_1(x,y,w) * R$, and by construction $\lfloor\varphi'\rfloor_{\lambda+1} = \lfloor\overline{\varphi'}\rfloor_\lambda$. Also $\varphi\ \mathrm{G}_{\lambda+1;\mathcal{A}'}\ \varphi'$.

- $\overline{\varphi'} \in \overline{P''_p}(x,y,w,\sigma_1) * I(\mathbf{t}_a^\lambda(\vec{z},x)) * Q_2(x,y) * \overline{R}$ for some $(y,w) \in Y \times W$. In this case, $\overline{\varphi'} = \varphi'' \bullet \overline{\overline{\varphi'}}$ where

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

  and $\varphi'' \in \overline{P''_p}(x,y,w,\sigma_1) * Q_2(x,y) * R$. Let

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

  Hence, by the guarantee, $\varphi' \in P''_p(x,y,w,\sigma_1) * \mathbf{t}_a^\lambda(\vec{z},x) * Q_2(x,y) * R$, and by construction $\lfloor\varphi'\rfloor_{\lambda+1} = \lfloor\overline{\varphi'}\rfloor_\lambda$. Also $\varphi\ \mathrm{G}_{\lambda+1;\mathcal{A}'}\ \varphi'$.

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

$$\varphi' \in \left(\begin{array}{c} P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rrightarrow \blacklozenge \vee \\ \exists (y, w) \in Y \times W.\, P''_p(x, y, w, \sigma_1) * \left(\begin{array}{c} \mathbf{t}_a^\lambda(\vec{z}, w) * Q_1(x, y, w) * a \Rrightarrow (x, w) \\ \vee \, \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x, y) * a \Rrightarrow \blacklozenge \end{array} \right) \end{array} \right) * R.$$

So we have established that

$$\lambda + 1; \mathcal{A}' \vDash \left\langle \begin{array}{c} \left\langle P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rrightarrow \blacklozenge \right\rangle \\ \alpha \\ P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rrightarrow \blacklozenge \vee \\ \exists (y, r) \in Y \times W.\, P''_p(x, y, w, \sigma_1) * \left(\begin{array}{c} \mathbf{t}_a^\lambda(\vec{z}, w) * Q_1(x, y, w) * a \Rrightarrow (x, w) \vee \\ \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x, y) * a \Rrightarrow \blacklozenge \end{array} \right) \end{array} \right\rangle.$$

By (4.99), it follows from the coinductive hypothesis that

$$\lambda + 1; \mathcal{A}' \vDash \begin{array}{c} \forall\!\!\!\forall x \in X. \left\langle P'_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * P(x) * a \Rrightarrow \blacklozenge \right\rangle \\ c \\ \exists\!\!\!\exists (y, w) \in Y \times W. \left\langle Q_p(x, y, w) \,\middle|\, \begin{array}{c} \mathbf{t}_a^\lambda(\vec{z}, w) * Q_1(x, y, w) * a \Rrightarrow (x, w) \vee \\ \mathbf{t}_a^\lambda(\vec{z}, x) * Q_2(x, y) * a \Rrightarrow \blacklozenge \end{array} \right\rangle \end{array}$$

Fix $y \in Y$ and $w \in W$. Because $P''_p(x, y, w) = \overline{P''_p}(x, y, w) * a \Rrightarrow (x, w)$, we can extend the atomicity context and have

$$\lambda + 1; \mathcal{A}' \vDash \left\{ P''_p(x, y, w) \right\} \; c_1 \; \left\{ Q_p(x, y, w) \right\}$$

□

Lemma 4.17 (Drop Context)**.** If, for $P \in \mathsf{View}_{\mathcal{A}}$, Q, $x \in X$, $y \in Y$

$$\eta; \lambda; a : x \in X \rightsquigarrow Y, \mathcal{A} \vDash \left\{ P * a \Rrightarrow (x, y) \right\} \; c \; \left\{ \exists x, y.\, Q(x, y) * a \Rrightarrow (x, y) \right\}$$

then

$$\eta; \lambda; \mathcal{A} \vDash \left\{ P \right\} \; c \; \left\{ Q(x, y) \right\}$$

Lemma 4.18 (MakeAtomic Rule)**.** Suppose that

$$\{(x, y) \mid x \in X, y \in Y\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^*$$

$$\eta; \lambda'; \mathcal{A} \vDash \left\{ P_p * \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * a \Rrightarrow \blacklozenge \right\} \; c \; \left\{ \exists x \in X, y \in Y.\, Q_p(x, y) * a \Rrightarrow (x, y) \right\}$$

where $\mathcal{A} = a : x \in X \rightsquigarrow Y, \mathcal{A}'$ and $a \notin \mathrm{dom}(\mathcal{A}')$. Then

$$\eta; \lambda'; \mathcal{A}' \vDash \forall\!\!\!\forall x \in X. \left\langle P_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \right\rangle \; c \quad \exists\!\!\!\exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, y) * [\mathrm{G}]_a \right\rangle$$

Proof. The proof is by coinduction. Consider the case where c performs an action. Suppose that $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_1, c_1)$ with $\alpha \in \mathsf{AAction}$. It is sufficient to show that, for all $x \in X$ there exist P'_p and P''_p such that

$$\eta; \lambda'; \mathcal{A} \vDash \langle P_p(\sigma) * P(x) \rangle \ \alpha \ \langle P'_p(\sigma_1) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * Q(x, y) \rangle \tag{4.101}$$

$$\eta; \lambda'; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle \ c_1 \quad \exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle, \tag{4.102}$$

$$\text{and for all } y \in Y,\ \eta; \lambda'; \mathcal{A} \vDash \left\{ P''_p(x, y) \right\} \ c_1 \ \left\{ Q_p(x, y) \right\}. \tag{4.103}$$

By the premiss, there must be some $\overline{P'_p}$ with

$$\lambda'; \mathcal{A} \vDash \left\langle P_p(\sigma) * \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * a \Mapsto \blacklozenge \right\rangle \ \alpha \ \left\langle \overline{P'_p}(\sigma_1) \right\rangle \tag{4.104}$$

$$\lambda'; \mathcal{A} \vDash \left\{ \overline{P'_p} \right\} \ c_1 \ \left\{ \exists x \in X, y \in Y.\, Q_p(x, y) * a \Mapsto (x, y) \right\}. \tag{4.105}$$

Fix $x \in X$. Fix $R \in \mathsf{View}_{\mathcal{A}'}$. Fix $\varphi \in P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a * R$.

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

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

Let $\overline{R} = R * [\mathrm{G}]_a * a \Mapsto -$. ($\overline{R}$ is stable with respect to $\mathcal{A}$ since the additional interference will be $a : x \in X \rightsquigarrow Y$, and the subset of $\overline{R}$ that is compatible with $[\mathrm{G}]_a$ must be closed under this.) Let $\overline{\varphi} = \varphi \bullet a \Mapsto \blacklozenge$. By construction, $\lfloor \varphi \rfloor_{\lambda'} = \lfloor \overline{\varphi} \rfloor_{\lambda'}$. We have that $\overline{\varphi} \in (P_p(\sigma) * \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * a \Mapsto \blacklozenge) * \overline{R}$.

By (4.104) there exists $\overline{\varphi'}$ with

$$\overline{\varphi}\ \mathrm{G}_{\lambda';\mathcal{A}}\ \overline{\varphi'}, \tag{4.106}$$

$$h' \in \lfloor \overline{w'} \rfloor_{\lambda'}, \tag{4.107}$$

$$\text{and } \overline{\varphi'} \in \overline{P'_p}(\sigma_1) * \overline{R}. \tag{4.108}$$

From (4.106) 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 $y \in Y$.

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

$$\varphi' \in P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, y)$$

since $\overline{\varphi'} \in \overline{P'_p}(\sigma_1) * \overline{R}$, by (4.108). By (4.106) 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 (4.107). Hence, we have established

$$\lambda'; \mathcal{A} \vDash \left\langle P_p(\sigma) * \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \right\rangle \ \alpha \ \left\langle P'_p(\sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * \mathbf{t}_a^\lambda(\vec{z}, y) \right\rangle.$$

We have that $P'_p * \exists x \in X.\, \mathbf{t}_a^\lambda(\vec{z}, x) * a \Mapsto \blacklozenge \vDash \overline{P'_p}$ and is stable with respect to $\mathcal{A}$. From (4.105), by left consequence and the coinductive hypothesis, we have

$$\lambda'; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P'_p \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, x) * [\mathrm{G}]_a \right\rangle \ c_1 \quad \exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, \mathbf{t}_a^\lambda(\vec{z}, y) * [\mathrm{G}]_a \right\rangle$$

Finally, from (4.105) and Lemma 4.17, we have, for all $y \in Y$

$$\lambda'; \mathcal{A}' \vDash \left\{ P''_p(x, y) \right\} \ c_1 \ \left\{ Q_p(x, y) \right\}.$$

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

Lemma 4.19 (Frame Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$, suppose $\mathrm{mods}(c) \cap \mathrm{pvars}(R') = \emptyset$, $\mathrm{mods}(c) \cap \mathrm{pvars}(R(x)) = \emptyset$ and

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle R' * P_p \,\middle|\, R(x) * P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle R' * Q_p(x,y) \,\middle|\, R(x) * Q(x,y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. ☐

Lemma 4.20 (Substitution Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$, suppose $f : X' \to X$, $g : Y' \to Y$ and

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; c \quad \exists y' \in Y'. \left\langle Q_p(x, g(y')) \,\middle|\, Q(x, g(y')) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x' \in X'. \left\langle P_p \,\middle|\, P(f(x')) \right\rangle \; c \quad \exists y \in Y. \left\langle Q_p(f(x'), y) \,\middle|\, Q(f(x'), y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. ☐

Lemma 4.21 (Consequence Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$, suppose $\lambda; \mathcal{A} \vDash P_p \preceq P'_p$, $\forall x \in X, y \in Y.\, \lambda; \mathcal{A} \vDash Q'_p(x,y) \preceq Q_p(x,y)$, $\forall x \in X, y \in Y.\, \lambda; \mathcal{A} \vDash Q'(x,y) \preceq Q(x,y)$ and

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle Q'_p(x,y) \,\middle|\, Q'(x,y) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule using the definition of view shift. ☐

Lemma 4.22 (AExists Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}z \in \mathbf{1}. \left\langle P_p \,\middle|\, \exists x \in X.\, P(x) \right\rangle \; c \quad \exists x \in X, y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q(x,y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. ☐

Lemma 4.23 (AWeakening1 Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V}x \in X. \left\langle P_p \,\middle|\, P' * P(x) \right\rangle \; c \quad \exists y \in Y. \left\langle Q_p(x,y) \,\middle|\, Q'(x,y) * Q(x,y) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V} x \in X. \left\langle P_p * P' \middle| P(x) \right\rangle \; c \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) * Q'(x, y) \middle| Q(x, y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. □

Lemma 4.24 (AWeakening2 Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda \in \mathsf{Level}$ and $\mathcal{A} \in \mathsf{AContext}$:

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V} x \in X. \left\langle P_p \middle| P(x) \right\rangle \; c \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \middle| Q(x, y) \right\rangle$$

then

$$\forall x \in X.\, \eta; \lambda; \mathcal{A} \vdash \mathbb{V} z \in \mathbf{1}. \left\langle P_p \middle| P(x) \right\rangle \; c \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \middle| Q(x, y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. □

Lemma 4.25 (AWeakening3 Rule)**.** For all $\eta \in \mathsf{FEnv}$, $\lambda, \lambda' \in \mathsf{Level}$, $\mathcal{A} \in \mathsf{AContext}$, suppose $\lambda' \leq \lambda$ and:

$$\eta; \lambda'; \mathcal{A} \vDash \mathbb{V} x \in X. \left\langle P_p \middle| P(x) \right\rangle \; c \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \middle| Q(x, y) \right\rangle$$

then

$$\eta; \lambda; \mathcal{A} \vDash \mathbb{V} x \in X. \left\langle P_p \middle| P(x) \right\rangle \; c \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y) \middle| Q(x, y) \right\rangle$$

Proof. The proof follows directly from the definition of semantic judgement and follows a similar reasoning to that for the OpenRegion rule. □

5 Using TaDA

In this chapter, we use the TaDA logic on a series of examples. We first specify an atomic lock module (§5.1). From this specification, we then derive the CAP-style lock specification given in §3.2, which illustrates the weakening of the atomic specification to a specific use case. We also prove that a spin lock implementation satisfies the atomic lock specification. We show how the logic supports vertical reasoning about modules, by verifying an implementation of multiple-compare-and-set (MCAS) using the lock specification (§5.2), and an implementation of a concurrent double-ended queue (deque) using the MCAS specification (§5.3). Moreover, we demonstrate how to apply the logic to fit the different protocols required by each concurrent module.

The examples demonstrate that TaDA combines the benefits of abstract atomicity and abstract disjointness within a single program logic. Further examples can be found in [64] where we use TaDA to prove a concurrent skiplist map implementation and various clients.

5.1 Spin Lock

We define a lock module with the operations `acquire` and `release` and a constructor `makeLock`, similarly to the one shown in §3.2.

5.1.1 Atomic Specification

The lock operations are specified using abstract predicates that represent the state of a lock: $\mathsf{Lock}(s, x, v)$ asserts the existence of a lock, addressed by x, that is in the unlocked state when $v = 0$ and is in the locked state when $v = 1$. These predicates confer ownership of the lock: it is not possible to have more than one $\mathsf{Lock}(s, x, v)$ for the same value of x. This contrasts with the style of specification given with CAP [12] shown in §3.2.1, but we shall see how the CAP specification can be derived using the atomic specification given here.

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

$$\lambda; \mathcal{A} \vdash \left\{\mathsf{True}\right\}\ \texttt{makeLock()}\ \left\{\exists s \in \mathbb{T}_1.\, \mathsf{Lock}(s, \mathsf{ret}, 0)\right\}$$

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

In the context of the specification, λ is a constant number that indicates the *level* of the specification. To a client, λ is abstract (much like $\mathsf{Lock}(s, x, v)$ is); different implementations may fix different values for λ. The purpose of the level in our specifications is to prevent certain unsound circularities. This is achieved by requiring the level to be reduced at each step where such a circularity could be introduced.

The specification for the `release` operation uses an *atomic* triple:

$$\lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle \ \mathtt{release}(\mathtt{x}) \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \right\rangle$$

Intuitively, this specification means that `release` will atomically take the lock `x` from the locked to unlocked state. The atomic triple makes a strong guarantee: as long as the concurrent environment guarantees that the (possibly) shared resource $\mathsf{Lock}(s, \mathtt{x}, 1)$ is available, the `release` operation will preserve $\mathsf{Lock}(s, \mathtt{x}, 1)$ until it transforms it into $\mathsf{Lock}(s, \mathtt{x}, 0)$; after the transformation, the operation no longer requires $\mathsf{Lock}(s, \mathtt{x}, 0)$, and is consequently oblivious to subsequent transformations by the environment (such as another thread acquiring the lock).

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

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

$$\lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \vee \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle \ \mathtt{acquire}(\mathtt{x}) \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle$$

This specification has two significant flaws. Firstly, it allows `acquire` to do nothing at all when the lock is already locked. This is contrary to what it should do, which is wait for it to become unlocked and then (atomically) lock it. Secondly, as the level of abstraction given by the precondition is $\mathsf{Lock}(s, \mathtt{x}, 0) \vee \mathsf{Lock}(s, \mathtt{x}, 1)$, an implementation could change the state of the lock arbitrarily *without appearing to have done anything.* In particular, an implementation could transition between the two states any number of times, so long as it is in the $\mathsf{Lock}(s, \mathtt{x}, 1)$ state when it finishes.

A second attempt to overcome these issues might be:

$$\lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle \ \mathtt{acquire}(\mathtt{x}) \ \left\langle \mathsf{False} \right\rangle$$
$$\lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \right\rangle \ \mathtt{acquire}(\mathtt{x}) \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle$$

In the left-hand triple, the lock is initially locked; the implementation may not terminate, nor change the state of the lock. In the right-hand triple, the lock is initially unlocked; the implementation may only make one atomic transformation from unlocked to locked. These specifications have a subtle flaw, in that they assume that the environment will not change the state of the lock. They prevent us from having multiple threads competing to acquire the lock, which is the essential purpose of a lock.

An equivalent specification makes use of a boolean logical variable:

$$\forall v \in \{0, 1\} \,.\, \lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, v) \right\rangle \ \mathtt{acquire}(\mathtt{x}) \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) * v = 0 \right\rangle$$

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

These specifications do not express the subtlety that the interference permitted before the atomic update is different for the environment and the operation. The environment should be allowed to change the value of v (i.e. acquire and release the lock) but the `lock` operation should not. The correct

specification expresses this by binding the variable v in a new way:

$$\lambda; \mathcal{A} \vdash \mathbb{V} v \in \{0,1\}. \left\langle \mathsf{Lock}(s, \mathtt{x}, v) \right\rangle \ \mathtt{acquire}(\mathtt{x}) \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) * v = 0 \right\rangle$$

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

5.1.2 CAP Specification

The atomic specification of the lock captures its essence as a synchronisation primitive. In practice, a lock is often used to protect some resource. We demonstrate how a CAP-style lock specification [12], which views the lock as a mechanism for protecting a resource invariant, can be derived from the atomic specification. This illustrates a typical use of a TaDA specification: first prove a strong abstract-atomic specification; then specialise to whatever is required by the client.

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

$$\lambda'; \mathcal{A} \vdash \left\{ \mathsf{True} \right\} \ \mathtt{makeLock}() \ \left\{ \mathsf{IsLock}(\mathsf{ret}) \right\}$$

$$\lambda'; \mathcal{A} \vdash \left\{ \mathsf{Locked}(\mathtt{x}) \right\} \ \mathtt{release}(\mathtt{x}) \ \left\{ \mathsf{True} \right\}$$

$$\lambda'; \mathcal{A} \vdash \left\{ \mathsf{IsLock}(\mathtt{x}) \right\} \ \mathtt{acquire}(\mathtt{x}) \ \left\{ \mathsf{IsLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x}) \right\}$$

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

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

To implement this specification, we must provide an interpretation for the abstract predicates IsLock and Locked. For this, we need to introduce a shared region. As in CAP, a shared region encapsulates some resource that is available to multiple threads. In our example, this resource will be the predicates $\mathsf{Lock}(s, x, v)$, plus some additional guard resource (described below). A shared region is associated with a protocol, which determines how its contents change over time. Following iCAP [54], the state of a shared region is abstracted, and protocols are expressed as transition systems over these abstract states. A thread may only change the abstract state of a region when it has the *guard* resource associated with the transition to be performed. An interpretation function associates each abstract state of a region with a concrete assertion. In summary, to specify a region we must supply the guards for the region, an abstract state transition system that is labelled by these guards, and a function interpreting abstract states as assertions.

For the **CAPLock**, we need only a very simple guard separation algebra: there is a single, indivisible guard named $\textsc{Unlock}$, as well as the empty guard $\mathbf{0}$. As a separation algebra, guard resources must have a partial composition operator that is associative and commutative. In this case, $\mathbf{0} \bullet x = x = x \bullet \mathbf{0}$ for all $x \in \{\mathbf{0}, \textsc{Unlock}\}$, and $\textsc{Unlock} \bullet \textsc{Unlock}$ is not defined.

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

$$\mathbf{0} : \ 0 \rightsquigarrow 1$$
$$\textsc{Unlock} : \ 1 \rightsquigarrow 0$$

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

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

$$I(\mathbf{CAPLock}_a^{\lambda'}(s, x, 0)) \triangleq \mathsf{Lock}(s, x, 0) * [\textsc{Unlock}]_a$$
$$I(\mathbf{CAPLock}_a^{\lambda'}(s, x, 1)) \triangleq \mathsf{Lock}(s, x, 1)$$

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

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

$$\begin{aligned} \mathsf{IsLock}(x) &\triangleq \exists a, s \in \mathbb{T}_1, v \in \{0,1\} \,.\, \mathbf{CAPLock}_a^{\lambda'}(s, x, v) \\ \mathsf{Locked}(x) &\triangleq \exists a, s \in \mathbb{T}_1 .\, \mathbf{CAPLock}_a^{\lambda'}(s, x, 1) * [\textsc{Unlock}]_a \end{aligned}$$

We also interpret the level λ' as $\lambda' \triangleq \lambda + 1$, where λ is the level of the atomic lock specification.

It remains to prove the specifications for the operations and the axioms.

The proofs of the `release` and `acquire` operations are given in Figure 5.1 and Figure 5.2. In the `release` proof, note that the immediate postcondition of the UseAtomic rule is not stable, since it is possible for the environment to acquire the lock. For illustrative purposes, we weaken it minimally to a stable assertion, although it could be weakened to True directly.

While a region is opened, it must not be re-opened. This is enforced by requiring the level in the context of the conclusion of UseAtomic to be higher than the level of the region being opened, which is also the level in the context of the premiss. We see this in the proof for `release` as $\lambda' = \lambda + 1$ is higher than the level of the **CAPLock** region, namely λ. The atomic specification for `release` indicates that it only opens regions below level λ. In particular, this assures us that it will not reopen the **CAPLock** region. Since the level constraints are typically straightforward, we shall omit them from subsequent proof outlines.

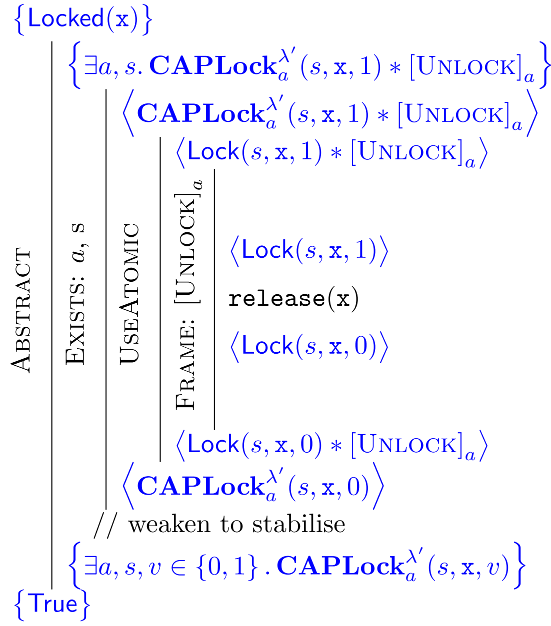


Figure 5.1: Derivation of the `release` specification.

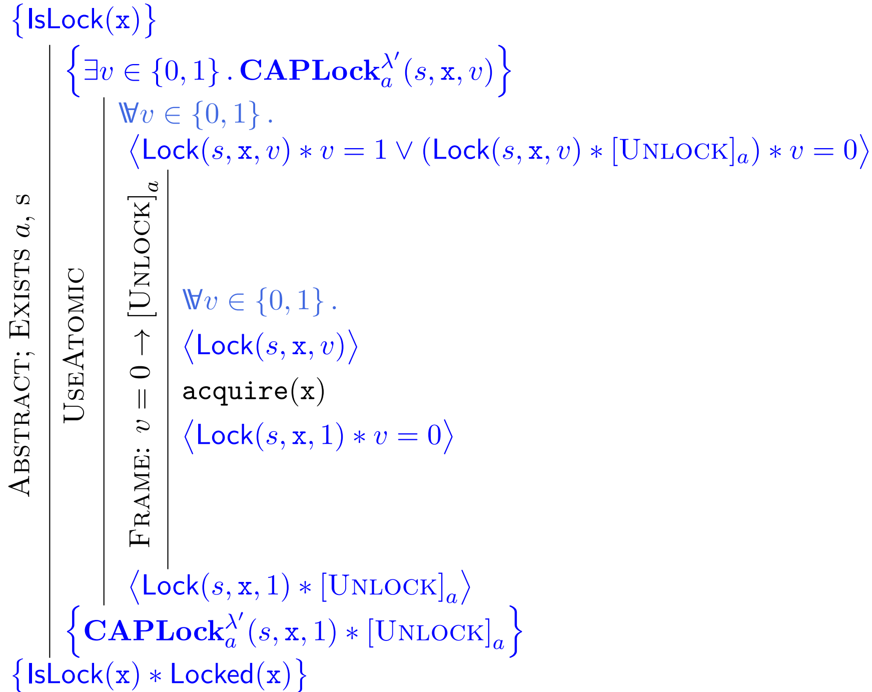


Figure 5.2: Derivation of the `acquire` specification.

