Verifying Concurrent Randomized Algorithms

Joseph Tassarotti

CMU-CS-19-100

January 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Robert Harper (Chair)
Jan Hoffmann
Jeremy Avigad
Derek Dreyer (MPI-SWS)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by Oracle Labs in support of "Formal Verification of Probabilistic Programs"; by a National Defense Science and Engineering Graduate (NDSEG) Fellowship; and, by a fellowship from the ARCS (Achievement Rewards for College Scientists) Foundation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: concurrency, program logics, separation logic, randomized algorithms, verification

For my grandparents.

Abstract

Concurrency and randomization are difficult to use correctly when programming. Because programs that use them no longer behave deterministically, programmers must take into account the set of all possible interactions and random choices that may occur. This dissertation describes a logic for reasoning about programs using both of these effects. The logic extends a recent concurrent separation logic with ideas from denotational semantics for probabilistic and nondeterministic choice, along with principles for probabilistic relational reasoning originally developed for sequential programs. The resulting logic is used to verify probabilistic behaviors of a randomized concurrent counter algorithm and a two-level concurrent skip list. The soundness of the logic, as well as the proofs of these examples, have been mechanized in Coq.

Acknowledgments

I have been very fortunate to have had many excellent mentors and colleagues. When I re-read what is written in this dissertation, I can see how the things I have learned from these people have shaped my work. I hope that if they read this, they too will see these connections.

I start by thanking my advisor, Robert Harper. Early on, Bob told me that his style as an advisor was to suggest interesting ideas to his students, and then let them pursue what they found most compelling. In my own case, I became very interested in two topics which Bob and I often discussed over the years: reasoning about programs with "benign effects" and the cost semantics of parallel languages. Randomization is an effect that can often be used in benign ways, and is used in many efficient implementations of parallel languages. Those implementations are also concurrent, by necessity. Interest in these two topics then led to the work described in this dissertation. But even though I can trace the origins of my interest in this topic to conversations Bob and I had very early in my PhD, the path I took to get there was circuitous. I am extremely grateful that along the way, Bob was always interested in and open to what I wanted to work on at the moment. It is hard not to be inspired by his genuine passion for the field, and I have learned so much from him over the years.

At some point in my first year, I asked Bob what I ought to do over the summer. He said that I should try to do an internship with Derek Dreyer. That turned out to have been an excellent suggestion. Working with Derek, I learned the fundamentals of many of the techniques that are used in this dissertation. He devoted a lot of time to working with me, not just that summer, but also after I went back to Pittsburgh and ever since. I am very grateful for his mentorship.

Jan Hoffmann came to CMU during my third year as a student, and has been an invaluable resource. Listening to talks about his work and reading his papers carefully inspired some of the core ideas for the logic described herein. He has very patiently discussed these ideas with me on many occasions.

As an undergraduate, I was fascinated by the formalization projects that Jeremy Avigad had worked on, and so when I arrived at CMU I asked if he would meet with me to talk about some of that work. He agreed, and he turned out to be as kind and approachable as I had heard. I am very pleased that he has agreed to be on my thesis committee.

I have learned a lot from the rest of the CMU faculty. Karl Crary gave inspirational lectures about parametricity in a course I took my first year, and introduced me to a useful perspective on programming languages for concurrency. Frank Pfenning's talks and lecture notes are models of pedagogy and present a beautiful perspective on logic. Stephen Brookes helped me understand some finer points about concurrent separation logic and was very encouraging when I presented my early work.

Viktor Vafeiadis, who co-advised me during my internship at MPI-SWS, taught me a great deal about concurrent separation logic and the many subtleties involved in reasoning about concurrent systems. His paper on operational soundness proofs for concurrent separation logic made many things clear to me for the first time.

Greg Morrisett got me started on research when I was an undergraduate, and I learned a lot from him in lectures, seminars, and informal conversations. It is hard to imagine a better undergraduate advisor. I thank him and Gang Tan for letting me work with them on the RockSalt project. During my time as an undergraduate, I also learned an immense amount in courses taught by Margo Seltzer and Stephen Chong.

Jean-Baptiste Tristan has taught me so much over the seven years I have known him. A recurring theme is that Jean-Baptiste will tell me about some topic he's started to learn about, and before I know it I am deeply fascinated by the subject and reading papers about it. This has happened so often that he promised not to talk to me about computer science for the last few months I was writing this dissertation so that I would not get distracted from finishing it. I thank him for teaching me about approximate counting algorithms, and suggesting them as an example to consider in this dissertation. He has been an excellent collaborator, and has looked out for my career and given me advice at many crucial times.