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

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

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

Figure 5.3: Spin lock operations.

The `lock` proof uses the $\forall\!\!\!\forall$ quantifier in the premiss of the UseAtomic rule to account for the fact that, in the precondition, the lock could be in either state. The proof uses the frame rule, with a frame that is conditional on the state of the lock. To derive the final postcondition, we use the fact that region assertions, since they refer to shared resource, are freely duplicable: *i.e.* $\mathbf{CAPLock}_a^{\lambda'}(s, \mathtt{x}, 1) \equiv \mathbf{CAPLock}_a^{\lambda'}(s, \mathtt{x}, 1) * \mathbf{CAPLock}_a^{\lambda'}(s, \mathtt{x}, 1)$.

The axiom $\mathsf{IsLock}(x) \iff \mathsf{IsLock}(x) * \mathsf{IsLock}(x)$ follows from the duplicability of region assertions. Finally, the axiom $\mathsf{Locked}(x) * \mathsf{Locked}(x) \implies \mathsf{False}$ follows from Unlock $\bullet$ Unlock being undefined.

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

5.1.3 Implementation

We consider a spin lock implementation of the atomic lock specification. The code is given in Figure 5.3. To verify this implementation against the atomic specification, we must give a concrete interpretation of the abstract predicates. To do this, we introduce a new region type, **SLock**, with only one non-empty guard for a **SLock** region, named G, much as for **CAPLock**. There are two states for an **SLock** region: 0 and 1, representing unlocked and locked, respectively. The key difference from **CAPLock** is that transitions in both directions are guarded by G. The labelled transition system is as follows:

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

We also give an interpretation to each abstract state:

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

We now define the interpretation of the abstract predicate:

$$\mathsf{Lock}(a, x, v) \triangleq \mathbf{SLock}_a^0(x, v) * [\mathrm{G}]_a$$

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

To prove the implementation against our atomic specification, we use the MakeAtomic rule. The proof of the `acquire(x)` implementation is given in Figure 5.4. The proof first massages the specification into a form where we can apply the MakeAtomic rule. The atomicity context allows the region a to be in either state, but insists that it must have been in the unlocked state when the atomic operation takes effect. The UpdateRegion rule conditionally performs the atomic action—transitioning the region from state 0 to 1, and recording this in the atomic tracking resource—if the atomic compare-and-set operation succeeds. The MakeAtomic rule achieves this, by requiring that the implementation can only update the abstract region state once and also that the environment cannot invalidate the

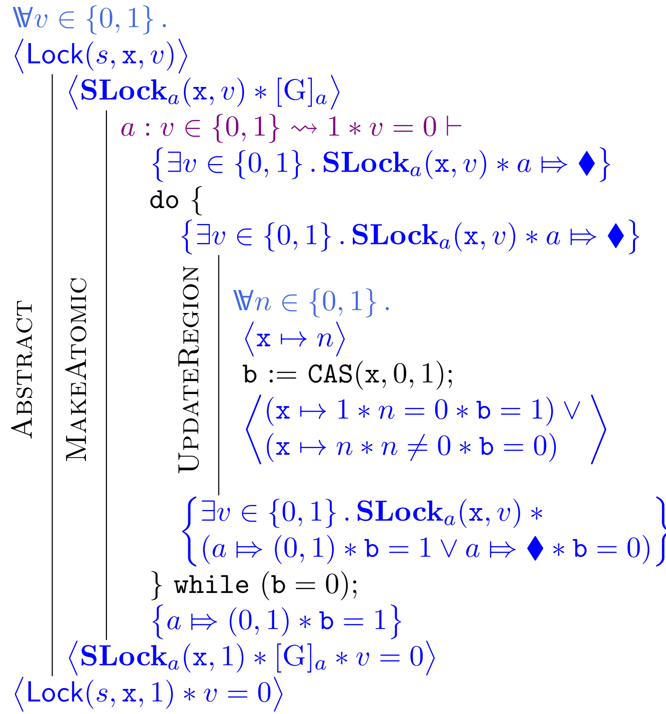


Figure 5.4: Proof outline for the `acquire` operation.

precondition for any $v \in \{0,1\}$, until the implementation performs the atomic update. This means that the environment is allows to change the abstract region state to any state, as long as it is 0 or 1.

The proofs for `makeLock` and `release` are similar, and as such, we omit them.

Remark 5.1 (On alternative implementations)**.** It is possible to prove the following alternative implementation of `release` with the same atomic specification:

$$\lambda; \mathcal{A} \vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle \ [\mathtt{x}] := 1; \ [\mathtt{x}] := 0; \ \left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \right\rangle$$

The first write to `x` has no effect, since the specification asserts that the lock must be locked initially. This code would clearly not be atomic in a different context; it would not satisfy the specification $\vdash \left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \lor \mathsf{Lock}(s, \mathtt{x}, 1) \right\rangle$ `release(x)` $\left\langle \mathsf{Lock}(s, \mathtt{x}, 0) \right\rangle$, for example. Since the specification constrains the client, it allows flexibility in the implementation. □

5.2 Multiple Compare-and-Set (MCAS)

We look at an interface over the heap which provides atomic double- (`dcas`) and triple-compare-and-set (`3cas`) operations, in addition to the basic read, write and compare-and-set operations.

5.2.1 Atomic Specification

Our atomic specification makes use of two abstract predicates: $\mathsf{MCL}(s, l)$ to represent an instance of the MCAS module with address l; and $\mathsf{MCP}(s, l, x, v)$ to represent the "MCAS heap cell" at address x with value v, protected by instance l. There is an abstract disjointness, as we can view each heap cell as disjoint from the others at the abstract level, even if that is not the case with the implementation

itself. The specification for creating the interface, transferring memory cells to and from it as well as manipulating it is given in Figure 5.5.

$$\lambda; \mathcal{A} \vdash \left\{\mathsf{True}\right\}\ \texttt{makeMCL()}\ \left\{\mathsf{MCL}(s, \mathsf{ret})\right\}$$

$$\lambda; \mathcal{A} \vdash \left\{\mathsf{MCL}(s, \texttt{l}) * \texttt{x} \mapsto v\right\}\ \texttt{makeMCP(l, x)}\ \left\{\mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\}$$

$$\lambda; \mathcal{A} \vdash \left\{\mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\}\ \texttt{unmakeMCP(l, x)}\ \left\{\texttt{x} \mapsto v\right\}$$

$$\lambda; \mathcal{A} \vdash \mathbb{V} v. \left\langle \mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\rangle\ \texttt{read(l, x)}\ \left\langle \mathsf{ret} = v * \mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\rangle$$

$$\lambda; \mathcal{A} \vdash \mathbb{V} v. \left\langle \mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\rangle\ \texttt{write(l, x, w)}\ \left\langle \mathsf{MCP}(s, \texttt{l}, \texttt{x}, \texttt{w})\right\rangle$$

$$\lambda; \mathcal{A} \vdash \mathbb{V} v. \left\langle \mathsf{MCP}(s, \texttt{l}, \texttt{x}, v)\right\rangle\ \texttt{cas(l, x, v1, v2)}\ \left\langle \begin{array}{r} \underline{\textbf{if}}\ v = \texttt{v1}\ \underline{\textbf{then}}\ \mathsf{ret} = 1 * \mathsf{MCP}(s, \texttt{l}, \texttt{x}, \texttt{v2}) \\ \underline{\textbf{else}}\ \mathsf{ret} = 0 * \mathsf{MCP}(s, \texttt{l}, \texttt{x}, v) \end{array}\right\rangle$$

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

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

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

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

Figure 5.5: The abstract specification for the MCAS module.

5.2.2 Implementation

We give a straightforward coarse-grained implementation of the MCAS specification. The full code is given in Figure 5.6. The operation `makeMCL` creates a lock which protects updates to pointers under the control of the module. The other operations simply acquire the lock, perform the appropriate reads and writes, and then release the lock. The module supports different kinds of compare-and-set operations, that allow more than one pointer to be updated atomically. The `makeMCP` and `unmakeMCP` operations allow pointers to be moved to the module and from the module. The `read` operation has to acquire the lock as well, this prevents reading intermediate states of the compare-and-set operations that update more than one pointer. Moreover, the `unmakeMCP` also has to acquire the lock to guarantee that no other operation is attempting to read the pointer. By acquiring the lock it ensures full ownership over the pointer which can then be moved away from the module.

We interpret the abstract predicates using a single shared region, with type name **MCAS**. The abstract states of the region are *partial heaps*, which represent the part of the heap that is protected by the module. For instance, the abstract state $x \mapsto v \bullet y \mapsto w$ indicates that heap cells x and y are under the protection of the module, with logical values v and w respectively. Note that the physical

```
function makeMCL() {
  l := makeLock();
  return l;
}

function makeMCP(l, x) {
  skip;
}

function read(l, x) {
  acquire(l);
  v := [x];
  release(l);
  return v;
}

function write(l, x, v) {
  acquire(l);
  [x] := v;
  release(l);
}

function dcas(l, x, y, v1, w1, v2, w2) {
  acquire(l);
  v := [x];
  w := [y];
  if (v = v1 and w = w1) {
    [x] := v2;
    [y] := w2;
    r := 1;
  } else {
    r := 0;
  }
  release(l);
  return r;
}

function unmakeMCP(l, x) {
  acquire(l);
  release(l);
}

function cas(l, x, v1, v2) {
  acquire(l);
  v := [x];
  if (v = v1) {
    [x] := v2;
    r := 1;
  } else {
    r := 0;
  }
  release(l);
  return r;
}

function 3cas(l, x, y, z, v1, w1, u1, v2, w2, u2) {
  acquire(l);
  v := [x];
  w := [y];
  u := [z];
  if (v = v1 and w = w1 and u = u1) {
    [x] := v2;
    [y] := w2;
    [z] := u2;
    r := 1;
  } else {
    r := 0;
  }
  release(l);
  return r;
}
```

Figure 5.6: Multiple compare-and-set module operations.

values at x and y need not be the same as their logical values, specifically when the lock has been acquired and they are being modified.

For the **MCAS** region, there are five kinds of guards. The $\textsc{Own}(x)$ guard confers ownership of the heap cell at address x under the control of the region. This guard is used by all operations of the module that access the heap cell x. In order to ensure that there can only be one instance of $\textsc{Own}(x)$ we require $\textsc{Own}(x) \bullet \textsc{Own}(x)$ to be undefined. We amalgamate the $\textsc{Own}$ guards for heap cells that are not currently under the protection of the module into $\textsc{Owned}(X)$, where X is the set of all cells that *are* protected. We define the following equality on guards:

$$\textsc{Owned}(X) = \textsc{Owned}(X \uplus \{x\}) \bullet \textsc{Own}(x)$$

Initially, the set X will be empty. When we add an element $x \mapsto v$ to the region, we get a guard $\textsc{Own}(x)$ that allows us to manipulate the abstract state for that particular x. There can be only one $\textsc{Owned}$ guard, we enforce this by requiring $\textsc{Owned}(X) \bullet \textsc{Owned}(Y)$ is not defined.

The remaining guards are effectively used as auxiliary state. When a thread acquires the lock, it removes some heap cells from the shared region in order to access them. The $\textsc{Locked}(h)$ guard will

be used to record that the heap cells in h have been removed in this way. The thread that acquired the lock will have a corresponding $\textsc{Key}(h)$ guard. When it releases the lock, the two guards will be reunited inside the region to form the $\textsc{Unlocked}$ guard. This is expressed by the following equivalence:

$$\textsc{Unlocked} = \textsc{Locked}(h) \bullet \textsc{Key}(h)$$

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

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

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

$$\begin{aligned} I(\mathbf{MCAS}_a(s', l, h)) \triangleq [\textsc{Owned}(\mathrm{dom}(h))]_a * (&\mathsf{Lock}(s', l, 0) * h * [\textsc{Unlocked}]_a \lor \\ &\exists h_1, h_2.\, \mathsf{Lock}(s', l, 1) * h_1 * [\textsc{Locked}(h_2)]_a * h = h_1 \bullet h_2) \end{aligned}$$

Internally, the region may be in one of two states, indicated by the disjunction. Either the lock l is unlocked, and the heap cells corresponding to the abstract state of the region are actually in the region, as well as the $\textsc{Unlocked}$ guard, or the lock l is locked, and some portion h_1 of the abstract heap is in the region, while the remainder h_2 has been removed, together with the $\textsc{Key}(h_2)$ guard, leaving behind the $\textsc{Locked}(h_2)$ guard. In both cases, the $\textsc{Owned}(\mathrm{dom}(h))$ guard belongs to the region, encapsulating the $\textsc{Own}$ guards for heap addresses that are not protected.

We now give an interpretation to the predicates as follows:

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

The predicate $\mathsf{MCL}(s, l)$ states the existence of the shared region, but makes no assumptions about its state. The predicate $\mathsf{MCP}(s, l, x, v)$ states that there is x with value v, which it owns, and possibly other heap cells in the region. The axiom $\mathsf{MCP}(s, l, x, v) * \mathsf{MCP}(s, l, x, w) \implies \mathsf{False}$ follows from the fact that $\textsc{Own}(x) \bullet \textsc{Own}(x)$ is not defined.

We can now prove that the specification is satisfied by the implementation. We show the `dcas` command in Fig. 5.7. The other commands have very similar proofs.

5.2.3 Resource Transfer

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

Figure 5.7: Proof of the `dcas` implementation.

implemented as follows:

```
function readTo(l, x, y) {
   v := read(l, x);
   [y] := v;
}
```

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

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

The readTo operation can be specified as follows:

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

One way of understanding such specifications is in terms of ownership transfer between a client and a module, as in [22]: ownership of the private precondition is transferred from the client; ownership of the private postcondition is transferred to the client. In this example, the same resources (albeit modified) are transferred in and out, but this need not be the case in general. For instance, an operation could allocate a fresh location in which to store the retrieved value, which is then transferred to the client.

5.3 Deque

We show how to use TaDA to specify a double-ended queue (deque) and verify its fine-grained implementation. A deque allows elements to be inserted and removed from both ends of a list.

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

We define a constructor makeDeque(), that creates a new empty deque. The deque supports the insertion and removal of elements from the left side and the right side. On the left side we can insert using pushLeft(d, v) and remove using popLeft(d). The right side has similar operations, pushRight(d, v) and popRight(d).

5.3.1 Atomic specification

We represent the deque state by the abstract predicate $\mathsf{Deque}(s, d, vs)$. It asserts that there is a deque at address d with list of elements vs.

The makeDeque() operation creates an empty deque and returns its address. It has the following specification:

$$\lambda; \mathcal{A} \vdash \left\{ \mathsf{True} \right\} \ \mathtt{makeDeque}() \ \left\{ \exists s \in \mathbb{T}_8.\, \mathsf{Deque}(s, \mathsf{ret}, []) \right\}$$

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

$$\lambda; \mathcal{A} \vdash \mathbb{V} vs. \left\langle \mathsf{Deque}(s, \mathtt{d}, vs) \right\rangle \ \mathtt{pushLeft}(\mathtt{d}, \mathtt{v}) \ \left\langle \mathsf{Deque}(s, \mathtt{d}, \mathtt{v} : vs) \right\rangle$$

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

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

5.3.2 Implementation

We consider an implementation that represents the deque as a doubly-linked list of nodes, based on the *Snark* linked-list deque [16]. The code is shown in Figure 5.8.

An example of the shape of the data structure is shown in Figure 5.9. Each node consists of a left-link pointer, a right-link pointer, and a value. There are two anchor variables, *left hat* and *right hat* ($\hat{l}$ and $\hat{r}$ in the figure), that generally point to the leftmost and rightmost node in the list, except when the deque is empty. When the deque is not empty, its leftmost node's left-link and the rightmost node's right-link point to a so-called *dead* node—a node whose left- and right-links point to itself (e.g. node a in the figure). When the deque is empty, then the left hat and the right hat point to dead nodes.

We focus on the `popLeft` implementation. This implementation first reads the left hat value to a local variable. It then reads the left-link of the node referenced by that variable. If both values are the same, it means that the node is dead and the list might be empty. It is necessary to recheck the left hat to confirm, since the node might have died since the left hat was first read. If the deque is indeed empty, the operation returns 0; otherwise it is restarted. If the left node is not dead, it tries to atomically update the left hat to point to the node to its right, and, at the same time, update the left node to be dead. This could fail, in which case the operation restarts. An example of such an update is shown in Figure 5.9. In order to update three pointers atomically, the implementation makes use of the `3cas` command described in §5.2.

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

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

In order to provide an interpretation for the abstract state, we first define a number of auxiliary predicates. A node predicate at address n in the deque is defined using the MCAS cells predicates:

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

```
function makeDeque() {
  L := makeMCL();
  n := makeNode(0, 0, 0);
  write(L, n.left, n);
  write(L, n.right, n);
  p := alloc(3);
  [p.left] := n;  // left hat
  [p.right] := n;  // right hat
  [p.mcl] := L;  // pointer to MCAS interface
  [p.dummy] := n;  // pointer to dummy node
  makeMCP(L, p.left);
  makeMCP(L, p.right);
  return p;
}

function makeNode(L, l, r, v) {
  n := alloc(3);
  [n.left] := l;
  [n.right] := r;
  [n.value] := v;
  makeMCP(L, l);
  makeMCP(L, r);
  return n;
}

function pushLeft(d, v) {
  L := [d.mcl];
  m := [d.dummy];
  n := makeNode(L, m, 0, v);
  while (true) {
    lh := read(L, d.left);
    lhL := read(L, lh.left);
    if (lhL = lh) {
      write(L, n.right, m);
      rh := read(L, d.right);
      b := dcas(L, d.left, lh.right, lh, rh, n, n);
      if (b = 1) {
        return ;
      }
    } else {
      write(L, n.right, lh);
      b := dcas(L, d.left, lh.left, lh, lhL, n, n);
      if (b = 1) {
        return ;
      }
    }
  }
}

function popLeft(d) {
  L := [d.mcl];
  while (true) {
    lh := read(L, d.left);
    lhR := read(L, lh.right);
    lhL := read(L, lh.left);
    if (lhL = lh) { // left hat seems dead
      lh2 := read(L, d.left);
      if (lh2 = lhL) { // left hat confirmed dead
        return 0;
      } // left hat not dead — try again
    } else {
      b := 3cas(L, d.left, lh.right, lh.left,
                lh, lhR, lhL, lhR, lh, lh);
      if (b = 1) {
        v := [lh.value];
        return v;
      }
    }
  }
}
```

where

$$\mathbb{E}.\texttt{left} \stackrel{\text{def}}{=} \mathbb{E} \quad \mathbb{E}.\texttt{right} \stackrel{\text{def}}{=} \mathbb{E}+1 \quad \mathbb{E}.\texttt{value} \stackrel{\text{def}}{=} \mathbb{E}+2 \quad \mathbb{E}.\texttt{mcl} \stackrel{\text{def}}{=} \mathbb{E}+2 \quad \mathbb{E}.\texttt{dummy} \stackrel{\text{def}}{=} \mathbb{E}+3.$$

Figure 5.8: Snark deque operations.

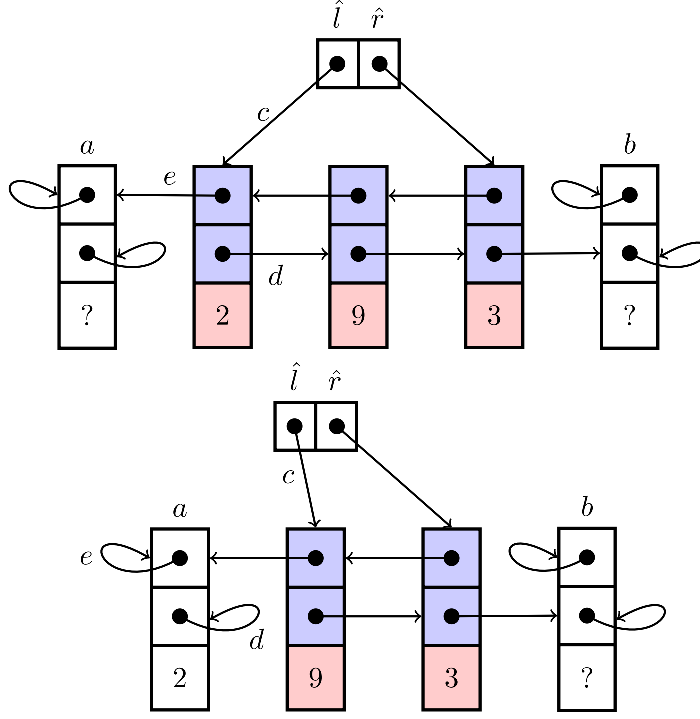


Figure 5.9: Examples of a deque before and after performing `popLeft`, which uses `3cas` to updated pointers c, d and e.

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

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

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

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

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

$$\mathsf{dls}(s, L, l, r, ns, ds) \equiv \exists a, b.\, (a, -), (b, -) \in ds * \mathsf{dlseg}(s, L, l, r, a, b, ns) * \circledast_{(n,v) \in ds} \mathsf{dead}(s, L, n, v)$$

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

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

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

We now define the interpretation of abstract states as follows:

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

We define the interpretation of the Deque predicate as follows:

$$\mathsf{Deque}((a, s', L), d, vs, (ns, ds)) \triangleq \mathbf{Deque}_a(s', d, L, ns, ds) * [\mathrm{G}]_a * vs = proj_2(ns)$$

where $proj_2$ is the second projection over the list of pairs ns.

We show the proof of the `popLeft` operation in Figure 5.10. To prove the implementation against our atomic specifications, we use the MakeAtomic rule again. The UpdateRegion rule is applied at two different points. The first one is when performing a read to confirm that the left hat is dead. We require the UpdateRegion rule as there is a possibility that the deque is empty and as such this corresponds to the atomic update, which requires updating the atomicity tracking component. The second point is when performing the `3cas` operation to remove an element of the deque, in the case of success it corresponds to the atomic update. The remaining proofs are similar.

We have shown that by using TaDA, we can prove atomicity for fine-grained implementations that use other concurrent implementations. We achieve scalability by using atomicity to abstract the updates at each abstraction level.

We have shown how we can use atomicity in our specifications to build up abstraction levels. The deque's abstract atomic specification is implemented using the abstract atomicity of the MCAS, which is, in turn, implemented using the abstract atomicity of the lock. With CAP, we could not do this because we cannot specify atomicity, only weaker specifications.

$\forall vs.$

$\langle \mathsf{Deque}(\mathtt{d}, vs) \rangle$

ABSTRACT: $s = (a, s', L),\ t = (ns, ds)$

$\forall (ns, vs).$

$\langle \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * [\mathrm{G}]_a * vs = proj_2(ns) \rangle$

MAKEATOMIC

$a : (ns, ds) \rightsquigarrow \underline{\textbf{if}}\ ns = []\ \underline{\textbf{then}}\ (ns, ds)\ \underline{\textbf{else}}\ (ns', (n, v) : ds) * ns = (n, v) : ns' \vdash$

$\{\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge\}$

$\mathtt{L} := [\mathtt{d.mcl}];$

$\mathtt{while\ (true)\ \{}$

$\{\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge * \mathtt{L} = L\}$

$\mathtt{lh} := \mathtt{read(L, d.left)}; \mathtt{lhR} := \mathtt{read(L, lh.right)}; \mathtt{lhL} := \mathtt{read(L, lh.left)};$

$\left\{\begin{array}{l}\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge * \mathtt{L} = L * \\ \underline{\textbf{if}}\ \mathtt{lh} = \mathtt{lhL}\ \underline{\textbf{then}}\ (\mathtt{lh}, -) \in ds \\ \quad \underline{\textbf{else}}\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns +\!\!+\ ds\end{array}\right\}$

$\mathtt{if\ (lhL = lh)\ \{}$ // left hat seems dead

$\{\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge * \mathtt{L} = L * (\mathtt{lhL}, -) \in ds\}$

UPDATEREGION

$\forall ns, ds.$

$\langle \mathsf{deque}(s', \mathtt{d}, L, ns, ds) * \mathtt{L} = L * (\mathtt{lhL}, -) \in ds \rangle$

$\mathtt{lh2} := \mathtt{read(L, d.left)};$

$\left\langle \begin{array}{l}\mathsf{deque}(s', \mathtt{d}, L, ns, ds) * \mathtt{L} = L * \\ (\mathtt{lh2} = \mathtt{lhL} \rightarrow ns = [])\end{array}\right\rangle$

$\left\{\begin{array}{l}\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * \mathtt{L} = L * \\ \underline{\textbf{if}}\ \mathtt{lh2} = \mathtt{lhL}\ \underline{\textbf{then}}\ a \Rrightarrow ([], ds), ([], ds)\ \underline{\textbf{else}}\ a \Rrightarrow \blacklozenge\end{array}\right\}$

$\mathtt{if\ (lh2 = lhL)\ \{}$ // left hat confirmed dead

$\mathtt{return}\ 0;$

$\{\exists ds.\ \mathsf{ret} = 0 * a \Rrightarrow ([], ds), ([], ds)\}$

$\}$ // left hat not dead — try again

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

$\left\{\begin{array}{l}\exists ns, ds.\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge * \mathtt{L} = L * \\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns +\!\!+\ ds\end{array}\right\}$

UPDATEREGION

$\forall ns, ds.$

$\left\langle \begin{array}{l}\mathsf{deque}(s', \mathtt{d}, L, ns, ds) * \mathtt{L} = L * \\ \{(\mathtt{lh}, -), (\mathtt{lhL}, -), (\mathtt{lhR}, -)\} \in ns +\!\!+\ ds\end{array}\right\rangle$

$\mathtt{b} := \mathtt{3cas(L, d.left, lh.right, lh.left, lh, lhR, lhL, lhR, lh, lh)};$

$\left\langle \begin{array}{l}\exists ns', v.\ \underline{\textbf{if}}\ \mathtt{b} = 1\ \underline{\textbf{then}}\ \left(\begin{array}{c}\mathsf{deque}(s', \mathtt{d}, L, ns', (\mathtt{lh}, v) : ds) * \\ \mathtt{L} = L * (\mathtt{lh}, v) \in ds * ns = (\mathtt{lh}, v) : ns'\end{array}\right) \\ \quad \underline{\textbf{else}}\ \mathsf{deque}(s', \mathtt{d}, L, ns, ds) * \mathtt{L} = L\end{array}\right\rangle$

$\left\{\begin{array}{l}\exists ns, ds, v.\ \underline{\textbf{if}}\ \mathtt{b} = 1\ \underline{\textbf{then}}\ \left(\begin{array}{c}a \Rrightarrow ((\mathtt{lh}, \mathtt{v}) : ns, ds), (ns, (\mathtt{lh}, \mathtt{v}) : ds) \\ * \mathtt{L} = L * (\mathtt{lh}, v) \in ds\end{array}\right) \\ \quad \underline{\textbf{else}}\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * a \Rrightarrow \blacklozenge * \mathtt{L} = L\end{array}\right\}$

$\mathtt{if\ (b = 1)\ \{}$

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

$\{\exists ns, ds.\ \mathsf{ret} = \mathtt{v} * a \Rrightarrow ((\mathtt{lh}, \mathtt{v}) : ns, ds), (ns, (\mathtt{lh}, \mathtt{v}) : ds)\}$

$\}\ \}\ \}$

$\left\langle \begin{array}{l}\underline{\textbf{if}}\ vs = []\ \underline{\textbf{then}}\ \mathsf{ret} = 0 * \mathbf{Deque}_a(s', \mathtt{d}, L, ns, ds) * [\mathrm{G}]_a \\ \quad \underline{\textbf{else}}\ \left(\begin{array}{c}\exists ns', v.\ ns = (n, v) : ns' * \mathsf{ret} = v * \\ \mathbf{Deque}_a(s', \mathtt{d}, L, ns', (n, v) : ds) * [\mathrm{G}]_a * vs' = proj_2(ns')\end{array}\right)\end{array}\right\rangle$

$\left\langle \begin{array}{l}\underline{\textbf{if}}\ vs = []\ \underline{\textbf{then}}\ \mathsf{ret} = 0 * \mathsf{Deque}(\mathtt{d}, vs) \\ \underline{\textbf{else}}\ \exists vs', v.\ vs = v : vs' * \mathsf{ret} = v * \mathsf{Deque}(\mathtt{d}, vs')\end{array}\right\rangle$

Figure 5.10: Proof of the `popLeft` implementation.

6 Related Work

TaDA inherits from a family of logics deriving from concurrent separation logic [47]: RGSep [63], Deny-Guarantee [15], CAP [12], Higher-Order CAP (HOCAP) [55] and Impredicative CAP (iCAP) [54]. In particular, it makes use of dynamic *shared regions* with capability resources (called *guards* in TaDA) that determine how the regions may be updated. Following iCAP, TaDA eschews the use of boxed assertions to describe the state of shared regions and instead represents regions by *abstract states.* The protocol for updating the region is specified as a transition system on these abstract states, labelled by guards. This use of transition systems to describe protocols derives from previous work by Dreyer *et al.* [17], and also appears in Turon *et al.* [61] as "local life stories".

By treating the abstract state-space of a region as a separation algebra, it is possible to localise updates on it, as in the MCAS example (§5.2). Such locality is in the spirit of local life stories [61], and can be seen as an instance of Ley-Wild and Nanevski's "subjective auxiliary state" [37].

While HOCAP and iCAP do not support abstract atomic specifications, they support an approach to atomicity introduced by Jacobs and Piessens [32] that achieves similar effects. In their work, operations may be parametrised by an update to auxiliary state that is performed when the abstract atomic operation appears to take effect. This update is performed atomically by the implementation, and can therefore involve shared regions. This approach is inherently higher-order, which has the disadvantage of leading to complex specifications, which are parametrised by the specification of the auxiliary code that performs the atomic update.

Recently, Iris [35] combined TaDA with iCAP [54] into a new higher order program logic, which encodes TaDA's proof rules in logic. Atomic triples have been encoded using view shifts in Iris [35] by interpreting them as specifications in the Jacobs-Piessens style. This captures the intensional meaning behind atomic triples—that is, what they can be used for—which in TaDA is expressed through the proof rules for using atomic triples. Because it is higher-order, it has expressive power to handle higher-order programs and reentrancy. TaDA takes a first-order approach, leading to simpler specifications. An alternative treatment of TaDA, in a refinement calculus setting, can be found in [45].

There has been extensive work understanding and generalising linearisability, especially in light of work on separation logic. Vafeiadis [63] has combined the ownership given by his RGSep reasoning with linearisability. Gotsman and Yang [22] have generalised linearisability to include ownership transfer of memory between a client and a module, which is also supported by our approach. We can do ownership transfer with the private and the public components. Filipovic *et al.* [19] have demonstrated that linearisability can be viewed as a particular proof technique for contextual refinement. Turon *et al.* [59] have introduced CaReSL, a logic that combines contextual refinement and Hoare-style reasoning to prove higher-order concurrent programs. Like linearisability, contextual refinement requires a whole-module approach.

7 Reasoning about Termination

Throughout this thesis, we have proved properties of programs using the partial correctness interpretation: if a program is run in a state satisfying the precondition and the program terminates, then the resulting state will be described by the postcondition. In some programs, it is also important to know *that* they terminate. To prove such a property is especially challenging for concurrent programs. When multiple threads are changing some shared resource, knowing if each thread terminates can often depend on the behaviour of the other threads and even on the scheduler that decides which thread should run at a particular moment.

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

If we prove that a program produces the correct results and also always completes in a finite time, we establish *total correctness*. Turing [58] and Floyd [20] introduced the use of well-founded relations, combined with partial-correctness arguments, to prove the termination of sequential programs. The same technique is general enough to prove concurrent programs, too. However, previous applications of this technique in the concurrent setting, which we discuss in §7.5.4, do not support straightforward reasoning about clients.

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

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

Using an atomic triple, an increment operation of a counter is specified as:

$$\vdash \mathbb{V} n \in \mathbb{N}. \left\langle \mathsf{Counter}(s, \mathtt{x}, n) \right\rangle \ \mathtt{incr}(\mathtt{x}) \ \left\langle \mathsf{Counter}(s, \mathtt{x}, n+1) * \mathsf{ret} = n \right\rangle$$

The internal structure of the counter is abstracted using the abstract predicate [50] $\mathsf{Counter}(s, x, n)$, which states that there is a counter at address x with value n, whereas s abstracts implementation-

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

Non-blocking Algorithms

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

There are two common non-blocking properties: *wait-freedom* [25] and *lock-freedom* [41]. Wait-freedom requires that operations complete irrespective of the interference caused by other threads: termination cannot depend on the amount of interference caused by the environment. Lock-freedom is less restrictive. It requires that, when multiple threads are performing operations, then at least one of them must make progress. This means that a thread might never terminate if the amount of interference caused by the environment is unlimited.

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

Moreover, there is another non-blocking property in the literature called *obstruction-freedom* [27]. It is the weakest requirement, guaranteeing progress only if a thread is run in isolation for a sufficiently long amount of time. In the presence of other operations, it can cause a livelock with each thread undoing the work done by other threads.

We cannot express in general obstruction-freedom in Total-TaDA, as these programs do not may not terminate without further constraints. It is possible to reason about this class of algorithms in Total-TaDA if one limits the amount of interference to guarantee termination.

Termination

Well-founded relations provide a general way to prove termination. In particular, Floyd [20] used well-founded relations to prove the termination of sequential programs. In fact, it is sufficient to use ordinal numbers [6] without losing expressivity. A Loop rule, using ordinals and adapted from Floyd's

work, has the form:

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

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

Total-TaDA

We obtain the program logic Total-TaDA by modifying TaDA to have a total-correctness semantics. The details are given in §7.2. With Total-TaDA, we can specify and verify non-blocking algorithms. Wait-free operations always terminate, independently of the operations performed by the environment. For lock-free operations, however, we need to restrict the amount of interference the environment can cause in order to guarantee termination. Our key insight is that, as well as bounding the number of iterations of loops, ordinals can bound the interference on a module. This allows us to give total-correctness specifications for lock-free algorithms. In §7.1, we specify and verify lock-free implementations of a counter. The specification introduces ordinals to bound the number of times a client may update the counter. This makes it possible to guarantee that the lock-free increment operation will terminate, since either it will succeed or some other concurrent increment will succeed. As the number of increments is bounded, the operation must eventually succeed.

Total-TaDA retains the modularity of TaDA. In particular, we can verify the termination of clients of modules using the total-correctness specifications, without reference to the implementation. We show an example of this in §7.1.2. Since the client only depends on the specification, we can replace the implementation. In §7.1.3, we show that two different implementations of a counter satisfy the same total-correctness specification. With Total-TaDA, we can verify the operations of a module independently, exploiting locality.

As a case study for Total-TaDA, we show how to specify and verify both functional correctness and termination of Treiber's stack in §7.3. In §7.4, we present the total-correctness semantics and the soundness proof of Total-TaDA. In §7.5, we show how lock-freedom and wait-freedom can be expressed with Total-TaDA specifications. We also introduce the concept of non-impedance in §7.5.3 and argue for its value in specifying non-blocking algorithms.

7.1 Motivating Examples

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

Consider a counter module with a constructor `makeCounter` and two operations: `incr`, which increments the value of the counter by 1 and returns its previous value; and `read`, which returns the

value of the counter. We give an implementation of the spin counter operations in Figure 7.1, and an alternative implementation of `incr` in Figure 7.5.

7.1.1 Atomic Specification

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

$$\forall\alpha.\ \vdash_\tau \left\{\mathsf{True}\right\}\ \mathtt{makeCounter}()\ \left\{\exists s.\ \mathsf{Counter}(s, \mathsf{ret}, 0, \alpha)\right\}$$

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

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

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

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

The increment operation is specified as follows:

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

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

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

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

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

Figure 7.1: Spin counter operations.

counter's ordinal cannot be increased to guarantee termination.

The read operation is specified as follows:

$$\vdash_\tau \mathbb{V} n \in \mathbb{N}, \alpha. \Big\langle \mathsf{Counter}(s, \mathtt{x}, n, \alpha) \Big\rangle \; \mathtt{read(x)} \; \Big\langle \mathsf{Counter}(s, \mathtt{x}, n, \alpha) * \mathsf{ret} = n \Big\rangle$$

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

Finally, we have an axiom that allows the client to decrease the ordinal without requiring any physical operation to be performed:

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

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

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

7.1.2 Clients

A Sequential Client

Consider a program that creates a counter and contains two nested loops. As in the previous example, the outer loop runs a finite but randomly determined number of times. The inner loop also runs a randomly determined number of times, and increments the counter on each iteration. Figure 7.2 shows this client, together with its total-correctness proof.

The Loop rule is used for each of the loops: for the outer loop, the variant is `n`; for the inner loop, the variant is `m`. Since the number of iterations of each loop is determined before it is run, the variants need only be considered up to finite ordinals (*i.e.* natural numbers). We could modify the code to use a single loop that conditionally decrements `n` (and randomises `m`) or decrements `m`. This variation would require a transfinite ordinal for the variant.

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

A Concurrent Client

Consider a program that creates two threads, each of which increments the counter a finite, but unbounded, number of times. We again prove this client using the abstract specification of the counter, as shown in Figure 7.3. For presentation purposes, we present the example using parallel composition rather than using fork. One could rewrite the example to use fork and encode the join of threads using auxiliary heap state.

We reuse the Parallel from §2.6:

$$\textsc{Parallel}\quad \frac{\left\{P_1\right\}\ \mathbb{C}_1\ \left\{Q_1\right\} \qquad \left\{P_2\right\}\ \mathbb{C}_2\ \left\{Q_2\right\}}{\left\{P_1 * P_2\right\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \left\{Q_1 * Q_2\right\}}$$

In this example, the counter is shared between the two threads, which may concurrently update it. To reason about sharing, we use a *shared region*.

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

```
{True}
x := makeCounter();
{∃s. Counter(s, x, 0, ω²)}
n := random();
{∃s. Counter(s, x, 0, ω · n) ∗ (α = n)}
while (n > 0) {
    ∀γ.
    {∃s, υ. Counter(s, x, υ, ω · n) ∗ (γ = n) ∗ n > 0}
    m := random();
    {∃s, υ. Counter(s, x, υ, ω · (n − 1) + m) ∗ (γ = n) ∗ n > 0}
    | while (m > 0) {                                              [Frame: (γ = n) ∗ n > 0]
    |     ∀δ.
    |     {∃s, υ. Counter(s, x, υ, ω · (n − 1) + m) ∗ (δ = m) ∗ m > 0}
    |     incr(x);
    |     {∃s, υ. Counter(s, x, υ, ω · (n − 1) + m − 1) ∗ (δ = m) > 0}
    |     m := m − 1;
    |     {∃s, ζ, υ. Counter(s, x, υ, ω · (n − 1) + m) ∗ (ζ = m) ∗ ζ < δ}
    | }
    {∃s, υ. Counter(s, x, υ, ω · (n − 1)) ∗ (γ = n) ∗ n > 0}
    n := n − 1;
    {∃s, β, υ. Counter(s, x, υ, ω · n) ∗ (β = n) ∗ β < γ}
}
{∃s, υ. Counter(s, x, υ, 0)}
```

Figure 7.2: Proof of a sequential client of the counter.

corresponding transition. All guards are allocated along with the region they are associated with.

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

There are two types of guard resources associated with **CClient** regions. The guard $\textsc{Inc}(m, \beta, \pi)$ provides capability to increment the counter. Conceptually, multiple threads may have $\textsc{Inc}$ guards, and a fractional permission $\pi \in (0, 1]$ (in the style of [4]) is used to keep track of these capabilities. The parameter m expresses the *local contribution* to the value of the counter — the actual value is the sum of the local contributions. The ordinal parameter β represents a local bound on the number of increments. Again, the actual bound is a sum of the local bounds. Standard ordinal addition is inconvenient since it is not commutative; we use the natural (or Hessenberg) sum [28], denoted $\oplus$, which is associative, commutative, and monotone in its arguments.

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

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

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

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

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

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

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

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

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

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

It remains to define the interpretation of the region states:

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

By interpreting the state $\circ$ as True, we allow a thread transitioning into that state to acquire the counter that previously belonged to the region. This justifies the last step of the proof in Figure 7.3.

The proof rule that allows us to use the atomic specification of the `incr` operation to update the shared region is the UseAtomic rule, inherited from TaDA. The $\{\}$-assertions in Total-TaDA are also required to be stable. That is, the region states must account for the changes that the concurrent environment could make, under the assumption that it has guards that are compatible with those of the thread. This is why the state of the **CClient** region is always existentially quantified in Figure 7.3.

$$\{\mathsf{True}\}$$
$$\mathtt{x} := \mathtt{makeCounter}();$$
$$\{\exists s.\, \mathsf{Counter}(s, \mathtt{x}, 0, \omega \oplus \omega)\}$$
$$\{\exists s, a.\, \mathbf{CClient}_a(s, \mathtt{x}, 0) * [\textsc{Inc}(0, \omega \oplus \omega, 1)]_a\}$$

Left thread:

$$\{\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) * [\textsc{Inc}(0, \omega, \tfrac{1}{2})]_a * 0 \leq v\}$$
$$\mathtt{n} := \mathtt{random}();$$
$$\mathtt{i} := 0;$$
$$\{\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \mathtt{n}, \tfrac{1}{2})]_a * 0 \leq v * \mathtt{i} = 0\}$$
$$\mathtt{while}\ (\mathtt{i} < \mathtt{n})\ \{$$
$$\forall \beta.\ \left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \beta, \tfrac{1}{2})]_a * \mathtt{i} \leq v \\ * \mathtt{i} < \mathtt{n} * \beta = \mathtt{n} - \mathtt{i}\end{array}\right\}$$
$$\mathtt{incr}(\mathtt{x});\ \mathtt{i} := \mathtt{i} + 1;$$
$$\left\{\begin{array}{l}\exists s, \delta, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{i}, \delta, \tfrac{1}{2})]_a * \mathtt{i} \leq v \\ * \mathtt{i} \leq \mathtt{n} * \delta = \mathtt{n} - \mathtt{i} * \delta < \beta\end{array}\right\}$$
$$\}$$
$$\{\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) * [\textsc{Inc}(\mathtt{n}, 0, \tfrac{1}{2})]_a\}$$

$\|$

Right thread:

$$\left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) \\ * [\textsc{Inc}(0, \omega, \tfrac{1}{2})]_a\end{array}\right\}$$
$$\mathtt{m} := \mathtt{random}();$$
$$\mathtt{j} := 0;$$
$$\mathtt{while}\ (\mathtt{j} < \mathtt{m})\ \{$$
$$\mathtt{incr}(\mathtt{x});$$
$$\mathtt{j} := \mathtt{j} + 1;$$
$$\}$$
$$\left\{\begin{array}{l}\exists s, v.\, \mathbf{CClient}_a(s, \mathtt{x}, v) \\ * [\textsc{Inc}(\mathtt{m}, 0, \tfrac{1}{2})]_a\end{array}\right\}$$

After the parallel composition:

$$\{\exists s, a.\, \mathbf{CClient}_a(s, \mathtt{x}, \mathtt{n} + \mathtt{m}) * [\textsc{Inc}(\mathtt{n} + \mathtt{m}, 0, 1)]_a\}$$
$$\{\exists s.\, \mathsf{Counter}(s, \mathtt{x}, \mathtt{n} + \mathtt{m}, 0)\}$$

Figure 7.3: Proof of a concurrent client of the counter.

7.1.3 Implementations

We prove the total correctness of the two distinct increment implementations against the abstract specification given in §7.1.1.

Spin Counter Increment

Consider `incr` as shown in Figure 7.1. Note that the read, write and compare-and-set operations are atomic. We want to prove the total correctness of `incr` against the atomic specification. The first step is to give a concrete interpretation of the abstract predicate $\mathsf{Counter}(s, x, n, \alpha)$. We introduce a new region type, **Counter**, with only one non-empty guard, G. The abstract states of the region are pairs of the form (n, α), where n is the value of the counter and α is a bound on the number of increments. All transitions are guarded by G with the transition:

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

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

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

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

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

We define the interpretation of the abstract predicate as follows:

$$\mathsf{Counter}(a, x, n, \alpha) \triangleq \mathbf{Counter}_a(x, n, \alpha) * [\mathrm{G}]_a$$

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

The specification for the increment is atomic and as such, we use the MakeAtomic rule which is just the same as that of TaDA. The only difference is that termination is enforced. Whereas in TaDA it would be possible for an abstract atomic operation to loop forever without performing its atomic update, in Total-TaDA it is guaranteed to eventually perform the update.

A proof of the increment implementation is shown in Figure 7.4. The atomicity context allows the environment to modify the abstract state of the counter, without restriction on the number of times. The **Counter** transition system enforces that the ordinal α must decrease every time the value of the counter is increased. This means that the number of times the region's abstract state is updated is finite. Our loop invariant is parametrised with a variant γ that takes the value of α at the beginning of each loop iteration. When we first read the value of the counter n, we can assert: $n > \mathtt{v} \implies \gamma > \alpha$.

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

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

Backoff Increment

Consider a different implementation of the increment operation, given in Figure 7.5, which loops attempting to perform the operation, like the previous increment. However, if the compare-and-set fails due to contention, it waits for a random number of iterations before retrying.