Before starting graduate school, I did a very pleasant internship at Oracle Labs in Burlington. I thank Guy Steele and Victor Luchangco, along with the rest of the lab, for many conversations that guided my thinking about parallelism and concurrency. I was lucky to be able to do my internship at the same time as Daniel Huang. Michael Wick arrived at Oracle after I left but subsequently became a valued collaborator.

My dissertation depends enormously on the Iris project, and I thank all of the people who have contributed to that project. Ralf Jung and Robbert Krebbers deserve particular thanks for many conversations about Iris and related ideas. I am also grateful to Robbert for suggesting to me the possibility of modifying the Iris adequacy proof along the lines worked out in Chapter 4 of this dissertation.

Lars Birkedal's research has had a large influence on much of my work. I am appreciative of the interest he has shown in my ideas over the years, which was very encouraging to me at important times.

I thank Justin Hsu for explaining to me the work that he and his collaborators have done on probabilistic program logics. As will be clear, my dissertation owes much to the ideas they have developed. Justin also gave me helpful feedback on drafts of papers about the work that became this dissertation.

I am very grateful to the administrative support staff in the Computer Science Department at CMU for their help over the years. In particular, it is hard to imagine how the graduate program could run without Deb Cavlovich, whose prompt answers to my many questions and gentle reminders about various deadlines were invaluable. Ben Cook similarly helped me navigate and deal with many university policies and procedures.

I've learned a lot from students and post-docs at CMU. Among them, Carlo Angiuli, Jon Sterling, Daniel Gratzer, Adrien Guatto, Favonia, Ed Morehouse, and Michael Sullivan deserve special thanks for many conversations about both the technical and philosophical aspects of our field.

My family has always supported me, and I thank them for their love. It goes without saying that I would not be where I am today without them. My parents did so much for my education and have encouraged me to pursue my interests.

Finally, I thank Mari Tanaka for always being there, even as she pursued her own incredibly challenging education and career. Her hard work has been inspirational to me over the years. I am so lucky to have her love, friendship, and support.

Note

This document contains text and figures from [114]. I have used LaTeX macros from [116] to typeset proof rules and examples.

Contents

List of Figures

Chapter 1

Introduction

Mechanized program verification has advanced considerably in recent decades. For experienced users of interactive theorem provers, verifying the correctness of purely functional programs is often not much harder than doing a thorough pencil-and-paper proof. In part, this is because the semantics of purely functional languages are so well-behaved that there is little or no gap between programs written in them and the idealized pseudo-code that a textbook or paper might use to describe an algorithm.

However, when programs use effects, the situation changes. In some sense, this is to be expected: effects can make it more difficult to understand and informally reason about programs, so it is not surprising that they also make formal verification more challenging. But in addition, the gap between common informal reasoning principles and the formal semantics of these languages grows in the presence of effects. For instance, for pointer manipulating programs, one wants to employ *local* reasoning when manipulating distinct parts of a data structure: making a change to one piece of a structure does not affect other parts. However, working "directly" from the operational semantics in a theorem prover, one would need to constantly re-iterate that certain pointers are non-aliasing to make this reasoning precise. Similarly, in the concurrent setting, one often thinks of a lock as "protecting" a certain part of the heap, so that when the lock is held, another thread cannot be manipulating the protected region. But in many programming languages, there is no actual concrete association between a lock and parts of the heap. The lock is merely a bit, and it is a convention upheld by the programmer that a given part of the heap will not be accessed without first acquiring the lock.

To address this gap, a long standing research tradition has focused on developing logics and methodologies to make reasoning about effectful programs easier. In the best of cases, these logics codify informal reasoning principles into formal rules. Sometimes, these logics also provide support for *relational* or *refinement* reasoning: to understand a complex program, we first prove a correspondence between its behavior and that of some simpler program. Then, we can analyze or verify properties of this simpler program and draw conclusions about the original program.

One of the most well known examples of a logic developed for reasoning about effectful programs is *separation logic* [103], developed by Reynolds, O'Hearn, Ishtiaq, Yang, and Pym. The idea of separation logic was to introduce a new connective $P * Q$, called separating conjunction, which expressed that the program state could be divided up into two separate pieces,

one satisfying the assertion P and the other satisfying Q. This logic thus formalized the principle mentioned above about reasoning locally about pieces of code that operate on disjoint parts of program state. Yang [129] subsequently extended separation logic to support relational reasoning, making it possible to re-use the idea of separation while establishing a relation between two programs. O'Hearn [94] and Brookes [26] later showed that separation logic could be naturally used to reason about concurrent programs, by formalizing principles like the notion of a lock protecting or "owning" a region of the heap.

Subsequent work has extended these logics in various ways, providing new support for reasoning about concurrent algorithms. However, a common criticism [96] of much of the research in this area is that to reason about some clever use of concurrency, one often seems to need to extend the logic with a new feature or rule. In response to this criticism, recent research has tried to unify many previous concurrency logics. Jung et al. [64] argue convincingly that many features developed for reasoning about concurrency can be encoded in a logic built on simpler foundations.