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

The main difference in the proof is that each iteration of the loop depends on not only the amount of interference on the counter, but also on the variable `n` that is randomised when the compare-and-set

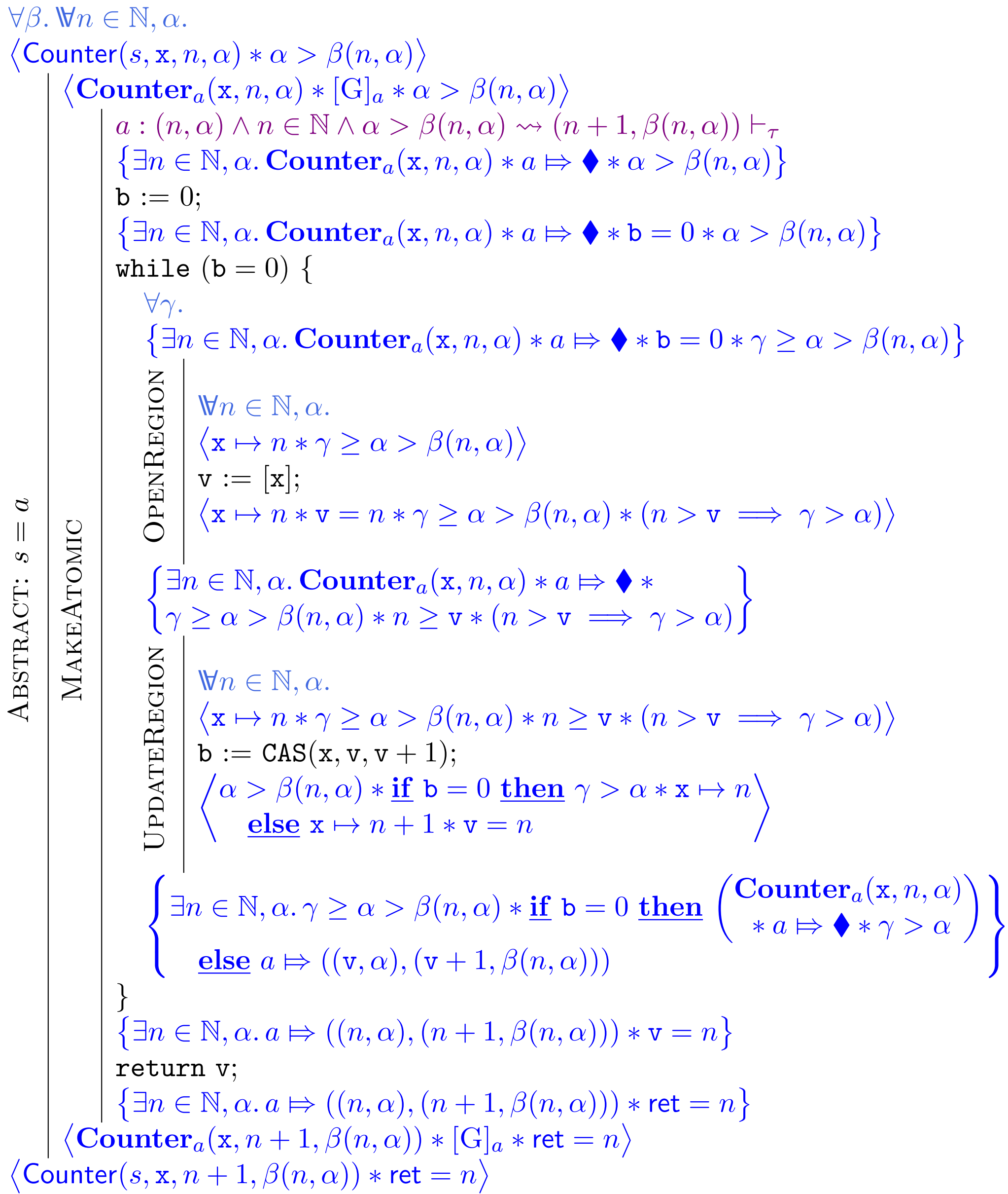


Figure 7.4: Proof of total correctness of increment.

fails. Any random number will be smaller than ω, and the maximum amount of times that the compare-and-set can fail is α, the parameter of the Counter predicate. This is because α is a bound on the number of times the counter can be incremented. We therefore use $\omega \cdot \alpha + \mathtt{n}$ as the upper bound on the number of loop iterations.

Let γ be equal to $\omega \cdot \alpha + \mathtt{n}$ at the start of the loop iteration. At each loop iteration, we have two cases, when $\mathtt{n} = 0$ or otherwise. In the first case we try to perform the increment by doing a compare-and-set. If the compare-and-set succeeds, then the increment occurs and the loop will exit. If it fails, then the environment must have decreased α. This means that $\gamma \geq \omega \cdot \alpha + \omega$ for the new value of α. We then set $\mathtt{n}$ to be a new random number, which is less than ω, and end up with $\gamma > \omega \cdot \alpha + \mathtt{n}$. In the second case of the loop iteration, we simply decrement $\mathtt{n}$ by 1 and we know that $\gamma > \omega \cdot \alpha + \mathtt{n}$ for the new value of $\mathtt{n}$. The proof of the backoff increment is shown in Figure 7.6.

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

Figure 7.5: Backoff increment.

7.2 Logic

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

The Total-TaDA proof judgement has the form:

$$\lambda; \mathcal{A} \vdash_\tau \mathbb{V}x \in X. \left\langle P_p \middle| P(x) \right\rangle \mathbb{C} \quad \exists y \in Y. \left\langle Q_p(x,y) \middle| Q(x,y) \right\rangle.$$

The meaning of the judgement is similar to TaDA's, however it has a total correctness interpretation, which additionally enforces that $\mathbb{C}$ will eventually terminate. The pre- and postconditions are split into a private part (the P_p and $Q_p(x,y)$) and a public part (the $P(x)$ and $Q(x,y)$). The idea is that the command may make multiple, non-atomic updates to the private part, but must only make a single atomic update to the public part. Before the atomic update, the environment is allowed to change the public part of the state, but only by changing the parameter x of P which must remain within X. After the atomic update, the specification makes no constraint on how the environment modifies the public state. All that is known is that, immediately after the atomic update, the public and private parts satisfy the postcondition for a common value of y. The private assertions in our judgements must be *stable*: that is, they must account for any updates other threads could have sufficient resources to perform. The main difference is that the judgement has a total-correctness interpretation, i.e. $\mathbb{C}$ must eventually terminate. We use the τ subscript to emphasise the total-correctness interpretation.

The proof rules are mostly the same as in TaDA, except for the LOOP rule and, FUNCTION rule, which use variants to prohibit non-termination as follows:

$$\forall \beta.\, \mathbb{W} n \in \mathbb{N}, \alpha.$$
$$\left\langle \mathsf{Counter}(s, \mathtt{x}, n, \alpha) * \alpha > \beta(n, \alpha) \right\rangle$$

Abstract: $s = a$

$$\left\langle \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * [\mathrm{G}]_a * \alpha > \beta(n, \alpha) \right\rangle$$

MakeAtomic

$$a : (n, \alpha) \wedge n \in \mathbb{N} \wedge \alpha > \beta(n, \alpha) \rightsquigarrow (n+1, \beta(n, \alpha)) \vdash_\tau$$

```
n := 0; b := 0;
```

$$\left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \mathtt{n} = 0 * \mathtt{b} = 0 * \alpha > \beta(n, \alpha) \right\}$$

```
while (b = 0) {
```

$$\forall \gamma.$$
$$\left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \mathtt{b} = 0 * \gamma \geq \omega \cdot \alpha + \mathtt{n} * \alpha > \beta(n, \alpha) \right\}$$

```
if (n = 0) {
```

$$\left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \gamma \geq \omega \cdot \alpha * \alpha > \beta(n, \alpha) \right\}$$

OpenRegion

$$\mathbb{W} n \in \mathbb{N}, \alpha.$$
$$\left\langle \mathtt{x} \mapsto n * \gamma \geq \omega \cdot \alpha * \alpha > \beta(n, \alpha) \right\rangle$$

```
v := [x];
```

$$\left\langle \mathtt{x} \mapsto n * \mathtt{v} = n * \gamma \geq \omega \cdot \alpha * \alpha > \beta(n, \alpha) * (n > \mathtt{v} \implies \gamma \geq \omega \cdot \alpha + \omega) \right\rangle$$

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \gamma \geq \omega \cdot \alpha * \alpha > \beta(n, \alpha) \\ * n \geq \mathtt{v} * (n > \mathtt{v} \implies \gamma \geq \omega \cdot \alpha + \omega) \end{array} \right\}$$

UpdateRegion

$$\mathbb{W} n \in \mathbb{N}, \alpha.$$
$$\left\langle \mathtt{x} \mapsto n * \gamma \geq \omega \cdot \alpha * \alpha > \beta(n, \alpha) * n \geq \mathtt{v} * (n > \mathtt{v} \implies \gamma \geq \omega \cdot \alpha + \omega) \right\rangle$$

```
b := CAS(x, v, v + 1);
```

$$\left\langle \alpha > \beta(n, \alpha) * \underline{\textbf{if}}\ \mathtt{b} = 0\ \underline{\textbf{then}}\ \gamma \geq \omega \cdot \alpha + \omega * \mathtt{x} \mapsto n\ \underline{\textbf{else}}\ \mathtt{x} \mapsto n + 1 * \mathtt{v} = n \right\rangle$$

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) * \underline{\textbf{if}}\ \mathtt{b} = 0\ \underline{\textbf{then}}\ \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \gamma \geq \omega \cdot \alpha + \omega \\ \quad \underline{\textbf{else}}\ a \Mapsto ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\}$$

```
n := random();
```

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) * \underline{\textbf{if}}\ \mathtt{b} = 0\ \underline{\textbf{then}}\ \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \gamma > \omega \cdot \alpha + \mathtt{n} \\ \quad \underline{\textbf{else}}\ a \Mapsto ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\}$$

```
} else {
```

$$\left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \mathtt{b} = 0 * \gamma \geq \omega \cdot \alpha + \mathtt{n} * \alpha > \beta(n, \alpha) \right\}$$

```
n := n - 1;
```

$$\left\{ \exists n \in \mathbb{N}, \alpha.\, \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \mathtt{b} = 0 * \gamma > \omega \cdot \alpha + \mathtt{n} * \alpha > \beta(n, \alpha) \right\}$$

```
}
```

$$\left\{ \begin{array}{l} \exists n \in \mathbb{N}, \alpha.\, \alpha > \beta(n, \alpha) * \underline{\textbf{if}}\ \mathtt{b} = 0\ \underline{\textbf{then}}\ \mathbf{Counter}_a(\mathtt{x}, n, \alpha) * a \Mapsto \blacklozenge * \gamma > \omega \cdot \alpha + \mathtt{n} \\ \quad \underline{\textbf{else}}\ a \Mapsto ((\mathtt{v}, \alpha), (\mathtt{v} + 1, \beta(n, \alpha))) \end{array} \right\}$$

```
}
```

$$\left\{ \exists n \in \mathbb{N}, \alpha, .\, a \Mapsto ((n, \alpha), (n + 1, \beta(n, \alpha))) * \mathtt{v} = n \right\}$$

```
return v;
```

$$\left\{ \exists n \in \mathbb{N}, \alpha.\, a \Mapsto ((n, \alpha), (n + 1, \beta(n, \alpha))) * \mathsf{ret} = n \right\}$$

$$\left\langle \mathbf{Counter}_a(\mathtt{x}, n + 1, \beta(n, \alpha)) * [\mathrm{G}]_a * \mathsf{ret} = n \right\rangle$$

$$\left\langle \mathsf{Counter}(s, \mathtt{x}, n + 1, \beta(n, \alpha)) * \mathsf{ret} = n \right\rangle$$

Figure 7.6: Proof of total correctness of backoff increment.

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

$$\frac{\begin{array}{c} \Gamma = \Gamma', \forall \gamma < \alpha.\, \lambda; \mathcal{A} \vdash_\tau \mathbb{V} x \in X. \left\langle P_p(\vec{z}, \gamma) \,\middle|\, P(x, \vec{z}) \right\rangle \texttt{f}(\vec{z}) \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \\ \mathit{code}(\eta(\texttt{f})) = \mathbb{C}\ \texttt{return}\ \mathbb{E}; \qquad \mathit{vars}(\eta(\texttt{f})) = \vec{\texttt{x}} \\ \Gamma'; \lambda; \mathcal{A} \vdash_\tau \begin{array}{c} \mathbb{V} x \in X. \left\langle P_p(\vec{z}, \alpha) * \vec{\texttt{x}} = \vec{z} \,\middle|\, P(x, \vec{z}) \right\rangle \\ \mathbb{C} \\ \exists\!\!\exists (y, \mathsf{ret}) \in Y \times \mathsf{Val}. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathbb{E} \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle \end{array} \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash_\tau \mathbb{V} x \in X. \left\langle P_p(\vec{z}, \alpha) \,\middle|\, P(x, \vec{z}) \right\rangle \texttt{f}(\vec{z}) \quad \exists\!\!\exists y \in Y. \left\langle Q_p(x, y, \vec{z}, \mathsf{ret}) \,\middle|\, Q(x, y, \vec{z}, \mathsf{ret}) \right\rangle} \text{ FUNCTION}$$

The LOOP rule enforces that the number of times that the loop body can run is finite. The rule allows us to perform a while loop if we can guarantee that each loop iteration decreases the ordinal parametrising the invariant P. By the finite-chain property of ordinals, there cannot be an infinite number of iterations.

The FUNCTION rule enforces that the number of times that function body can call itself or other functions is finite. This is enforced similarly to the loop, by decreasing the ordinal parametrising the precondition of the function. Note that this has a direct impact on the FUNCTIONCALL rule and FORK rule, as they make use of the function specifications. We are guaranteeing that both function calls and forks occur a finite amount of times within a program.

Moreover, we add a new proof rule to the non-deterministic function used throughout this chapter:

$$\frac{}{\Gamma; \lambda; \mathcal{A} \vdash_\tau \left\{ \mathsf{True} \right\} \texttt{x := random();} \left\{ \exists n \in \mathbb{N}.\, \texttt{x} = n \right\}} \text{ RANDOM}$$

7.3 Case Study: Treiber's Stack

We now consider a version of Treiber's stack [57] to demonstrate how Total-TaDA can be applied to verify the total correctness of larger modules.

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

$$\forall \beta.\ \vdash_\tau \mathbb{V} vs, t, \alpha. \left\langle \mathsf{Stack}(s, \texttt{x}, vs, t, \alpha) * \alpha > \beta(vs, \alpha) \right\rangle \texttt{push(x, v)} \left\langle \exists t'.\, \mathsf{Stack}(s, \texttt{x}, \texttt{v} : vs, t', \beta(vs, \alpha)) \right\rangle$$

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

Figure 7.7: Stack operation specifications.

7.3.1 Atomic Specification

In Figure 7.7, we give the specification of the lock-free stack operations. This is a Total-TaDA specification satisfiable by a reasonable non-blocking implementation. As with the counter, the predicate representing the stack is parametrised by an ordinal that bounds the number of operations on the stack, in order to guarantee termination. The $\mathsf{Stack}(s, x, vs, t, \alpha)$ predicate has five parameters: the address of the stack x; its contents vs; an ordinal α that decreases every time a `push` operation is performed; and two parameters, s and t that range over abstract types $\mathbb{T}_1$ and $\mathbb{T}_2$ respectively. These last two parameters encapsulate implementation-specific information about the configuration of the stack (s is invariant, while t may vary) and hence their types are abstract to the client. Note that it is possible to abstract the parameter t using the AExists rule in this case.

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

Note that the ordinal parametrising the stack is not required to decrease when popping the stack. This means that the stack operations cannot be starved by an unbounded number of `pop` invocations. This need not be the case in general for a lock-free stack, but it is true for Treiber's stack. We discuss the ramifications of this kind of specification further in §7.5.3.

7.3.2 Implementation

Figure 7.8 gives an implementation of the stack operations based on Treiber's stack [57]. The stack is represented as a heap cell containing a pointer (the head pointer) to a singly-linked list of the values on the stack. Values are pushed onto the stack by allocating a new node holding the value to be pushed and a pointer to the old head of the stack. A compare-and-set operation updates the old head of the stack to point to the new node. If the operation fails, it will be because the head of the stack has changed, and so the operation is retried. Values are popped from the stack by moving the head pointer one step along the list. Again, a compare-and-set operation is used for this update, so if the head of the stack changes the operation can be retried. If the stack is empty (*i.e.* the head points to 0), then pop simply returns 0, without affecting the stack.

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

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

It is important for the correctness of the algorithm that nodes that have been popped can never reappear as the head of the stack. Otherwise, a `pop` operation could have seen the node when it was previously the head, and update the head to point to what was then the next node, but may no longer be. To account for this, in our representation of the stack we track the set of previously popped nodes, and ensure that they are disjoint from the nodes in the stack. The $\mathsf{stack}(x, ns, ds)$ predicate, therefore,

consists of a list starting at address x, with contents ns, and a disjoint set of discarded nodes ds:

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

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

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

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

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

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

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

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

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

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

The proof for the pop operation is given in Figure 7.9. When the stack is non-empty, if the compare-

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

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

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

where

$$\mathbb{E}.\mathtt{value} \stackrel{\text{def}}{=} \mathbb{E} \qquad \mathbb{E}.\mathtt{next} \stackrel{\text{def}}{=} \mathbb{E} + 1.$$

Figure 7.8: Treiber's stack operations.

and-set fails then another thread must have succeeded in updating the stack, and so reduced the ordinal or the length of the stack; by basing the loop variant on the ordinal and stack length, we can guarantee that the operation will eventually succeed.

The proof of the `push` operation is in Figure 7.10. If the compare-and-set fails, then another thread must have updated the stack by removing or adding an element. As in the `pop` operation proof, updating the stack reduces the ordinal or the length of the stack. We related the loop variant on the ordinal and the stack length to guarantee that the operation will eventually succeed.

7.4 Semantics and Soundness

The semantics of Total-TaDA are mostly similar to that for TaDA, we will focus on the differences.

We first extend the operational semantics of the language to include the new function as follows:

$$\textsc{Random}\quad \frac{n \in \mathbb{N}}{(\sigma, \mathtt{x} := \mathtt{random}();) \xrightarrow{\mathsf{id}}_{\eta} (\sigma[\mathtt{x} \mapsto n], \mathtt{skip};)}$$

We use the same model for assertions as that for TaDA. We also use a similar semantic judgement, $\vDash_\tau$, which ensures that the concrete behaviours of programs simulate the abstract behaviours represented by the specifications.

Definition 7.1 (Semantic Judgement)**.** The semantic judgement

$$\eta; \lambda; \mathcal{A} \vDash_\tau \forall x \in X. \left\langle P_p \,\middle|\, P(x) \right\rangle\ c\quad \exists y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle$$

where

- $\eta \in \mathsf{FEnv}$ is a function environment;
- $\lambda \in \mathsf{Level}$ 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;
- $P_p \in \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ 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;
- $P \in X \to \mathsf{View}_{\mathcal{A}}$ is the public part of the precondition, which may correspond to resources from some opened shared regions, and is parametrised by $x \in X$ that tracks the precondition at the atomic update and by the valuation of program variables;
- $c \in \mathsf{ExtCmd}$ is the program under consideration;
- $Q_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_{\mathcal{A}}$ is the private part of the postcondition, which is parametrised by $x \in X$ that tracks the precondition at the atomic update, by $y \in Y$ that tracks the postcondition at the atomic update, and by the valuation of program variables;
- $Q \in X \times Y \to \mathsf{View}_{\mathcal{A}}$ is the public part of the postcondition, which is similarly parametrised by $x \in X$ and $y \in Y$, and by the valuation of program variables;

$$\mathbb{V} vs, t, \alpha.$$
$$\langle \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) \rangle$$

ABSTRACT: $s = a,\ t = (ns, ds)$

$$\langle \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * [\mathrm{G}]_a * vs = \mathsf{snds}(ns) \rangle$$

MAKEATOMIC

$$a : (ns, ds, \alpha) \rightsquigarrow \underline{\mathbf{if}}\ ns = []\ \underline{\mathbf{then}}\ (ns, ds, \alpha)\ \underline{\mathbf{else}}\ (ns', (n, v) \uplus ds, \alpha) \land ns = (n, v) : ns' \vdash_\tau$$
$$\{\exists ns, ds, \alpha.\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge\}$$

```
do {
```

$$\forall \gamma.$$
$$\{\exists ns, ds, \alpha.\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * \gamma \geq 2 \cdot \alpha + |ns|\}$$

UPDATEREGION

$$\mathbb{V} ns, ds, \alpha.$$
$$\langle \exists w.\ \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) * \gamma \geq 2 \cdot \alpha + |ns| \rangle$$

```
y := [x];
```

$$\left\langle \begin{array}{l} \mathtt{x} \mapsto \mathtt{y} * \mathsf{stack}(\mathtt{y}, ns, ds) * \gamma \geq 2 \cdot \alpha + |ns| * \\ \underline{\mathbf{if}}\ \mathtt{y} = 0\ \underline{\mathbf{then}}\ ns = []\ \underline{\mathbf{else}}\ \exists v.\ (\mathtt{y}, v) = \mathsf{head}(ns) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\ \underline{\mathbf{if}}\ \mathtt{y} = 0\ \underline{\mathbf{then}}\ a \Rrightarrow (([], ds, \alpha), ([], ds, \alpha)) \\ \quad \underline{\mathbf{else}}\ \exists v. \left(\begin{array}{l} \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * (\mathtt{y}, v) \in ns +\!\!+ ds * \\ \gamma \geq 2 \cdot \alpha + |ns| * \mathsf{head}(ns) \neq (\mathtt{y}, v) \implies \gamma > 2 \cdot \alpha + |ns| \end{array} \right) \end{array} \right\}$$

```
if (y = 0) {
  return 0;
```

$$\{\exists ds, \alpha.\ a \Rrightarrow (([], ds, \alpha), ([], ds, \alpha)) * \mathsf{ret} = 0\}$$

```
}
```

$$\left\{ \begin{array}{l} \exists ns, ds, v, \alpha.\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * (\mathtt{y}, v) \in ns +\!\!+ ds * \\ \gamma \geq 2 \cdot \alpha + |ns| * \mathsf{head}(ns) \neq (\mathtt{y}, v) \implies \gamma > 2 \cdot \alpha + |ns| \end{array} \right\}$$

```
z := [y.next];
```

$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * \gamma \geq 2 \cdot \alpha + |ns| * \\ \left(\begin{array}{l} (\exists v, v', ns'.\ ns = [(\mathtt{y}, v), (\mathtt{z}, v')] +\!\!+ ns') \lor (\exists v.\ ns = [(\mathtt{y}, v)] * \mathtt{z} = 0) \\ \lor (\exists v.\ (\mathtt{y}, v) \in ns +\!\!+ ds * \mathsf{head}(ns) \neq (\mathtt{y}, v) * \gamma > 2 \cdot \alpha + |ns|) \end{array} \right) \end{array} \right\}$$

UPDATEREGION

$$\mathbb{V} ns, ds, \alpha.$$
$$\left\langle \begin{array}{l} \exists w.\ \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) * \gamma \geq 2 \cdot \alpha + |ns| * \\ \left(\begin{array}{l} (\exists v, v', ns'.\ ns = [(\mathtt{y}, v), (\mathtt{z}, v')] +\!\!+ ns') \\ \lor (\exists v.\ ns = [(\mathtt{y}, v)] * \mathtt{z} = 0) \\ \lor (\exists v.\ (\mathtt{y}, v) \in ns +\!\!+ ds * \mathsf{head}(ns) \neq (\mathtt{y}, v) * \gamma > 2 \cdot \alpha + |ns|) \end{array} \right) \end{array} \right\rangle$$

```
b := CAS(x, y, z);
```

$$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \exists w.\ \mathtt{x} \mapsto w * \mathsf{stack}(w, ns, ds) * \gamma > 2 \cdot \alpha + |ns| \\ \quad \underline{\mathbf{else}}\ \exists v, ns'.\ \mathtt{x} \mapsto \mathtt{z} * \mathsf{stack}(\mathtt{z}, ns', (\mathtt{y}, v) \uplus ds) * ns = (\mathtt{y}, v) : ns' \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} \exists ns, ds, \alpha.\ \gamma \geq 2 \cdot \alpha + |ns| * \\ \underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * \gamma > 2 \cdot \alpha + |ns| \\ \quad \underline{\mathbf{else}} \left(\begin{array}{l} \exists v, ns', ds', \alpha'.\ (\mathtt{y}, v) \in ds' * \mathbf{TStack}_a(\mathtt{x}, ns', ds', \alpha') \\ \quad * a \Rrightarrow (((\mathtt{y}, v) : ns, ds, \alpha), (ns, (\mathtt{y}, v) \uplus ds), \alpha) \end{array} \right) \end{array} \right\}$$

```
} while (b = 0);
```

$$\left\{ \begin{array}{l} \exists v, ns, ds, \alpha, ns', ds', \alpha'.\ (\mathtt{y}, v) \in ds' * \mathbf{TStack}_a(\mathtt{x}, ns', ds', \alpha') \\ * a \Rrightarrow (((\mathtt{y}, v) : ns, ds, \alpha), (ns, (\mathtt{y}, v) \uplus ds, \alpha)) \end{array} \right\}$$

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

$$\{\exists ns, ds, \alpha.\ a \Rrightarrow (((\mathtt{y}, \mathtt{v}) : ns, ds, \alpha), (ns, (\mathtt{y}, \mathtt{v}) \uplus ds, \alpha))\}$$

```
return v;
```

$$\{\exists y, ns, ds, \alpha.\ a \Rrightarrow (((y, \mathsf{ret}) : ns, ds, \alpha), (ns, (y, \mathsf{ret}) \uplus ds), \alpha)\}$$

$$\left\langle \begin{array}{l} \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * [\mathrm{G}]_a * vs = \mathsf{snds}(ns) * \mathsf{ret} = 0 \\ \quad \underline{\mathbf{else}}\ \exists ns', vs', y.\ \mathbf{TStack}_a(\mathtt{x}, ns', (y, \mathsf{ret}) \uplus ds, \alpha) * [\mathrm{G}]_a * vs' = \mathsf{snds}(ns') * ns = (y, \mathsf{ret}) : ns' \end{array} \right\rangle$$

$$\langle \underline{\mathbf{if}}\ vs = []\ \underline{\mathbf{then}}\ \mathsf{Stack}(s, \mathtt{x}, vs, t, \alpha) * \mathsf{ret} = 0\ \underline{\mathbf{else}}\ \exists vs', t'.\ \mathsf{Stack}(s, \mathtt{x}, vs', t', \alpha) * vs = \mathsf{ret} : vs' \rangle$$

Figure 7.9: Proof of total correctness of Treiber's stack `pop` operation.

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

ABSTRACT: $s = a,\ t = (ns, ds)$

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

MAKEATOMIC

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

OPENREGION

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

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

UPDATEREGION

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

$$\left\{\begin{array}{l}\underline{\mathbf{if}}\ \mathtt{b} = 0\ \underline{\mathbf{then}}\ \left(\begin{array}{c}\exists ns, ds, \alpha.\, \mathbf{TStack}_a(\mathtt{x}, ns, ds, \alpha) * a \Rrightarrow \blacklozenge * \\ \mathsf{node}(\mathtt{y}, \mathtt{v}, _) * \alpha > \beta(\alpha, \mathsf{snds}(ns)) * \gamma > 2 \cdot \alpha + |ns|\end{array}\right) \\ \quad \underline{\mathbf{else}}\ \exists ns, ds, \alpha.\, a \Rrightarrow ((ns, ds, \alpha), ((\mathtt{y}, \mathtt{v}) : ns, ds, \beta(\alpha, \mathsf{snds}(ns))))\end{array}\right\}$$
```
} while (b = 0);
```
$$\{\exists ns, ds, \alpha.\, a \Rrightarrow ((ns, ds, \alpha), ((\mathtt{y}, \mathtt{v}) : ns, ds, \beta(\alpha, \mathsf{snds}(ns))))\}$$

$$\langle \exists n.\, \mathbf{TStack}_a(\mathtt{x}, (n, \mathtt{v}) : ns, ds, \beta(\alpha, vs)) * [\mathrm{G}]_a * \mathtt{v} : vs = \mathsf{snds}((n, \mathtt{v}) : ns) \rangle$$
$$\langle \exists t'.\, \mathsf{Stack}(s, \mathtt{x}, \mathtt{v} : vs, t', \beta(\alpha, vs)) \rangle$$

Figure 7.10: Proof of total correctness of Treiber's stack `push` operation.

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

- For all $\sigma, \sigma_1 \in \mathsf{Store}$, $c_1 \in \mathsf{ExtCmd}$, $\alpha \in \mathsf{AAction}$ with $(\sigma, c) \xrightarrow{\alpha}_\eta (\sigma_1, c_1)$, for all $x \in X$, there exists $P'_p \in \mathsf{Store} \to \mathsf{View}_\mathcal{A}$, $P''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_\mathcal{A}$ such that

$$\eta; \lambda; \mathcal{A} \vDash \langle P_p(\sigma) * P(x) \rangle \; \alpha \; \langle P'_p(\sigma_1) * P(x) \vee \exists y \in Y.\, P''_p(x, y, \sigma_1) * Q(x, y) \rangle \tag{7.1}$$

$$\eta; \lambda; \mathcal{A} \vDash_\tau \mathbb{V} x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle \; c_1 \quad \mathbb{I} y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle, \tag{7.2}$$

$$\text{and for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P''_p(x, y) \right\} \; c_1 \; \left\{ Q_p(x, y) \right\}. \tag{7.3}$$

- For all $\sigma, \sigma_1 \in \mathsf{Store}$, $c_1 \in \mathsf{ExtCmd}$, $\mathtt{f}$, $\vec{v}$ with $(\sigma, c) \xrightarrow{\mathsf{spawn}(\mathtt{f}, \vec{v})}_\eta (\sigma_1, c_1)$, for all $x \in X$, there exist $P'_p \in \mathsf{Store} \to \mathsf{View}_\mathcal{A}$, $P''_p \in X \times Y \to \mathsf{Store} \to \mathsf{View}_\mathcal{A}$ and $P_\mathtt{f} \in \mathsf{Store} \to \mathsf{View}$ such that for all $\sigma_\mathtt{f} \in \mathsf{Store}$ with $\mathit{vars}(\eta(\mathtt{f})) = \vec{\mathtt{x}}$, $\sigma_\mathtt{f}(\vec{\mathtt{x}}) = \vec{v}$ and $\mathit{code}(\eta(\mathtt{f})) = \mathbb{C}\ \mathtt{return}\ \mathbb{E};$,

$$\lambda; \mathcal{A} \vDash P_p(\sigma) * P(x) \preceq P'_p(\sigma_1) * P_\mathtt{f}(\sigma_\mathtt{f}) * P(x) \vee \exists y \in Y.\, P''_p(\sigma_1) * P_\mathtt{f}(\sigma_\mathtt{f}) * Q(x, y), \tag{7.4}$$

$$\eta; \lambda; \mathcal{A} \vDash_\tau \mathbb{V} x \in X. \left\langle P'_p \,\middle|\, P(x) \right\rangle \; c_1 \quad \mathbb{I} y \in Y. \left\langle Q_p(x, y) \,\middle|\, Q(x, y) \right\rangle, \tag{7.5}$$

$$\text{for all } y \in Y,\ \eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P''_p(x, y) \right\} \; c_1 \; \left\{ Q_p(x, y) \right\}, \tag{7.6}$$

$$\text{and } \eta; \lambda; \emptyset \vDash_\tau \begin{array}{c} \mathbb{V} x \in \mathbf{1}. \left\langle \lambda \sigma_\mathtt{f}.\, P_\mathtt{f}(\vec{z}) * \sigma_\mathtt{f}(\vec{\mathtt{x}}) = \vec{z} \,\middle|\, \mathsf{True} \right\rangle \\ \mathbb{C} \\ \mathbb{I}(y, \mathsf{ret}) \in \mathbf{1} \times \mathsf{Val}. \left\langle \lambda \sigma_\mathtt{f}.\, Q_\mathtt{f}(\vec{z}, \mathsf{ret}) * \mathsf{ret} = \mathcal{E}[\![\mathbb{E}]\!]_{\sigma_\mathtt{f}} \,\middle|\, \mathsf{True} \right\rangle \end{array}. \tag{7.7}$$

- If $c = \mathtt{skip}$; then, for all $\sigma \in \mathsf{Store}$, $x \in X$, there exists $y \in Y$ such that

$$\lambda; \mathcal{A} \vDash P_p(\sigma) * P(x) \preceq Q_p(x, y, \sigma) * Q(x, y). \tag{7.8}$$

The key distinction is that, whereas in TaDA the judgement (4.61) is defined coinductively (as a greatest fixed point), in Total-TaDA the judgement is defined inductively (as a least fixed point). By the Knaster-Tarski theorem [56], we know it has a least fixed-point. This means that TaDA admits executions that never terminate, while Total-TaDA requires executions to always terminate: that is, reach a base case of the inductive definition.

The proof of soundness of Total-TaDA is similar to that for TaDA. The soundness proof consists of lemmas that justify each of the proof rules for the semantic judgement. Most of the Total-TaDA rules have similar proofs to the corresponding TaDA rules, but proceed by induction instead of coinduction. Of course, the Loop and Function rules are different, since termination does not follow trivially. We give the proof for Loop.

Lemma 7.1 (Loop Rule)**.** Let α be an ordinal. If, for all $\gamma \leq \alpha$,

$$\eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P_p(\gamma) \wedge \mathbb{B} \right\} \; \mathbb{C} \; \left\{ \exists \beta.\, P_p(\beta) \wedge \beta < \gamma \right\} \tag{7.9}$$

then

$$\eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P_p(\alpha) \right\} \; \mathtt{while}\ (\mathbb{B})\ \{\mathbb{C}\} \; \left\{ \exists \beta.\, P_p(\beta) \wedge \neg \mathbb{B} \wedge \beta \leq \alpha \right\}. \tag{7.10}$$

Proof. The proof is by transfinite induction on the ordinal α. As the inductive hypothesis, we shall assume that the lemma holds for all $\delta < \alpha$. Since `while` $(\mathbb{B})$ $\{\mathbb{C}\}$ has two possible reductions, both with transition id, to show (7.10), it is sufficient to establish:

$$\eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P_p(\alpha) \wedge \mathbb{B} \right\} \mathbb{C}\ \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\} \left\{ \exists \beta.\, P_p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha \right\} \tag{7.11}$$

$$\eta; \lambda; \mathcal{A} \vDash_\tau \left\{ P_p(\alpha) \wedge \neg\mathbb{B} \right\} \texttt{skip;} \left\{ \exists \beta.\, P_p(\beta) \wedge \neg\mathbb{B} \wedge \beta \leq \alpha \right\} \tag{7.12}$$

This is sufficient since the first condition of the semantic judgement is the only one that may apply. For the reduction $(\sigma, \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}) \xrightarrow{\mathsf{id}} (\sigma, \mathbb{C}\ \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\})$ (which requires $\mathbb{B}(\sigma)$), take $P'_p = P_p(\alpha) \wedge \mathbb{B}$ and $P''_p = \mathsf{False}$. The first and third sub-conditions become trivial, while the second reduces to (7.11). For the reduction $(\sigma, \texttt{while}\ (\mathbb{B})\ \{\mathbb{C}\}) \xrightarrow{\mathsf{id}} (\sigma, \texttt{skip;})$ (which requires $\neg\mathbb{B}(\sigma)$), take $P'_p = P_p(\alpha) \wedge \neg\mathbb{B}$ and $P''_p = \mathsf{False}$. Similarly, the first and third sub-conditions are trivial and the second reduces to (7.12).

To establish (7.11), we have from (7.9), for $\gamma = \alpha$ and renaming β to δ:

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

Next, for all $\delta < \alpha$, we have, by the inductive hypothesis[1], that

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

and hence

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

Now (7.11) follows from the above by the sequencing lemma.

On the other hand, it is trivial to establish (7.12) by choosing α as the witness for β. □

7.5 Non-blocking Properties

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

7.5.1 Lock-freedom

We have described lock-freedom in terms of an informal notion of "progress". In order to properly characterise modules as lock-free, we need a more formal definition. We can characterise global progress for a module as follows: at any time, eventually either a pending operation will be completed or another operation will be begun. If we assume that the number of threads is bounded, then as long as there are pending module operations, some operation will eventually complete. If the number of threads

[1] Since we have that $\forall \gamma \leq \alpha.\ \eta; \lambda; \mathcal{A} \vDash_\tau \{P_p(\gamma) \wedge \mathbb{B}\}\ \mathbb{C}\ \{\exists \beta.\, P_p(\beta) \wedge \beta < \gamma\}$, we certainly have that $\forall \gamma \leq \delta.\ \lambda; \mathcal{A} \vDash_\tau \{P_p(\gamma) \wedge \mathbb{B}\}\ \mathbb{C}\ \{\exists \beta.\, P_p(\beta) \wedge \beta < \gamma\}$, since $\delta < \alpha$. This allows us to obtain the consequent of the inductive hypothesis.

is unbounded, then there is no guarantee that any operation will complete, even if it is scheduled arbitrarily often, since additional operations can always begin.

Based on this observation, Gotsman *et al.* [23] reduced lock-freedom to the termination of a simple class of programs, the bounded most-general clients (BMGCs) of a module. Hoffmann *et al.* [30] generalised the result to apply to algorithms where the identity or the number of threads is significant. An (m, n)-bounded general client consists of m threads which each invoke n module operations in sequence. If all such bounded general clients (for all n and m)[2] terminate, then the module is lock-free.

Definition 7.2 ((m, n)-bounded General Client)**.** Consider a module $\mathcal{M}$ with initialiser `init` and a set of operations O. We define the following sets of programs:

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

Theorem 7.2 (Hoffmann *et al.* [30])**.** Given a module $\mathcal{M}$, if, for all m and n, every program $c \in C_{m,n}$ terminates, then $\mathcal{M}$ is lock free.

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

Theorem 7.3 (Lock-freedom)**.** Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract parameters and an ordinal parameter), suppose that the following specifications are provable:

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

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

Proof. By Theorem 7.2, it is sufficient to show that, for arbitrary m, n and $c \in C_{m,n}$, the program c terminates. Fix the number of threads m.

We define a region type $\mathbf{M}$ whose abstract states consist of vectors $\bar{x} \in \mathbb{N}^m$. We denote by x_i, for $1 \le i \le m$, the i-th component of vector $\bar{x}$. We denote by $\sum \bar{x}$ the sum $\sum_{i=1}^{i=m} x_i$. Region states are interpreted as follows:

$$I(\mathbf{M}_a(s, \bar{x})) \triangleq \exists u.\, \mathsf{M}\left(s, u, \textstyle\sum \bar{x}\right).$$

The guard algebra for $\mathbf{M}$ consists of m distinct guards $\mathrm{G}_1, \ldots, \mathrm{G}_m$. Formally, the guards are subsets of $\{\mathrm{G}_i \mid 1 \le i \le m\}$ under disjoint union. The state transition system for $\mathbf{M}$ allows a thread holding guard G_i to decrease the i-th component of the abstract state:

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

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

For $1 \leq i \leq m$, arbitrary n, and $\texttt{op} \in O$, we have

$$\left\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a * x_i = n + 1\right\}$$

$$\left\{\exists \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a * x_i = n + 1\right\}$$

$$\left\langle \exists u.\, \mathsf{M}(s, u, k + n + 1) \right\rangle$$

$$\mathbb{V} u, k.$$

$$\left\langle \mathsf{M}(s, u, k + n + 1) \right\rangle$$

$$\texttt{op}$$

$$\left\langle \exists u'.\, \mathsf{M}(s, u', k + n) \right\rangle$$

$$\left\langle \exists u'.\, \mathsf{M}(s, u', k + n) * [\mathrm{G}_i]_a * x_i = n \right\rangle$$

$$\left\{\exists \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a * x_i = n\right\}$$

$$\left\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_i]_a * x_i = n\right\}$$

(Rules applied, from outermost to innermost: EXISTS, USEATOMIC, AEXISTS.)

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

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

Let $c = \texttt{init}; (t_1 \| \ldots \| t_m) \in C_{m,n}$ be arbitrary. We derive $\vdash_\tau \left\{\mathsf{True}\right\}\ c\ \left\{\mathsf{True}\right\}$ by choosing $n \cdot m$ as the initial ordinal and creating an $\mathbf{M}$-region with initial state $(n, \ldots, n)$ as follows:

$$\left\{\mathsf{True}\right\}$$

$$\texttt{init};$$

$$\left\{\exists s, u.\, \mathsf{M}(s, u, n \cdot m)\right\}$$

$$//\ \text{create region}$$

$$\left\{\exists a, s.\, \mathbf{M}_a(s, (n, \ldots, n)) * [\mathrm{G}_1]_a * \ldots * [\mathrm{G}_m]_a\right\}$$

$$\left\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_1]_a * x_1 = n\right\} t_1 \left\{\mathsf{True}\right\} \Big\| \ldots \Big\| \left\{\exists s, \bar{x}.\, \mathbf{M}_a(s, \bar{x}) * [\mathrm{G}_m]_a * x_m = n\right\} t_m \left\{\mathsf{True}\right\}$$

$$\left\{\mathsf{True}\right\}$$

Consequently, c terminates, as required. $\square$

It is straightforward to apply Theorem 7.3 to the modules we have considered.

7.5.2 Wait-freedom

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

Theorem 7.4 (Wait-freedom)**.** Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract

parameters), suppose that the following specifications are provable:

$$\vdash_\tau \left\{\mathsf{True}\right\} \ \texttt{init} \ \left\{\exists s, t.\, \mathsf{M}(s, u)\right\}$$
$$\forall \texttt{op} \in O.\ \vdash_\tau \mathbb{V} u. \left\langle \mathsf{M}(s, u) \right\rangle \ \texttt{op} \ \left\langle \exists u'.\, \mathsf{M}(s, u') \right\rangle.$$

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

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

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

Theorem 7.5 (Wait-free Operation)**.** Given a module $\mathcal{M}$ and some abstract predicate M (with two abstract parameters), suppose that the following specifications are provable:

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

Then `op` is wait-free.

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

The specifications required by Theorem 7.5 do not follow from those given for our examples. However, where applicable, the proofs can easily be adapted. For instance, to show that the `read` operation of the counter is wait-free, we would remove the ordinals from the region definition, and abstract the value of the counter. This breaks the termination proof for the increment operations, but we can adapt it to a partial-correctness proof in TaDA. The termination proof for `read` does not depend on the ordinal parameter of the region, and so we can still establish total correctness, as required.

7.5.3 Non-impedance

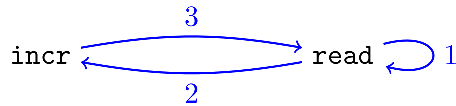
Recall the counter specification from §7.1.1. If we abstract the value and address of the counter (which are irrelevant to termination), the specification becomes:

$$\forall\alpha.\ \vdash_{\tau} \left\{\mathsf{True}\right\}\ \mathtt{x} := \mathtt{makeCounter}();\ \left\{\exists s \in \mathbb{T}_1, u \in \mathbb{T}_2.\, \mathsf{Counter}(s, u, \alpha)\right\}$$

$$\vdash_{\tau} \mathbb{V}u, \alpha. \left\langle \mathsf{Counter}(s, u, \alpha) \right\rangle\ \mathtt{read(x)}\ \left\langle \mathsf{Counter}(s, u, \alpha) \right\rangle$$

$$\forall\beta.\ \vdash_{\tau} \mathbb{V}u, \alpha. \left\langle \mathsf{Counter}(s, u, \alpha) * \alpha > \beta(\alpha) \right\rangle\ \mathtt{incr(x)}\ \left\langle \exists u'.\, \mathsf{Counter}(s, u', \beta(\alpha)) \right\rangle$$

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



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

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

Recall that we can prove that `read` is wait-free by giving a different specification as in Theorem 7.5. An operation is wait-free precisely when every operation does not impede it. For `read`, this is expressed by edges 1 and 3 in the above graph.

The stack specification in Figure 7.7, much like the counter specification, implies that `pop` does not impede either `push` or `pop`:

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

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

7.5.4 Related Work

Hoffmann *et al.* [30] introduced a concurrent separation logic for verifying total correctness. By adapting the most-general-client approach of Gotsman *et al.* [23], they establish that modules are lock-free. They do not, however, establish functional correctness. This method involves a thread passing “tokens” to other threads whose lock-free operations are impeded by modifications to the shared state. Subsequent approaches [3, 40] also use some form of tokens that are used up in loops or function calls. These approaches require special proof rules for the tokens. When these approaches restrict to dealing with finite numbers of tokens, support for unbounded non-determinism (as in the backoff increment example of Figure 7.6) is limited. In our extension of TaDA to reason about termination, named Total-TaDA, such token passing is not necessary. Instead, we require the client to provide a general (ordinal) limit on the amount of impeding interference. Consequently, we can guarantee the termination of loops with standard proof rules.

Jia *et al.* have introduced loop depth counters to prove lock-freedom [33]. Their approach adds auxiliary code to the loops that counts the number of times progress towards an operation’s completion is made. Their approach was applied to stack and queue modules using a tool.

Liang *et al.* [40] have developed a proof theory for termination-preserving refinement, applying it to verify linearisability and lock-freedom. Their approach constrains impedance by requiring that impeding actions correspond to progress at the abstract level. In Total-TaDA, such constraints are made by requiring that impeding actions decrease an ordinal associated with a shared region. Their approach does not freely combine lock-free and wait-free specifications, whereas with Total-TaDA we can reason about lock- and wait-freedom in combination, as well as about more subtle conditions, such as non-impedance. For example, we can show when a read operation of a lock-free data-structure is wait-free. Their specifications establish termination-preserving refinement (given a context, if the abstract program is guaranteed to terminate, then so is the concrete), whereas Total-TaDA specifications establish termination (in a context, the program will terminate).

Boström and Müller [3] have introduced an approach that can verify termination and progress properties of concurrent programs. The approach supports blocking concurrency and non-terminating programs, which Total-TaDA does not. However, the approach does not aim at racy concurrent programs and cannot deal with any of the examples shown in this chapter. Furthermore, the relationship between termination and lock- and wait-freedom is not considered.

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

8 Extending the logic

We have formally verified that implementations of concurrent data-structures with abstractly atomic operations. However, implementations can use techniques that make verification difficult. Among the most challenging of these are *helping* and *speculation* [1].

Typically, we would expect that an implementation of an abstractly atomic operation performs some concrete atomic update during its execution that effects the abstract operation. However, in general it is possible for this update to be performed by a different thread, which is in the process of performing another operation. We refer to this scenario as *helping*: one thread is performing an abstract operation on behalf of another.

Helping is particularly common in *wait-free* algorithms, where each operation is guaranteed to terminate. Suppose that one operation has partially completed when another begins. The second operation cannot be performed while the first is partially complete, so it must either wait for the first operation, undo the progress of the first operation, or help complete the first operation. The first two options do not guarantee termination, so helping becomes the only viable option.

We might also expect that, at the point the abstractly atomic operation of a thread is performed, we would know this to be the case. However, in general we may only know that it might be the case, and future behaviour will determine whether it was or not. We refer to this situation as *speculation*: we consider speculatively the cases where the abstract operation was and was not performed, and decide which applies later in the execution.

As an example of speculation, consider a data-structure that implements an abstractly atomic read operation as follows: it performs two concretely atomic reads and then randomly decides which result to return. The abstractly atomic read happens at one of the two concretely atomic reads, but exactly which is determined later. This example is illustrative but is, however, contrived. In practice, speculation does not arise as a result of random decisions of a single thread but from the non-deterministic behaviour of the concurrent environment.

We show how to verify abstractly atomic operations of concurrent data-structures that use helping and speculation by extending TaDA. To prove an atomic specification in TaDA, it is necessary to show that a single atomic step in the implementation performs the corresponding abstract atomic action. This is achieved by a special resource that allows the thread to perform the action, and witnesses when it is completed. However, in TaDA this resource cannot be transferred to other threads, and so it does not support helping. Our extension, TaDA 2, supports helping by allowing such resources (called *proxies*, which embody permission to perform the abstract action, and *witnesses*, which guarantee that the action has been performed) to be transferred between threads.

This may seem like a simple change, but it requires significant changes to the semantics and soundness proof of the logic. In particular, the TaDA semantics of atomic specifications ties the abstractly atomic update to a concrete atomic operation in the implementation, thus precluding helping.

We support speculation in TaDA 2 by introducing *speculative resources*. These speculative resources

consist of a set of possible (speculative) configurations of proxy and witness resources. When we speculatively perform the action of a proxy, we introduce cases in the set corresponding to whether or not the action is performed. We can resolve the speculation by choosing a particular configuration from the set.

8.1 Motivating Examples

We motivate our work with two examples that implement abstract atomicity using helping and speculation. Our examples build on an abstract counter module described in §3.1. We start by revising the ticket lock module (§8.1.1), but now we are going to specify it with an atomic specification. The ticket lock illustrates helping: when a thread releases the lock, it acquires the lock on behalf of the next thread waiting for the lock (if any). The second is a counter algorithm that internally increments in two steps (§8.1.2). The two-step counter illustrates speculation: when multiple threads have partially completed their increment operations, it will appear that half of them have abstractly incremented the counter; however, which 'half' is determined by the order in which they complete the operation.

8.1.1 Ticket Lock

We consider the ticket lock module from §3.2. The lock uses two counters, `next` and `owner`, which initially have value 0. The counter `next` is used to generate the tickets and the counter `owner` is used to indicate which ticket holds the lock. When their values match the lock is considered to be unlocked.

The `acquire` operation starts by incrementing the counter `next`; the previous value of the counter represents the ticket held by the thread. It then waits until the value in the counter `owner` matches its ticket. When they match, the thread has acquired the lock.

The `release` operation simply increments the value of the counter `owner`. Since only one thread can hold a lock at a time, the `owner` counter is not subject to concurrent increments, and so the `wkIncr` operation is appropriate.

Atomic Specification

We introduce an abstract predicate $\mathsf{TLock}(s, x, l, t)$ to represent the lock as a resource. The second parameter, $x \in \mathsf{Addr}$, is the address of the lock object. The third parameter, $l \in \{0, 1\}$, indicates the state of the lock: 0 for unlocked, 1 for locked. The two parameters, $s \in \mathbb{T}_2$ and $t \in \mathbb{T}_3$, range over *abstract types*. The types $\mathbb{T}_2$ and $\mathbb{T}_3$ will capture implementation-specific information about the configuration of the lock. The ticket lock has the following specification:

$$\vdash \left\{\mathsf{True}\right\} \ \texttt{makeLock()} \ \left\{\exists s \in \mathbb{T}_2, t \in \mathbb{T}_3.\, \mathsf{TLock}(s, \mathsf{ret}, 0, t)\right\}$$

$$\vdash \mathbb{A}(l, t) \in \{0, 1\} \times \mathbb{T}_3. \left\langle \mathsf{TLock}(s, \mathtt{x}, l, t)\right\rangle \ \texttt{acquire(x)} \ \left\langle \mathsf{TLock}(s, \mathtt{x}, 1, t) * l = 0\right\rangle$$

$$\vdash \left\langle \mathsf{TLock}(s, \mathtt{x}, 1, t)\right\rangle \ \texttt{release(x)} \ \left\langle \exists t' \in \mathbb{T}_3.\, \mathsf{TLock}(s, \mathtt{x}, 0, t')\right\rangle$$

Since the `release` operation makes use of `wkIncr`, it is important this operation does not race with any other increment of the `owner` counter. Without the t parameter, we could imagine specifying an operation `nop` that increments both the `owner` and `next` counters in a single atomic step. As this would

not conceptually change whether the lock is in the locked or unlocked state, we might specify it as:

$$\vdash \mathbb{V} l \in \{0,1\}. \left\langle \mathsf{TLock}(s, \mathtt{x}, l) \right\rangle \ \mathtt{nop}(\mathtt{x}) \ \left\langle \mathsf{TLock}(s, \mathtt{x}, l) \right\rangle$$

However, such an operation causes havoc: a thread waiting to acquire the lock may proceed, even though the thread that actually holds the lock has not released it! The implementation ties the additional parameter t to the value of the `owner` counter. The specification for the `release` operation consequently prevents the environment from concurrently modifying this counter (as it is not bound by $\mathbb{V}$).

Implementation

In order to verify the specification, we must provide an interpretation for the abstract predicate $\mathsf{TLock}(s, x, l, t)$. For this, we introduce shared regions. As in TaDA, a shared region encapsulates some resources that may be shared by multiple threads, with the proviso that they can only be accessed by atomic operations. A shared region has an abstract state (which has a concrete interpretation), and a protocol that determines how the state can change. A shared region is also equipped with resources called guards, which determine how threads can update the region's shared state.

For the ticket lock, we define a region type **TLock**. The abstract states of **TLock** regions consist of pairs $(l, n) \in \{0,1\} \times \mathbb{N}$, where l indicates whether the lock is currently locked and n indicates the current value of the `owner` counter. The protocol for the **TLock** region is specified by the following transition system, labelled by guards:

$$\begin{aligned} \mathrm{G} &: \forall n \in \mathbb{N}.\, (1, n) \rightsquigarrow (0, n+1) \\ \mathrm{G} &: \forall n \in \mathbb{N}.\, (0, n) \rightsquigarrow (1, n) \end{aligned}$$

The first of these transitions allows a thread holding the (unique) guard resource G for the region to release the lock, increasing the `owner` counter in doing so. The second transition allows a thread holding G to acquire the lock, without updating the `owner` counter.

In addition to the G guard, we define two other types of guards associated with **TLock** regions. These guards do not play a role in the protocol, but are used as ghost resources to manage bookkeeping. The first of these is the $\textsc{Ticket}(v, i)$ guard, where $v \in \mathbb{N}$ is a number and $i \in \mathsf{TId}$ is a thread identifier. This guard represents thread i's claim to hold the ticket number v, while waiting to acquire the lock. The final guard $\textsc{Queue}(S)$ accounts for the tickets that have been issued, where $S \subseteq \mathbb{N} \times \mathsf{TId}$. These guards are defined to obey the following equation:

$$\textsc{Queue}(S) = \textsc{Queue}(S \uplus \{(v, i)\}) \bullet \textsc{Ticket}(v, i) \text{ if } v \notin \mathit{proj}_1(S), i \notin \mathit{proj}_2(S)$$

As in the previous chapters, $\bullet$ is the composition operator on guard resources, which lifts to $*$ in assertions. It is associative, with $\mathbf{0}$ as the neutral element. The following compositions are not allowed:

$$\begin{array}{ll} \mathrm{G} \bullet \mathrm{G} & \\ \textsc{Queue}(S) \bullet \textsc{Queue}(S') & \\ \textsc{Ticket}(v, i) \bullet \textsc{Ticket}(v', i') & \text{if } v = v' \text{ or } i = i' \\ \textsc{Queue}(S) \bullet \textsc{Ticket}(v, i) & \text{if } (v, i) \notin S \end{array}$$

It remains to define the state interpretation for **TLock** regions. However, since the ticket lock algorithm uses helping, we must first describe how atomic specifications are proved in TaDA 2. There are two proof rules that are key to atomicity proofs. The first is the MakeAtomic rule, which allows us to prove that an operation can be seen as abstractly atomic. A slightly simplified version of this rule is as follows:

$$\frac{\{(x,y) \mid x \in X, y \in Q(x)\} \subseteq \mathcal{T}_{\mathbf{t}}(\mathrm{G})^* \qquad a:i \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\bar{z},x) * \left\langle \forall x \in X \rightsquigarrow Q(x) \right|_a^i \right\} \ \mathbb{C}\ \left\{\exists x \in X, y \in Q(x).\, \left| x \rightsquigarrow y \right\rangle_a^i \right\}}{\vdash \forall x \in X. \left\langle \mathbf{t}_a(\bar{z},x) * [\mathrm{G}]_a \right\rangle \ \mathbb{C}\ \left\langle \exists y \in Q(x).\, \mathbf{t}_a(\bar{z},y) * [\mathrm{G}]_a \right\rangle}$$

The conclusion of the rule specifies that $\mathbb{C}$ performs an atomic update, transforming the state of the region a (of type $\mathbf{t}$ and with parameters $\bar{z}$) from $x \in X$ to $y \in Q(x)$. The environment may change the state of the region before the atomic update occurs, so long as the state remains in the set X. Any update to a shared region has to be justified by a guard for that region; in the conclusion of the rule, this is the guard resource $[\mathrm{G}]_a$. The first premiss of the rule requires that the update from x to y is permitted by the transition system $\mathcal{T}_{\mathbf{t}}$ for region type $\mathbf{t}$ given guard G; the superscript * indicates, again, the reflexive-transitive closure.

While the first premiss ensures that the update is permitted, the second premiss ensures that $\mathbb{C}$ actually performs the atomic update. This is achieved by providing a *proxy* resource $\left\langle \forall x \in X \rightsquigarrow Q(x) \right|_a^i$ in the precondition, which $\mathbb{C}$ must update to a *witness* resource $\left| x \rightsquigarrow y \right\rangle_a^i$, featured in the postcondition. The proxy resource is conceptually a proxy for $[\mathrm{G}]_a$, in that it permits the update to the region by deriving its authority from the guard resource (which is not present in the premiss). The proxy records the atomic update to be performed, the region a in which it is to be performed, and has a unique identifier i: no two proxies or witnesses with the same identifier can coexist. Since the premiss is not an atomic triple, $\mathbb{C}$ may take multiple steps. However, the proxy resource can only be used once, and in doing so is updated to a witness resource that records the specific update that took place. This guarantees that $\mathbb{C}$ effectively performs an atomic update.

The second key proof rule is the UpdateRegion rule, which deals with using a proxy to update a region. A simplified version of this rule is as follows:

$$\frac{a:i \vdash \forall x \in X. \left\langle I(\mathbf{t}_a(\bar{z},x)) * p(x) \right\rangle \ \mathbb{C}\ \left\langle \exists y \in Q(x).\, \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} \left(I(\mathbf{t}_a(\bar{z},y)) * q(x,y)\right) \right\rangle}{a:i \vdash \forall x \in X. \left\langle \mathbf{t}_a(\bar{z},x) * p(x) \right\rangle \ \mathbb{C}\ \left\langle \exists y \in Q(x).\, \mathbf{t}_a(\bar{z},y) * q(x,y) \right\rangle}$$

In this rule, the region $\mathbf{t}_a(\bar{z},x)$ in the conclusion is replaced by its interpretation $I(\mathbf{t}_a(\bar{z},x))$ in the premiss. This allows the resources from the region to be accessed for the duration of an atomic update. To account for the change to the abstract state of the region (from x to y), it is necessary to update a proxy to a witness corresponding to the update. This is achieved by the $\underset{x \rightsquigarrow y}{\overset{a}{\Diamond}}$ modality, which appears in the postcondition of the premiss. Intuitively, the assertion $\underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} P$ should be read as "it is possible to obtain P by updating proxy resources to witnesses, corresponding to the update from x to y". The UpdateRegion rule subsumes the rule with the same name from TaDA, but also the OpenRegion rule, as OpenRegion is now just a special case of the new rule.

$$P \implies \underset{x \rightsquigarrow x}{\overset{a}{\Diamond}} P \tag{8.1}$$

$$\left(\underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} P\right) * R \implies \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} (P * R) \tag{8.2}$$

$$\forall x \in X, y \in Q(x).\, \langle \forall\!\!\!\forall z \in X \rightsquigarrow Q(z) \rvert_a^i \implies \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} \lvert x \rightsquigarrow y \rangle_a^i \tag{8.3}$$

$$\underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} \underset{y \rightsquigarrow z}{\overset{a}{\Diamond}} P \implies \underset{x \rightsquigarrow z}{\overset{a}{\Diamond}} P \tag{8.4}$$

$$\underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} (P \lor Q) \iff \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} P \lor \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} Q \tag{8.5}$$

$$\exists z.\, \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} P(z) \iff \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} (\exists z.\, P(z)) \tag{8.6}$$

Figure 8.1: Axioms for proxy modalities.

Figure 8.1 gives axioms for working with the proxy modality. Axiom (8.1) means that we do not need to update a proxy to perform the identity transition. Axiom (8.2) means that we can move a frame across the proxy modality. Axiom (8.3) allows us to introduce a proxy modality by updating a suitable proxy to a witness corresponding to the required update; this is the basic rule for using proxies. Axiom (8.4) allows us to chain proxies together transitively: if we have a proxy that allows a transition from x to y and another that allows a transition from y to z, then we can use both together to transition from x to z. Finally, axioms (8.5) and (8.6) mean that the proxy modality distributes over disjunction and existential quantification.

The proxies and witnesses of TaDA 2 play a similar role to the atomic tracking resources of TaDA. The most important difference here, however, is that they can be transferred between threads. This allows one thread to perform the atomic action on behalf of another, by using its proxy.

In the ticket lock, such helping occurs in the `release` operation. Conceptually, `release` releases the lock by incrementing the `owner` counter. However, if another thread is waiting to acquire the lock, and has incremented the `next` counter, the lock does not in fact enter the unlocked state. We resolve this incongruity by viewing the atomic increment to the `owner` counter as performing both the unlock action on behalf of its own thread and the lock action on behalf of the waiting thread. That is, the unlocking thread *helps* the locking thread to acquire the lock.

The shared state of the **TLock** region, therefore, holds proxies and witnesses for threads that are waiting to acquire the lock. We define an auxiliary predicate T to represent the shared resources belonging to the **TLock** region in Figure 8.2.

Lines (8.7) and (8.8) state that the `next` and `owner` counters are at addresses y and z, and have values nxt and own. Line (8.9) gives the guard resource $[\textsc{Queue}(S)]_r$, tracking tickets currently held by threads waiting to acquire the lock. A thread that increments the `next` counter from k to $k+1$ will acquire a ticket resource $[\textsc{Ticket}(k,i)]_r$; by doing so, the pair (k,i) is added to S. In the process, a thread gives up its proxy resource (with identifier i) so that another thread can lock on its behalf. The condition on S asserts that there is a ticket for every thread that incremented the `next` counter but has yet to acquire the lock, and that all tickets are below nxt (the next available ticket). Line (8.10) states that there are witnesses for each ticket that successfully acquired the lock, but has not yet reclaimed its witness. Line (8.11) states that there are proxies for each ticket waiting to acquire the lock.

$$T(a, x, s, t, y, z, nxt, own, S) \triangleq x.\texttt{next} \mapsto y * \mathsf{Counter}(s, y, nxt) \tag{8.7}$$
$$* x.\texttt{owner} \mapsto z * \mathsf{Counter}(t, z, own) \tag{8.8}$$
$$* [\textsc{Queue}(S)]_a * \{j \mid own < j < nxt\} \subseteq proj_1(S) \subseteq \{j \mid j < nxt\} \tag{8.9}$$
$$* \circledast_{(v,i) \in S} \left(v \leq own \implies |(0, v) \rightsquigarrow (1, v)\rangle_a^i\right) \tag{8.10}$$
$$* \circledast_{(v,i) \in S} \left(v > own \implies \langle \forall (l, n) \in \{0, 1\} \times \mathbb{N} \rightsquigarrow (1, n) \wedge b = 0|_a^i\right) \tag{8.11}$$

Figure 8.2: Auxiliary predicate describing the shared resources of the ticket lock.

We define the interpretation of abstract states for the ticket lock region:

$$I(\mathbf{TLock}_a(x, s, t, y, z, (l, n))) \triangleq \exists m, S.\, T(a, s, t, x, y, z, m, n, S) * m \geq n * (l = 1 \iff m > n)$$

We can now give the interpretation of the abstract types and abstract predicate as follows:

$$\mathbb{T}_2 \triangleq \mathsf{RId} \times \mathbb{T}_1 \times \mathbb{T}_1 \times \mathsf{Addr} \times \mathsf{Addr}$$
$$\mathbb{T}_3 \triangleq \mathbb{N}$$
$$\mathsf{TLock}((a, s', t', y, z), x, l, n) \triangleq \mathbf{TLock}_a(x, s', t', y, z, (l, n)) * [\mathrm{G}]_r$$

where RId is the set of region identifiers and Addr is the set of addresses.

It remains to prove that the implementations of the operations satisfy the specifications, given this interpretation. The proof for `acquire` is given in Figure 8.3, and the proof for `release` in Figure 8.4.

8.1.2 Two-step Counter

Speculation can be observed in many fine-grained implementations, such as the Michael-Scott queue [43] or Heller *et al.* lazy set [24]. However, we do not need to look at complex implementations to find speculation. We propose a simple counter module implemented using the counter previously presented:

```
function readVal(x) {        function incrVal(x) {
  v := read(x);                incr(x);
  return v / 2;                incr(x);
}                            }
```

Atomic Specification

The `readVal` operation consists of reading the current value of the counter and returns the floor of its value divided by 2. The `incrVal` operation performs two increments, where each increment is atomic. This means that in between each increment, other threads can perform reads or other increments. We

Figure 8.3: Proof of correctness of the `acquire` operation.

specify the counter module in a similar way as the counter we use to implement it, as follows:

$$\vdash \left\{\mathsf{True}\right\}\ \texttt{makeCounter()}\ \left\{\exists s \in \mathbb{T}_2.\, \mathsf{TSCounter}(s, \mathsf{ret}, 0)\right\}$$

$$\vdash \mathbb{V}n \in \mathbb{N}. \left\langle \mathsf{TSCounter}(s, \mathtt{x}, n)\right\rangle\ \texttt{readVal(x)}\ \left\langle \mathsf{TSCounter}(s, \mathtt{x}, n) * \mathsf{ret} = n\right\rangle$$

$$\vdash \mathbb{V}n \in \mathbb{N}. \left\langle \mathsf{TSCounter}(s, \mathtt{x}, n)\right\rangle\ \texttt{incrVal(x)}\ \left\langle \mathsf{TSCounter}(s, \mathtt{x}, n+1)\right\rangle$$

Implementation

We use the notation $\mathbb{S}_a(X)$, where a is a region identifier and X is a set of configurations (lists containing proxies and witnesses) to denote that we can speculatively choose between any of the configurations. When we create a region, we have $\mathbb{S}_a(\{\emptyset\})$—only the empty configuration. If we have a proxy or witness separate from $\mathbb{S}_a(X)$, then we can merge it in by adding it to all of the configurations. We can go from $\mathbb{S}_a(X)$ to $\mathbb{S}_a(Y)$ if $Y \subseteq X$. If we can pull out a specific proxy or witness from every configuration in the set, then we can pull it out from the overall speculation.

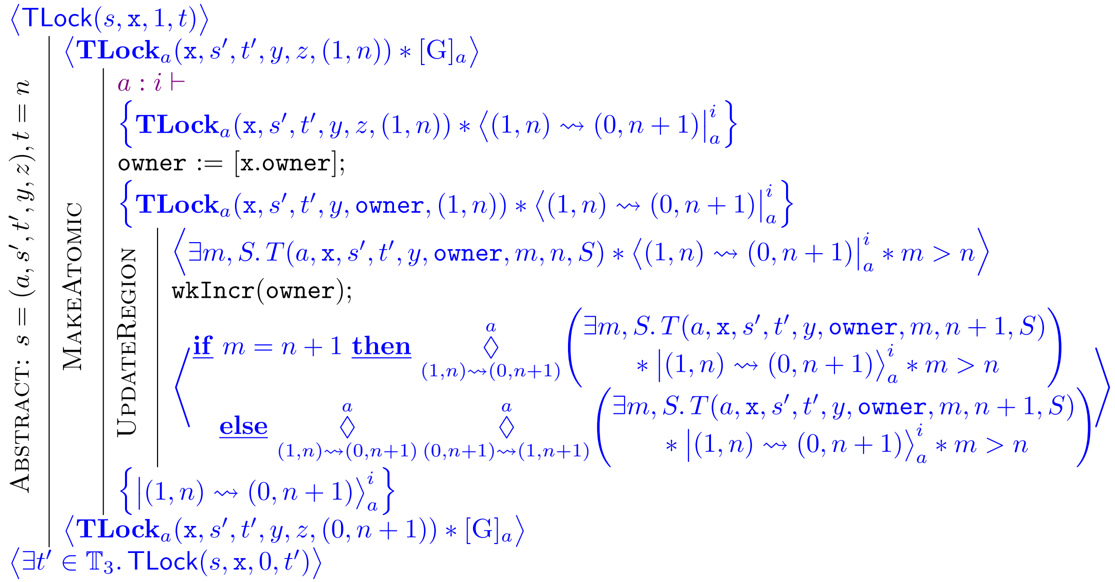


Figure 8.4: Proof of correctness of the `release` operation.

$$C_a(X,0,v) \stackrel{\text{def}}{=} \left\{ \circledast_{i \in X} \langle \forall n \in \mathbb{N}.\, n \rightsquigarrow n+1 |_a^i \right\}$$

$$C_a(X,m+1,v) \stackrel{\text{def}}{=} \left\{ |v-m-1 \rightsquigarrow v-m\rangle_a^i * x \;\middle|\; i \in X, x \in C_a(X \setminus \{i\}, m, v) \right\}$$

Figure 8.5: Auxiliary predicate describing the set of configurations for the two-step counter.

To describe the set of configurations for the two-step counter, we define $C_a(X, m, v)$ in Figure 8.5 where X is a set of proxy/witness configurations, $0 \leq m \leq |X|$ is the number of witnesses to have and v is the current value of the counter.

We extend the proxy modality to handle speculative sets of proxy/witnesses with the axiom:

$$\frac{\forall p' \in P'.\, \exists p \in P.\, p \,\mathsf{R}_{x,y}\, p'}{\mathbb{S}_a\big(P\big) \implies \underset{x \rightsquigarrow y}{\overset{a}{\Diamond}} \mathbb{S}_a\big(P'\big)}$$

The premiss requires that we justify the update for each element in the set. Essentially, if we update the region, we have enough proxies to justify such an update in all configurations. The relation R enforces this and is defined as follows:

$$\frac{}{p \,\mathsf{R}_{x,x}\, p} \tag{8.12}$$

$$\frac{p \,\mathsf{R}_{x,y}\, p' \qquad p' \,\mathsf{R}_{y,z}\, p''}{p \,\mathsf{R}_{x,z}\, p''} \tag{8.13}$$

$$\frac{p \,\mathsf{R}_{x,y}\, p'}{p * r \,\mathsf{R}_{x,z}\, p' * r} \tag{8.14}$$

$$\frac{\forall x \in X, y \in Q(x)}{\langle \forall z \in X \rightsquigarrow Q(z)|_a^i \,\mathsf{R}_{x,y}\, |x \rightsquigarrow y\rangle_a^i} \tag{8.15}$$

The relation resembles the axioms for proxies, where (8.12) means that we do not need to update a

proxy to perform the identity transition. (8.13) allows us to chain proxies together transitively: if we have a proxy that allows a transition from x to y and another that allows a transition from y to z, then we can use both together to transition from x to z. (8.14) means that we can frame resources. Finally, (8.15) allows us to update a suitable proxy to a witness corresponding to the required update.

Note that the MakeAtomic rules enforces that the speculation for an operation must be resolved before the operation terminates.

For the two-step counter, we introduce a region with type name **TSCounter**. This region has three types of guard resources. The unique non-empty guard Inc provides the capability to increment the counter. The second type of guard resource is $\textsc{Active}(X)$, which tracks the set X of proxy or witness identifiers of the threads incrementing the counter; these are the threads which have performed the first increment, but not the second. The last type of guard resource is the $\textsc{Inc}(i)$, which expresses that the identifier i exists in the set X described by the guard $\textsc{Active}(X)$. We wish to allow threads to add and remove identifiers from the set X. We enforce this using the following equivalence:

$$\textsc{Active}(X) = \textsc{Active}(X \uplus \{i\}) \bullet \textsc{Inc}(i)$$

The possible states of **TSCounter** are natural numbers, representing the value of the counter. All transitions over abstract states of the region are guarded by Inc, which has the following labelled transition system:

$$\textsc{Inc} : \forall n \in \mathbb{N}.\, n \rightsquigarrow n + 1$$

The transition system requires that every update must increase the value of the counter. This is necessary to guarantee the correctness of the `incrVal` operation.

$$I(\mathbf{TSCounter}_a(s', x, n)) \triangleq \exists m, X.\, \mathsf{Counter}(s', x, m) * [\textsc{Active}(X)]_a * \mathbb{S}_a\left(C_a\left(X, \left\lfloor\frac{|X|}{2}\right\rfloor, \left\lfloor\frac{m}{2}\right\rfloor\right)\right) * (m = 2n \vee m = 2n + 1)$$

The interpretation enforces that the threads performing increments ($|X|$ of them) delay the choice of picking which increment corresponds to which thread until after they have finished.

We now define the interpretation for the abstract types and abstract predicates:

$$\mathbb{T}_2 \triangleq \mathsf{RId}$$
$$\mathsf{TSCounter}((r, s'), x, n) \triangleq \mathbf{TSCounter}_a(s', x, n) * [\textsc{Inc}]_r$$

Proofs for the `readVal` and `incrVal` operations are shown in Figure 8.6 and Figure 8.7.

8.2 Case Study: Michael-Scott Queue

We now consider the `java.util.concurrent` package [21] variant of the Michael-Scott queue [43] to illustrate how we can apply this technique to verify larger modules.

$$\forall n \in \mathbb{N}.$$
$$\langle \mathsf{TSCounter}(s, \mathtt{x}, n) \rangle$$
ABSTRACT: $s = (a, s')$
$$\forall n \in \mathbb{N}.$$
$$\langle \mathbf{TSCounter}_a(s', \mathtt{x}, n) * [\mathrm{INC}]_a \rangle$$
MAKEATOMIC
$$a : i \vdash$$
$$\left\{ \exists n.\, \mathbf{TSCounter}_a(s', \mathtt{x}, n) * \langle \forall n \in \mathbb{N}.\, n \rightsquigarrow n|_a^i \right\}$$
UPDATEREGION
$$\forall n \in \mathbb{N}.$$
$$\left\langle \begin{array}{l} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m) * [\mathrm{ACTIVE}(X)]_a * \mathbb{S}_a\left(C_a\left(X, \left\lfloor \frac{|X|}{2} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right)\right) \\ * (m = 2n \vee m = 2n+1) * \langle \forall n \in \mathbb{N}.\, n \rightsquigarrow n|_a^i \end{array} \right\rangle$$
```
v := read(x);
```
$$\left\langle \overset{a}{\underset{n \rightsquigarrow n}{\Diamond}} \left(\begin{array}{c} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m) * [\mathrm{ACTIVE}(X)]_a * \mathbb{S}_a\left(C_a\left(X, \left\lfloor \frac{|X|}{2} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right)\right) \\ * (m = 2n \vee m = 2n+1) * |n \rightsquigarrow n\rangle_a^i * \mathtt{v} = m \end{array} \right) \right\rangle$$
$$\left\{ \exists n.\, (\mathtt{v} = 2n \vee \mathtt{v} = 2n+1) * |n \rightsquigarrow n\rangle_a^i \right\}$$
```
return v / 2;
```
$$\langle \mathbf{TSCounter}_a(s', \mathtt{x}, n) * [\mathrm{INC}]_a * \mathsf{ret} = n \rangle$$
$$\langle \mathsf{TSCounter}(s, \mathtt{x}, n) * \mathsf{ret} = n \rangle$$

Figure 8.6: Proof of correctness of the `readVal` operation.

$$\forall n \in \mathbb{N}.$$
$$\langle \mathsf{TSCounter}(s, \mathtt{x}, n) \rangle$$
ABSTRACT: $s = (a, s')$
$$\langle \mathbf{TSCounter}_a(s', \mathtt{x}, n) * [\mathrm{INC}]_a \rangle$$
MAKEATOMIC
$$a : i \vdash$$
$$\left\{ \exists n.\, \mathbf{TSCounter}_a(s', \mathtt{x}, n) * \langle \forall n \in \mathbb{N}.\, n \rightsquigarrow n+1|_a^i \right\}$$
UPDATEREGION
$$\forall n \in \mathbb{N}.$$
$$\left\langle \begin{array}{l} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m) * [\mathrm{ACTIVE}(X)]_a * \mathbb{S}_a\left(C_a\left(X, \left\lfloor \frac{|X|}{2} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right)\right) \\ * (m = 2n \vee m = 2n+1) * \langle \forall n \in \mathbb{N}.\, n \rightsquigarrow n+1|_a^i \end{array} \right\rangle$$
```
incr(x);
```
$$\left\langle \overset{a}{\underset{\lfloor \frac{m}{2} \rfloor \rightsquigarrow \lfloor \frac{m+1}{2} \rfloor}{\Diamond}} \left(\begin{array}{c} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m+1) * [\mathrm{ACTIVE}(X \uplus \{i\})]_a \\ * \mathbb{S}_a\left(C_a\left(X \uplus \{i\}, \left\lfloor \frac{|X \uplus \{i\}|}{2} \right\rfloor, \left\lfloor \frac{m+1}{2} \right\rfloor\right)\right) \\ * (m+1 = 2n+1 \vee m+1 = 2(n+1)) * [\mathrm{INC}(i)]_a \end{array} \right) \right\rangle$$
$$\{ \exists n.\, \mathbf{TSCounter}_a(s', \mathtt{x}, n) * [\mathrm{INC}(i)]_a \}$$
UPDATEREGION
$$\forall n \in \mathbb{N}.$$
$$\left\langle \begin{array}{l} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m) * [\mathrm{ACTIVE}(X \uplus \{i\})]_a * \mathbb{S}_a\left(C_a\left(X \uplus \{i\}, \left\lfloor \frac{|X \uplus \{i\}|}{2} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right)\right) \\ * (m = 2n \vee m = 2n+1) * [\mathrm{INC}(i)]_a \end{array} \right\rangle$$
```
incr(x);
```
$$\left\langle \overset{a}{\underset{\lfloor \frac{m}{2} \rfloor \rightsquigarrow \lfloor \frac{m+1}{2} \rfloor}{\Diamond}} \left(\begin{array}{c} \exists m, X.\, \mathsf{Counter}(s', \mathtt{x}, m+1) * [\mathrm{ACTIVE}(X)]_a * \mathbb{S}_a\left(C_a\left(X, \left\lfloor \frac{|X|}{2} \right\rfloor, \left\lfloor \frac{m+1}{2} \right\rfloor\right)\right) \\ * (m+1 = 2n+1 \vee m+1 = 2(n+1)) * \exists v.\, |v \rightsquigarrow v+1\rangle_a^i \end{array} \right) \right\rangle$$
$$\left\{ \exists v.\, |v \rightsquigarrow v+1\rangle_a^i \right\}$$
$$\langle \mathbf{TSCounter}_a(s', \mathtt{x}, n+1) * [\mathrm{INC}]_a \rangle$$
$$\langle \mathsf{TSCounter}(s, \mathtt{x}, n+1) \rangle$$

Figure 8.7: Proof of correctness of the `incrVal` operation.

8.2.1 Atomic Specification

We start by giving an abstract specification for a non-blocking queue as follows:

$$\vdash \left\{\mathsf{True}\right\} \texttt{makeQueue}() \left\{\exists s \in \mathbb{T}_1, t \in \mathbb{T}_2.\, \mathsf{Queue}(s, \mathsf{ret}, [], t)\right\}$$

$$\vdash \mathbb{W}(vs, t) \in \mathbb{L}_1 \times \mathbb{T}_2. \quad \left\langle \mathsf{Queue}(s, \mathtt{x}, vs, t) * \mathtt{v} \neq 0 \right\rangle$$
$$\texttt{enqueue}(\mathtt{x}, \mathtt{v})$$
$$\left\langle \exists vs', t'.\, \mathsf{Queue}(s, \mathtt{x}, vs', t') * vs' = vs +\!\!+ [\mathtt{v}] \right\rangle$$

$$\vdash \mathbb{W}(vs, t) \in \mathbb{L}_1 \times \mathbb{T}_2. \quad \left\langle \mathsf{Queue}(s, \mathtt{x}, vs, t) \right\rangle$$
$$\texttt{dequeue}(\mathtt{x})$$
$$\left\langle \begin{array}{l} \underline{\textbf{if}}\ vs = []\ \underline{\textbf{then}}\ \mathsf{Queue}(s, \mathtt{x}, vs, t) * \mathsf{ret} = 0 \\ \underline{\textbf{else}}\ \exists vs', t'.\, \mathsf{Queue}(s, \mathtt{x}, vs', t') * vs = \mathsf{ret} : vs' \end{array} \right\rangle$$

We represent the queue by the abstract predicate $\mathsf{Queue}(s, x, vs, t)$. It asserts that there is a queue at address x and with contents vs, where $vs \in \mathbb{L}_1$, and $\mathbb{L}_1$ is a list containing non-zero integers. The parameters $s \in \mathbb{T}_1$ and $t \in \mathbb{T}_2$ capture implementation-specific information of the queue.

The constructor returns an empty queue. The `enqueue` operation atomically adds an element to the end of the queue; this element cannot be 0. This restricts all the elements in the queue from being equal to 0. The `dequeue` operation atomically removes an element from the front of the queue, if one is available (*i.e.* the queue is not empty); otherwise it will return 0.

8.2.2 Implementation

The queue, given in Figure 8.8, is represented as a non-empty linked list. The first node in the list is considered to be outside of the queue. Its value is not part of the abstract state of the queue. Additionally, there are two anchor pointers: the head and the tail. The head pointer always points to the sentinel node. The sentinel node is a node that is not part of the queue, but points to the node that contains the first element of the queue. The tail pointer points to a node that is or was in the queue, as long as the true tail is accessible from that node.

An `enqueue` operation appends nodes to the end of the list, by updating the next pointer of the true tail from 0 to the address of the new node. A `dequeue` operation removes nodes from the front of the queue by updating the head pointer with a compare-and-set operation to the next node from the sentinel node. If the queue is empty, then the sentinel node in the list is the only node accessible from the head pointer and its next pointer has value 0. Note that when a `dequeue` operation removes a node from the queue, it is still possible to reach any node in the queue from that node. Moreover, we set the value of the node to 0 after removing it from the queue. This is an optimisation to improve garbage collection in the original algorithm and does not affect the correctness of the implementation.

In the `enqueue` operation, the atomic update always happens if `CAS(t.next, s, n)` succeeds. Otherwise, the algorithm retries. There are additional compare-and-set operations used to update the tail pointer. The outcome of such operations does not affect the abstract state of the queue.

In the `dequeue` operation, if the queue is not empty and it succeeds in advancing the sentinel node, the atomic update occurs on the `CAS(x.head, h, first)`. The reading and nullifying the value of the node does not affect the abstract state of the queue. If the queue is empty, the `dequeue` operation's atomic

```
function makeQueue() {
  n := alloc(2);
  [n.value] := 0;
  [n.next] := 0;
  x := alloc(2);
  [x.head] := n;
  [x.tail] := n;
  return x;
}

function enqueue(x, v) {
  n := alloc(2);
  [n.value] := v;
  [n.next] := 0;
  while (true) {
    t := [x.tail];
    s := [t.next];
    t' := [x.tail];
    if (t = t') {
      if (s = 0) {
        b := CAS(t.next, s, n);
        if (b = 1) {
          CAS(x.tail, t, n)
          return
        }
      } else {
        CAS(x.tail, t, s)
      }
    }
  }
}

function dequeue(x) {
  while (true) {
    h := [x.head];
    t := [x.tail];
    first := [h.next];
    h' := [x.head];
    if (h = h') {
      if (h = t) {
        if (first = 0) {
          return 0;
        } else {
          CAS(x.tail, t, first)
        }
      } else {
        b := CAS(x.head, h, first);
        if (b = 1) {
          v := [first.value];
          if (v ≠ 0) {
            [first.value] := 0;
            return v;
          }
        }
      }
    }
  }
}
```

where $\mathbb{E}.\mathtt{head} \stackrel{\text{def}}{=} \mathbb{E}$ $\quad \mathbb{E}.\mathtt{tail} \stackrel{\text{def}}{=} \mathbb{E}+1$ $\quad \mathbb{E}.\mathtt{value} \stackrel{\text{def}}{=} \mathbb{E}$ $\quad \mathbb{E}.\mathtt{next} \stackrel{\text{def}}{=} \mathbb{E}+1$.

Figure 8.8: Michael-Scott Queue implementation.

update depends on a future event. In particular, when the operation performs `first := [h.next];`, there are two cases: either `h` value matches the current head of the queue, or it doesn't. We can only be sure that `h` matches the head of the queue when the queue is empty, and in that case, `first` has value 0. However, we cannot determine if the atomic update occurs at this particular point, as it is dependent on the next read `h' := [x.head];` getting the same value as `h`. If it is not, then the entire operation needs to start over.

We want to prove the correctness of the implementation. In both operations, the atomic updates are always performed by the thread performing the operation. There is no helping involved for the Michael-Scott queue. However, the `dequeue` operation can experience speculation when dequeuing from an empty queue. This happens because we cannot determine if the operation will decide to retry or not, until a future operation is performed.

In order to prove the correctness of the queue, we define a region type **MSQueue**. The type is parametrised by the address of the queue and its abstract state. The abstract state of **MSQueue** regions consists of a pair of lists $(ds, ns) \in \mathbb{L}_2 \times \mathbb{L}_3$, where ds is a list of nodes that have been in the queue plus the initial sentinel node and ns is the list of nodes containing the values currently in the queue. We define their types as $\mathbb{L}_2 \triangleq (\mathsf{Addr} \times \mathbb{Z} \times \mathsf{Addr} \uplus \{0\})^+$ and $\mathbb{L}_3 \triangleq (\mathsf{Addr} \times \{v \in \mathbb{Z} \mid v \neq 0\} \times \mathsf{Addr} \uplus \{0\})^*$. Each element of each of the lists contains a node's address, its value and next pointer. Note that the

ds list cannot be empty, since its last element is the sentinel node. Furthermore, the list ns cannot contain empty values. It is important to the correctness of the algorithm that nodes that have been removed from the queue cannot reappear again. To account for this, we track all the nodes that have been previously removed from the head of the queue in the ds list. The protocol that governs how the **MSQueue** region can be updated is specified by the transition system as follows:

$$\begin{array}{ll}
\mathrm{G} : \forall n, m, w, ds. & (ds +\!\!+ [(n, v, 0)], []) \rightsquigarrow (ds +\!\!+ [(n, v, m)], [(m, w, 0)]) \\
\mathrm{G} : \forall n, m, w, ds, ns. & (ds, ns +\!\!+ [(n, v, 0)]) \rightsquigarrow (ds, ns +\!\!+ [(n, v, m)] +\!\!+ [(m, w, 0)]) \\
\mathrm{G} : \forall n, m, ds, ns. & (ds, (n, v, m) : ns) \rightsquigarrow (ds +\!\!+ [(n, v, m)], ns)
\end{array}$$

There is a single guard G that gives threads permissions to enqueue and dequeue. The first two transitions allow a thread to enqueue a new value, in the case the queue is empty and otherwise. The last transition allows a thread to dequeue a value, by moving the node from the list ns to the list ds.

In order to track operations that have observed the queue as empty, but have not yet decided what action to take, we introduce two guards: $\textsc{Empty}(i)$ and $\textsc{Deq}(S)$. The guard $\textsc{Empty}(i)$, where $i \in \mathsf{TId}$, states the thread i has observed the queue as empty when performing a `dequeue` operation. The guard $\textsc{Deq}(S)$ accounts for the $\textsc{Empty}(i)$ guards that have been issued, where $S \subseteq \mathsf{TId}$. These guards are defined to obey the following equation:

$$\textsc{Deq}(S) = \textsc{Deq}(S \uplus \{i\}) \bullet \textsc{Empty}(i) \qquad \text{if } i \notin \mathit{proj}_1(S)$$

When the `dequeue` succeeds in removing a node from the queue, it nullifies the value of a node. We need to guarantee that the only thread doing it is the thread that actually removes the node from the queue. Otherwise, another operation could potentially set the value of the node to 0 before the thread dequeuing read its value. For this purpose, we introduce a guard $\textsc{Value}(n, v)$, where $n \in \mathsf{Addr}$ and $v \in \{v \in \mathbb{Z} \mid v \neq 0\}$, that tracks the value v of the node n when it is part of the queue, and prevents other threads from manipulating its value. Moreover, we have an additional guard $\textsc{Out}(S)$ to track how many $\textsc{Value}(n, v)$ have been issued. The guards must satisfy the following equation:

$$\textsc{Out}(S) = \textsc{Out}(S \uplus \{(n, v)\}) \bullet \textsc{Value}(n, v) \;\; \text{if } n \notin \mathit{proj}_1(S)$$

We now introduce auxiliary predicates to represent both lists, as well as the data structure of the queue in Figure 8.9. The $\mathsf{node}(n, v, m)$ represents a node at address n, with value v and the next pointer with value m. The $\mathsf{outList}(a, x, ds, y)$ represents the ds linked list, where the sentinel node when the queue was created has address x, and the most recent node in the list has a next pointer with value y. We require that the nodes that have been nullified must have the guard $\textsc{Value}(x, v)$ to record the initial value. The $\mathsf{inList}(x, ns)$ represents the ns linked list, where the last node of the ds list points to its first node with value x, and the last node of the list always has next pointer with value 0. The predicate $\mathsf{queue}(a, x, ds, ns)$ represents the data structure of the queue: The `head` pointer points to the last node of the list ds, and the `tail` pointer points either to the last element of the list ds or one node in the list ns[1]. Moreover, we enforce the ds list to have at least one node and that its last node has a next pointer to the first element of the ns list. The final predicate is used in the proofs

[1] This contrasts the original Michael-Scott queue algorithm, where the tail pointer points to one of the last two nodes.

$$\mathsf{node}(n, v, m) \stackrel{\text{def}}{=} n.\mathtt{value} \mapsto v * n.\mathtt{next} \mapsto m$$

$$\mathsf{outList}(a, x, ds, y) \stackrel{\text{def}}{=} (x = y * ds = []) \vee \exists v, n, ds'. \begin{pmatrix} (\mathsf{node}(x, v, n) \vee \mathsf{node}(x, 0, n) * [\textsc{Value}(x, v)]_r) \\ * \mathsf{outList}(a, n, ds', y) * ds = (x, v, n) : ds' \end{pmatrix}$$

$$\mathsf{inList}(x, ns) \stackrel{\text{def}}{=} (x = 0 * ns = []) \vee \left(\exists v, n, ns'. \mathsf{node}(x, v, n) * \mathsf{inList}(n, ns') * ns = (x, v, n) : ns'\right)$$

$$\begin{aligned} \mathsf{queue}(a, x, ds, ns) \stackrel{\text{def}}{=} & \exists h, t, y, z, v, ds', S. x.\mathtt{head} \mapsto h * x.\mathtt{tail} \mapsto t * \mathsf{outList}(a, y, ds, z) * \mathsf{inList}(z, ns) \\ & * [\textsc{Out}(S)]_a * S = \{(n, v) \mid (n, v, m) \in ds\} * ds = ds' +\!\!+ [(h, v, z)] \\ & * t \in proj_1([(h, v, z)] +\!\!+ ns) \end{aligned}$$

$$\mathsf{after}(x, y, vs) \stackrel{\text{def}}{=} \exists vs'. vs = v : vs' * (x = v * y \in vs \vee x \neq v * \mathsf{after}(x, y, vs'))$$

Figure 8.9: Auxiliary predicates to describe the shared resources of the Michael-Scott queue.

$$\mathsf{PE}(i) = \left\langle \forall\!\!\!\forall (ds, ns) \rightsquigarrow \begin{array}{c} (ds', ns') \wedge \underline{\textbf{if}}\ ns = [] \\ \underline{\textbf{then}}\ \exists ds''. ds = ds'' +\!\!+ [(n, v, 0)] \wedge ds' = ds'' +\!\!+ [(n, v, m)] \wedge ns' = [(m, w, 0)] \\ \underline{\textbf{else}}\ \exists ns''. ds = ds' \wedge ns = ns'' +\!\!+ [(n, v, 0)] \wedge ns' = ns'' +\!\!+ [(n, v, m), (m, w, 0)] \end{array} \right|_a^i$$

$$\mathsf{WE}(i, m, w, ds, ns, ds', ns') = \left| (ds, ns) \rightsquigarrow \begin{array}{c} (ds', ns') \wedge \underline{\textbf{if}}\ ns = [] \\ \underline{\textbf{then}} \begin{pmatrix} \exists ds''. ds = ds'' +\!\!+ [(n, v, 0)] \wedge ds' = ds'' +\!\!+ [(n, v, m)] \\ \wedge ns' = [(m, w, 0)] \end{pmatrix} \\ \underline{\textbf{else}} \begin{pmatrix} \exists ns''. ds = ds' \wedge ns = ns'' +\!\!+ [(n, v, 0)] \\ \wedge ns' = ns'' +\!\!+ [(n, v, m), (m, w, 0)] \end{pmatrix} \end{array} \right\rangle_a^i$$

$$\mathsf{PD}(i) = \left\langle \forall\!\!\!\forall (ds, ns) \rightsquigarrow \underline{\textbf{if}}\ ns = []\ \underline{\textbf{then}}\ (ds, ns)\ \underline{\textbf{else}}\ (ds +\!\!+ [(n, v, m)], ns') \wedge ns = (n, v, m) : ns' \right|_a^i$$

$$\mathsf{WD1}(i) = \exists ds. \left| (ds, []) \rightsquigarrow (ds, []) \right\rangle_a^i$$

$$\mathsf{WD2}(i, n, v, ds, ns, ds', ns') = \exists m. \left| (ds, ns) \rightsquigarrow (ds', ns') \wedge ns = (n, v, m) : ns' \wedge ds' = ds +\!\!+ [(n, v, m)] \right\rangle_a^i$$

Figure 8.10: Proxies and witnesses for the Michael-Scott queue.

to describe that node with address x proceeds the node with address y in the list vs.

We define auxiliary notation for referring to proxies and witnesses for the enqueue and dequeue operations in Figure 8.10.

In order to describe the set of possible proxies and witnesses when the `dequeue` operation observes an empty queue, but has not yet decided if that is the atomic update, we define the following set:

$$\begin{aligned} D_a(\emptyset) &\stackrel{\text{def}}{=} \{\mathsf{True}\} \\ D_a(X) &\stackrel{\text{def}}{=} \{y * x \mid i \in X, x \in \{\mathsf{PD}(i), \mathsf{WD1}(i)\}, y \in D_a(X \setminus \{i\})\} \end{aligned}$$

The **MSQueue** region type has the following interpretation:

$$I(\mathbf{MSQueue}_a(x, (ds, ns))) = \exists X. \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X)]_a * \mathbb{S}_a\Big(D_a(X)\Big)$$

The abstract predicate $\mathsf{Queue}(s, x, vs, (ds, ns))$ combines the region and the guard:

$$\begin{aligned} \mathbb{T}_1 &\triangleq \mathsf{RId} \\ \mathbb{T}_2 &\triangleq \mathbb{L}_2 \times \mathbb{L}_3 \\ \mathsf{Queue}(a, x, vs, (ds, ns)) &\triangleq \mathbf{MSQueue}_a(x, (ds, ns)) * [\mathrm{G}]_a * vs = proj_2(ns) \end{aligned}$$

The proof of the `dequeue` operation is given in Figure 8.11.

When the queue is empty, we speculatively execute the atomic update, and keep the guard $\textsc{Empty}(i)$ to track it. Note that the node pointed by the `head`, must always be of the *ds* list. This means that if its next pointer is 0, then it must be the case that the queue is empty. If the following read value of the head pointer matches the previously read, we collapse the speculative state and retrieve the witness. Otherwise, we retrieve the original proxy from the speculative component and try to update the `tail` of the queue. Updating the tail does not affect the abstract state. If the queue was not empty, then we attempt to update the `head` to the next node using a compare-and-set. If successful, the atomic update occurs and we hold the guard $\textsc{Value}(n, v)$ to prevent other threads from nullifying the original value in the node. We then the read the value from the node and set its value to 0 before returning.

The proof of the `enqueue` operation is omitted, as it is very similar to the proof for `dequeue`.

$\mathbb{V}(vs, t) \in \mathbb{L}_1 \times \mathbb{T}_2.$

$\big\langle \mathsf{Queue}(s, \mathtt{x}, vs, t) \big\rangle$

ABSTRACT: $s = a, t = (ds, ns)$

$\quad \mathbb{V}(ds, ns) \in \mathbb{L}_2 \times \mathbb{L}_3.$

$\quad \big\langle \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * [\mathrm{G}]_a * vs = proj_2(ns) \big\rangle$

MAKEATOMIC

$\quad\quad a : i \vdash$

$\quad\quad$ `while (true) {`

$\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) \big\}$

$\quad\quad\quad$ `h :=` $[$`x.head`$]$`; t :=` $[$`x.tail`$]$`;`

$\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) * \mathtt{h} \in proj_1(ds) * (\mathtt{h} = \mathtt{t} \lor \mathsf{after}(\mathtt{h}, \mathtt{t}, proj_1(ds +\!\!+ ns))) \big\}$

UPDATEREGION

$\quad\quad\quad\quad \mathbb{V}(ds, ns) \in \mathbb{L}_2 \times \mathbb{L}_3.$

$\quad\quad\quad\quad \left\langle \begin{array}{l} \exists X.\, \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X)]_a * \mathbb{S}_a\big(D_a(X)\big) * \mathsf{PD}(i) * \mathtt{h} \in proj_1(ds) \\ * (\mathtt{h} = \mathtt{t} \lor \mathsf{after}(\mathtt{h}, \mathtt{t}, proj_1(ds +\!\!+ ns))) \end{array} \right\rangle$

$\quad\quad\quad\quad$ `first :=` $[$`h.next`$]$`;`

$\quad\quad\quad\quad \left\langle \begin{array}{l} \underline{\textbf{if}}\ \mathtt{first} = 0\ \underline{\textbf{then}} \overset{a}{\underset{(ds,[]) \rightsquigarrow (ds,[])}{\Diamond}} \left(\begin{array}{c} \exists X.\, \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X \uplus \{i\})]_a * \mathbb{S}_a\big(D_a(X \uplus \{i\})\big) \\ * [\textsc{Empty}(i)] * \mathtt{h} \in proj_1(ds) * \mathtt{h} = \mathtt{t} \end{array} \right) \\ \quad \underline{\textbf{else}} \overset{a}{\underset{(ds,ns) \rightsquigarrow (ds,ns)}{\Diamond}} \left(\begin{array}{l} \exists X.\, \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X)]_a * \mathbb{S}_a\big(D_a(X)\big) * \mathsf{PD}(i) * \mathtt{h} \in proj_1(ds) \\ * (\mathtt{h} = \mathtt{t} \lor \mathsf{after}(\mathtt{h}, \mathtt{t}, proj_1(ds +\!\!+ ns))) * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \end{array} \right) \end{array} \right\rangle$

$\quad\quad\quad \left\{ \begin{array}{l} \underline{\textbf{if}}\ \mathtt{first} = 0\ \underline{\textbf{then}}\ \big(\exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * [\textsc{Empty}(i)] * \mathtt{h} \in proj_1(ds) * \mathtt{h} = \mathtt{t}\big) \\ \quad \underline{\textbf{else}} \left(\begin{array}{c} \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) * \mathtt{h} \in proj_1(ds) \\ * (\mathtt{h} = \mathtt{t} \lor \mathsf{after}(\mathtt{h}, \mathtt{t}, proj_1(ds +\!\!+ ns))) * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \end{array} \right) \end{array} \right\}$

$\quad\quad\quad$ `h' :=` $[$`x.head`$]$`;`

$\quad\quad\quad$ `if (h = h') {`

$\quad\quad\quad\quad$ `if (h = t) {`

$\quad\quad\quad\quad\quad \left\{ \begin{array}{l} \underline{\textbf{if}}\ \mathtt{first} = 0\ \underline{\textbf{then}}\ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * [\textsc{Empty}(i)] * \mathtt{h} \in proj_1(ds) * \mathtt{h} = \mathtt{t} \\ \quad \underline{\textbf{else}}\ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) * \mathtt{h} \in proj_1(ds) * \mathtt{h} = \mathtt{t} * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \end{array} \right\}$

$\quad\quad\quad\quad\quad$ `if (first = 0) {`

$\quad\quad\quad\quad\quad\quad \big\{ \mathsf{WD1}(i) \big\}$ // open region and swap $[\textsc{Empty}(i)]$ by the witness.

$\quad\quad\quad\quad\quad\quad$ `return 0;`

$\quad\quad\quad\quad\quad$ `} else {`

$\quad\quad\quad\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) * \mathtt{h} = \mathtt{t} * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \big\}$

$\quad\quad\quad\quad\quad\quad$ `CAS(x.tail, t, first);`

$\quad\quad\quad\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) \big\}$

$\quad\quad\quad\quad\quad$ `}`

$\quad\quad\quad\quad$ `} else {`

$\quad\quad\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \big\}$

UPDATEREGION

$\quad\quad\quad\quad\quad\quad \mathbb{V}(ds, ns) \in \mathbb{L}_2 \times \mathbb{L}_3.$

$\quad\quad\quad\quad\quad\quad \big\langle \exists X.\, \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X)]_a * \mathbb{S}_a\big(D_a(X)\big) * \mathsf{PD}(i) * \mathsf{after}(\mathtt{h}, \mathtt{first}, proj_1(ds +\!\!+ ns)) \big\rangle$

$\quad\quad\quad\quad\quad\quad$ `b := CAS(x.head, h, first);`

$\quad\quad\quad\quad\quad\quad \left\langle \begin{array}{l} \underline{\textbf{if}}\ \mathtt{b} = 1\ \underline{\textbf{then}}\ \exists ds', ns', v. \overset{a}{\underset{(ds,ns) \rightsquigarrow (ds',ns')}{\Diamond}} \left(\begin{array}{c} \exists X.\, \mathsf{queue}(a, x, ds', ns') * [\textsc{Deq}(X)]_a * \mathbb{S}_a\big(D_a(X)\big) \\ * \mathsf{WD2}(i, \mathtt{first}, v, ds, ns, ds', ns') * [\textsc{Value}(\mathtt{first}, v)]_a \end{array} \right) \\ \quad \underline{\textbf{else}} \overset{a}{\underset{(ds,ns) \rightsquigarrow (ds,ns)}{\Diamond}} \big(\exists X.\, \mathsf{queue}(a, x, ds, ns) * [\textsc{Deq}(X)]_a * \mathbb{S}_a\big(D_a(X)\big) * \mathsf{PD}(i)\big) \end{array} \right\rangle$

$\quad\quad\quad\quad\quad \left\{ \begin{array}{l} \underline{\textbf{if}}\ \mathtt{b} = 1\ \underline{\textbf{then}} \left(\begin{array}{c} \exists ds, ns, ds', ns', ds'', ns'', v.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) \\ * \mathsf{WD2}(i, \mathtt{first}, v, ds', ns', ds'', ns'') * [\textsc{Value}(\mathtt{first}, v)]_a \end{array} \right) \\ \quad \underline{\textbf{else}}\ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) \end{array} \right\}$

$\quad\quad\quad\quad\quad$ `if (b = 1) {`

$\quad\quad\quad\quad\quad\quad \left\{ \begin{array}{l} \exists ds, ns, ds', ns', ds'', ns'', v.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) \\ * \mathsf{WD2}(i, \mathtt{first}, v, ds', ns', ds'', ns'') * [\textsc{Value}(\mathtt{first}, v)]_a \end{array} \right\}$

$\quad\quad\quad\quad\quad\quad$ `v :=` $[$`first.value`$]$`;` $[$`first.value`$]$ `:= 0;`

$\quad\quad\quad\quad\quad\quad \big\{ \exists ds', ns', ds'', ns''.\, \mathsf{WD2}(i, \mathtt{first}, \mathtt{v}, ds', ns', ds'', ns'') \big\}$

$\quad\quad\quad\quad\quad\quad$ `return v;`

$\quad\quad\quad\quad\quad$ `}`

$\quad\quad\quad\quad\quad \big\{ \exists ds, ns.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * \mathsf{PD}(i) \big\}$

$\quad\quad\quad\quad$ `} } }`

$\quad \left\langle \begin{array}{l} \underline{\textbf{if}}\ vs = []\ \underline{\textbf{then}}\ \mathbf{MSQueue}_a(\mathtt{x}, (ds, ns)) * [\mathrm{G}]_a * \mathsf{ret} = 0 \\ \quad \underline{\textbf{else}} \left(\begin{array}{c} \exists vs', ds', ns', n, v, m.\, \mathbf{MSQueue}_a(\mathtt{x}, (ds', ns')) * [\mathrm{G}]_a * vs' = proj_2(ns') * ns = (n, v, m) : ns' \\ * ds' = ds +\!\!+ [(n, v, m)] \end{array} \right) \end{array} \right\rangle$

$\big\langle \underline{\textbf{if}}\ vs = []\ \underline{\textbf{then}}\ \mathsf{Queue}(s, \mathtt{x}, vs, t) * \mathsf{ret} = 0\ \underline{\textbf{else}}\ \exists vs', t'.\, \mathsf{Queue}(s, \mathtt{x}, vs', t') * vs = \mathsf{ret} : vs' \big\rangle$

Figure 8.11: Proof of correctness of `dequeue` operation.

9 Conclusions

We have introduced a program logic, TaDA, which uses *atomic triples* for specifying abstract atomicity, as well as separation-style Hoare triples for specifying abstract disjointness. With the combination of abstract atomicity and abstract disjointness that TaDA provides, we can specify and verify modules which feature both atomic and non-atomic operations, possibly at different levels of abstraction. Additionally, TaDA allows us to easily extend modules with new operations, as well as build new modules on top of existing ones.

We have shown how TaDA can be extended to prove termination of non-blocking programs. Using our abstract specifications, clients can reason about total correctness without needing to know any of the details of the underlying implementation. Different implementations that satisfy the same specification may have different termination arguments, but these arguments are never exposed to the clients. By using ordinals to bound interference, our specifications can express traditional non-blocking properties. Moreover, they can also capture the notion of *non-impedance*: that one operation does not disrupt the progress of another.

Finally, we have proposed a potential extension to the logic to handle more advanced patterns of concurrency. Using the extension, we were able to give a novel atomic specification to a ticket lock module and to introduce a two-step counter that exhibits speculation, despite its simplicity. Additionally, we applied the technique to a larger example and shown that the approach scales. It remains an open question what the semantic model that justifies such an extension should be.

9.1 Future Work

9.1.1 Tool Support

So far, all of the soundness proofs and proofs shown in this thesis using TaDA or Total-TaDA have been done by hand. One of our goals is to formalise the logic in a proof assistant, such as Coq, and prove its soundness. Moreover, we have implemented Caper [14], a semi-automated verification tool, on a fragment of TaDA. We intend to extend the tool to be able to prove all of the the examples shown.

9.1.2 Helping/Speculation

We have presented a technique that supports reasoning with about abstract atomicity when implementations make use of helping and speculation chapter 8. It seems clear that we require a different semantic model. However, it is not clear what changes must be made in order to support the extension. There are approaches based on contextual refinement that support helping and speculation [59, 39], but are restricted to linearisability and suffer from the problems described in §2.7. Other approaches use higher-order to support helping such as [32, 54, 35], but have no support for speculation.

9.1.3 Higher-order Support

Recently, Iris [35] combined TaDA with iCAP [54] into a new higher order program logic, which encodes TaDA's proof rules in logic. This gives it the expressive power to handle higher-order programs and reentrancy. It would be interesting to explore if we can encode the extensions presented here to prove termination and speculation.

9.1.4 Weak Memory

Burkhardt *et al.* [5] have extended the concept of linearisability to the total store order (TSO) memory model [48]. Additionally, in recent years, there have been a series of program logics that adapt some of the program logics described in the background chapter to several weak memory models [62, 60, 36, 53]. An interesting research direction would be to investigate extensions of TaDA that can specify and verify programs that make use of weak memory models, such as TSO.

9.1.5 Liveness

Blocking

Many concurrent modules make use of *blocking*, for example by using locks or monitors. Properties such as starvation-freedom can be expressed in terms of termination, but require the assumption of a fair scheduler. Some aspects of our approach are likely to be applicable here as well. However, it is also necessary to constrain future behaviours, for instance, to specify that a lock that has been acquired will be released in finite time. Liang and Feng [38] have developed Lili, a program logic based on rely-guarantee and refinement that can verify linearisability and progress properties for concurrent modules under fair scheduling. Unfortunately, their approach suffers from the same problems common to linearisability, such as the lack of ownership transfer and not being able to handle modules with operations such as the `wkIncr`. Understanding how we could extend our termination extension to blocking algorithms and overcome the limitations of linearisability-based approaches would be another interesting research question.

Non-termination

Some programs, such as operating systems, are designed not to terminate. Nonetheless, such programs should still exhibit progress. It would be interesting to extend Total-TaDA to specify and verify progress properties of non-terminating systems. Progress can be seen as localised termination, so the same reasoning techniques should apply. However, it is not clear what kind of specifications will be necessary to express and verify these properties.

9.1.6 Fault Tolerance

An underlying assumption in our logic is that the machine where the program is running does not fail. In practice, machines can fail unpredictably for various reasons, e.g. power loss, corrupting resources. Critical software, e.g. file systems, employ recovery methods to mitigate these effects. In future, we want to combine abstract atomicity from concurrency with host failure atomicity in the style of [46].

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 165–175. IEEE Computer Society, 1988. doi: 10.1109/LICS.1988.5115. URL http://dx.doi.org/10.1109/LICS.1988.5115. (Cited on page 133.)

[2] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270, 2005. doi: 10.1145/1040305.1040327. URL http://doi.acm.org/10.1145/1040305.1040327. (Cited on page 32.)

[3] Pontus Boström and Peter Müller. Modular Verification of Finite Blocking in Non-terminating Programs. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 639–663. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.639. URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.639. (Cited on page 132.)

[4] John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 55–72, 2003. doi: 10.1007/3-540-44898-5_4. URL http://dx.doi.org/10.1007/3-540-44898-5_4. (Cited on pages 32 and 114.)

[5] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2012. doi: 10.1007/978-3-642-28869-2_5. URL http://dx.doi.org/10.1007/978-3-642-28869-2_5. (Cited on page 150.)

[6] Georg Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 49(2):207–246, 1897. ISSN 0025-5831. doi: 10.1007/BF01444205. URL http://dx.doi.org/10.1007/BF01444205. (Cited on page 109.)

[7] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark J. Wheelhouse. A simple abstraction for complex concurrent indexes. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 845–864. ACM, 2011. doi:

10.1145/2048066.2048131. URL http://doi.acm.org/10.1145/2048066.2048131. (Cited on page 43.)

[8] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In Richard Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014. doi: 10.1007/978-3-662-44202-9_9. URL http://dx.doi.org/10.1007/978-3-662-44202-9_9. (Cited on page 21.)

[9] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). *Electr. Notes Theor. Comput. Sci.*, 319:3–18, 2015. doi: 10.1016/j.entcs.2015.12.002. URL http://dx.doi.org/10.1016/j.entcs.2015.12.002. (Cited on page 21.)

[10] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular Termination Verification for Non-blocking Concurrency. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 176–201. Springer, 2016. doi: 10.1007/978-3-662-49498-1_8. URL http://dx.doi.org/10.1007/978-3-662-49498-1_8. (Cited on page 21.)

[11] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE, 2009. doi: 10.1109/ICSE-COMPANION.2009.5071046. URL http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5071046. (Cited on page 38.)

[12] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010. doi: 10.1007/978-3-642-14107-2_24. URL http://dx.doi.org/10.1007/978-3-642-14107-2_24. (Cited on pages 19, 31, 33, 38, 40, 43, 90, 92, and 107.)

[13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 287–300. ACM, 2013. doi: 10.1145/2429069.2429104. URL http://doi.acm.org/10.1145/2429069.2429104. (Cited on pages 47, 62, 65, and 68.)

[14] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic Verification for Fine-grained Concurrency. In *Programming Languages and*

Systems - 26th European Symposium on Programming, ESOP 2017, Lecture Notes in Computer Science. Springer, 2017. (Cited on page 149.)

[15] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-Guarantee Reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 363–377, 2009. doi: 10.1007/978-3-642-00590-9_26. URL http://dx.doi.org/10.1007/978-3-642-00590-9_26. (Cited on pages 31 and 107.)

[16] Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 216–224. ACM, 2004. doi: 10.1145/1007912.1007945. URL http://doi.acm.org/10.1145/1007912.1007945. (Cited on page 102.)

[17] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 143–156. ACM, 2010. doi: 10.1145/1863543.1863566. URL http://doi.acm.org/10.1145/1863543.1863566. (Cited on page 107.)

[18] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 315–327, 2009. doi: 10.1145/1480881.1480922. URL http://doi.acm.org/10.1145/1480881.1480922. (Cited on pages 31 and 38.)

[19] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2009. doi: 10.1007/978-3-642-00590-9_19. URL http://dx.doi.org/10.1007/978-3-642-00590-9_19. (Cited on pages 34 and 107.)

[20] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19:19–31, 1967. (Cited on pages 108 and 109.)

[21] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006. ISBN 978-0-321-34960-6. (Cited on page 141.)

[22] Alexey Gotsman and Hongseok Yang. Linearizability with Ownership Transfer. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2012. doi: 10.1007/

978-3-642-32940-1_19. URL http://dx.doi.org/10.1007/978-3-642-32940-1_19. (Cited on pages 101 and 107.)

[23] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 16–28. ACM, 2009. doi: 10.1145/1480881.1480886. URL http://doi.acm.org/10.1145/1480881.1480886. (Cited on pages 128 and 132.)

[24] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007. doi: 10.1142/S0129626407003125. URL http://dx.doi.org/10.1142/S0129626407003125. (Cited on page 138.)

[25] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi: 10.1145/114005.102808. URL http://doi.acm.org/10.1145/114005.102808. (Cited on page 109.)

[26] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972. (Cited on pages 19, 22, and 33.)

[27] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529. IEEE Computer Society, 2003. doi: 10.1109/ICDCS.2003.1203503. URL http://dx.doi.org/10.1109/ICDCS.2003.1203503. (Cited on page 109.)

[28] Gerhard Hessenberg. *Grundbegriffe der Mengenlehre*. Abhandlungen der Fries'schen Schule / Neue Folge. Vandenhoeck & Ruprecht, 1906. (Cited on page 114.)

[29] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259. (Cited on page 24.)

[30] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative Reasoning for Proving Lock-Freedom. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 124–133. IEEE Computer Society, 2013. doi: 10.1109/LICS.2013.18. URL http://dx.doi.org/10.1109/LICS.2013.18. (Cited on pages 17, 128, and 132.)

[31] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001. (Cited on page 31.)

[32] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 271–282. ACM, 2011. doi: 10.1145/1926385.1926417. URL http://doi.acm.org/10.1145/1926385.1926417. (Cited on pages 107 and 149.)

[33] Xiao Jia, Wei Li, and Viktor Vafeiadis. Proving lock-freedom easily and automatically. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 119–127, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3296-5. doi: 10.1145/2676724.2693179. URL http://doi.acm.org/10.1145/2676724.2693179. (Cited on page 132.)

[34] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983. (Cited on pages 22 and 29.)

[35] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi: 10.1145/2676726.2676980. URL http://doi.acm.org/10.1145/2676726.2676980. (Cited on pages 33, 43, 107, 149, and 150.)

[36] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2015. doi: 10.1007/978-3-662-47666-6_25. URL http://dx.doi.org/10.1007/978-3-662-47666-6_25. (Cited on page 150.)

[37] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 561–574. ACM, 2013. doi: 10.1145/2429069.2429134. URL http://doi.acm.org/10.1145/2429069.2429134. (Cited on pages 33 and 107.)

[38] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 385–399. ACM, 2016. doi: 10.1145/2837614.2837635. URL http://doi.acm.org/10.1145/2837614.2837635. (Cited on page 150.)

[39] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 455–468. ACM, 2012. doi: 10.1145/2103656.2103711. URL http://doi.acm.org/10.1145/2103656.2103711. (Cited on page 149.)

[40] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting*

of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014, pages 65:1–65:10. ACM, 2014. doi: 10.1145/2603088.2603123. URL http://doi.acm.org/10.1145/2603088.2603123. (Cited on page 132.)

[41] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel (abstract). *Operating Systems Review*, 26(2):8, 1992. doi: 10.1145/142111.993246. URL http://doi.acm.org/10.1145/142111.993246. (Cited on page 109.)

[42] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. doi: 10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729. (Cited on page 23.)

[43] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996. doi: 10.1145/248052.248106. URL http://doi.acm.org/10.1145/248052.248106. (Cited on pages 138 and 141.)

[44] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer, 2014. doi: 10.1007/978-3-642-54833-8_16. URL http://dx.doi.org/10.1007/978-3-642-54833-8_16. (Cited on page 33.)

[45] Gian Ntzik. *Reasoning about POSIX File Systems*. PhD thesis, Imperial College London, 2016. (Cited on page 107.)

[46] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-Tolerant Resource Reasoning. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, volume 9458 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2015. doi: 10.1007/978-3-319-26529-2_10. URL http://dx.doi.org/10.1007/978-3-319-26529-2_10. (Cited on page 150.)

[47] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi: 10.1016/j.tcs.2006.12.035. URL http://dx.doi.org/10.1016/j.tcs.2006.12.035. (Cited on pages 22, 31, and 107.)

[48] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27. URL http://dx.doi.org/10.1007/978-3-642-03359-9_27. (Cited on page 150.)

[49] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. doi: 10.1145/360051.360224. URL http://doi.acm.org/10.1145/360051.360224. (Cited on pages 22 and 27.)

[50] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 247–258. ACM, 2005. doi: 10.1145/1040305.1040326. URL http://doi.acm.org/10.1145/1040305.1040326. (Cited on pages 36 and 108.)

[51] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL http://dx.doi.org/10.1109/LICS.2002.1029817. (Cited on page 31.)

[52] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 2015. doi: 10.1007/978-3-662-46669-8_14. URL http://dx.doi.org/10.1007/978-3-662-46669-8_14. (Cited on page 33.)

[53] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 736–761. Springer, 2015. doi: 10.1007/978-3-662-46669-8_30. URL http://dx.doi.org/10.1007/978-3-662-46669-8_30. (Cited on page 150.)

[54] Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2014. doi: 10.1007/978-3-642-54833-8_9. URL http://dx.doi.org/10.1007/978-3-642-54833-8_9. (Cited on pages 33, 38, 65, 92, 107, 149, and 150.)

[55] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. Modular reasoning about separation of concurrent data structures. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2013. doi: 10.1007/978-3-642-37036-6_11. URL http://dx.doi.org/10.1007/978-3-642-37036-6_11. (Cited on pages 33 and 107.)

[56] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955. (Cited on page 126.)

[57] R Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986. (Cited on pages 120 and 121.)

[58] Alan M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949. URL http://www.turingarchive.org/browse.php/B/8. (Cited on page 108.)

[59] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 377–390. ACM, 2013. doi: 10.1145/2500365.2500600. URL http://doi.acm.org/10.1145/2500365.2500600. (Cited on pages 33, 34, 107, and 149.)

[60] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 691–707. ACM, 2014. doi: 10.1145/2660193.2660243. URL http://doi.acm.org/10.1145/2660193.2660243. (Cited on page 150.)

[61] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 343–356. ACM, 2013. doi: 10.1145/2429069.2429111. URL http://doi.acm.org/10.1145/2429069.2429111. (Cited on page 107.)

[62] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884. ACM, 2013. doi: 10.1145/2509136.2509532. URL http://doi.acm.org/10.1145/2509136.2509532. (Cited on page 150.)

[63] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, pages 256–271, 2007. doi: 10.1007/978-3-540-74407-8_18. URL http://dx.doi.org/10.1007/978-3-540-74407-8_18. (Cited on pages 31, 38, and 107.)

[64] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, Lecture Notes in Computer Science. Springer, 2017. (Cited on pages 21 and 90.)