But of course, there are effects other than state and concurrency. One of the most important is *randomization* (probabilistic choice). In a separate line of work, various extensions to Hoare logic and Dijkstra's weakest precondition calculus have been developed for randomized imperative programs [10, 13, 71, 86].

What do we do if we want to reason about programs that use both concurrency *and* randomization? In this document I describe a program logic called Polaris that I have developed which has support for reasoning about programs using both of these effects. The design of this logic, and the scientific conclusion of this dissertation, are summarized in the following thesis:

> Separation logic, extended with support for probabilistic relational reasoning, provides a foundation for the verification of concurrent randomized programs.

By "verification", I mean proving both functional correctness and complexity bounds. In the rest of this introduction, I provide the background needed to understand this claim. First, I will describe several examples of concurrent algorithms that use randomization. Next, I will survey related work on program logics, explaining in more detail the ideas behind the logics alluded to above. Having done so, I will motivate the design of the logic described herein.

1.1 Concurrent Randomized Algorithms

In this section, I discuss three concurrent algorithms and data structures which either use randomization directly, or whose analysis in the "average case" involves the consideration of randomness. The purpose is not to give the full details of these algorithms or their analyses, but simply to suggest the kinds of issues that come up when concurrency is combined with randomness.

In the examples that follow, concurrency and randomization *interact* directly: either the concurrent interactions between threads depend on earlier random choices they make, or the effects of their random choices are perturbed by concurrent interaction. This distinguishes them from a simpler way in which concurrency and randomized algorithms can be combined: In a certain sense, if we take any randomized sequential algorithm, and use it in a setting where

there are multiple interacting threads, we suddenly have to reason about both concurrency and randomness. For example, we can modify imperative randomized Quicksort so that it forks a new thread after the partitioning step to help sort one of the two sublists.

However, in this case, the threads do not really interact, because they operate on disjoint sublists, so their random choices do not affect one another[1]. Thus, analyzing the total number of comparisons performed by all threads is not considerably more complicated than the usual sequential analysis. That is not to say that formally proving this is simple, but in the examples that follow, there is a more fundamental interaction between concurrency and randomness.

1.1.1 Binary Search Trees

Binary search trees are a very old and well-studied data structure in computer science. The *height* of a tree, which is the number of edges in the longest path from the root to a leaf, is related to the worst case time to find an element in the tree. If n items are successively inserted into an empty tree using the traditional algorithm, then it is possible for the resulting tree to have height $n - 1$. In this case, the tree is *unbalanced*, and searching in such a tree is no better than linearly searching through a list. For that reason, a variety of algorithms for *self-balancing* trees have been developed that try to maintain a height of $O(\log n)$ by doing extra work to re-balance the tree when items are inserted.

However, even with the classical binary search tree, most insertion orders do not lead to this worst case height of $n - 1$. If we insert a set X of n elements, where the insertion order is given by a random permutation on X, each equally likely, then in expectation the height of the tree is $O(\log n)$. In spite of this, self-balancing binary tree algorithms are often still preferred in non-concurrent applications because they are guaranteed to avoid the worst case behavior[2].

However, in the concurrent setting the trade-offs are not so clear. Self-balancing algorithms generally need to acquire a lock while re-balancing the tree, which can prevent other threads from searching. Ellen et al. [38] proposed a non-blocking concurrent binary tree algorithm that used atomic compare-and-swap (CAS) instructions instead of locks, but did not perform rebalancing. Since then, a number of other non-balancing concurrent binary trees have been proposed [5, 91]. Depending on the number of threads and the work-load, non-balancing trees can perform better than balancing ones [5].

This raises new interest in properties of non-balancing binary search trees. However, as Aspnes and Ruppert [6] point out, the prior analysis of random binary search trees in the sequential setting does not necessarily carry over to the concurrent setting. Imagine a simplified scenario in which c threads are concurrently trying to insert items from some queue into a tree which is protected by a global lock. To do an insertion, a thread first acquires this lock, performs the usual insertion algorithm, and releases the lock. After completing an insertion, a thread gets the next item from the queue and tries to insert it. The threads stop once all the items from the queue have been inserted.

[1] In a sense, this is because the underlying algorithm is really data parallel, but when expressed in many languages the fact that there is no interaction is something that must be proven, rather than an immediate consequence of the semantics of the language.

[2] Another issue is that the above results about tree height do not hold under repeated deletions and insertions of additional elements using standard algorithms [62, 69].

The problem is that the order in which items are actually inserted into the tree is not necessarily the same as the order they appear in the queue. In particular, the order of insertions will depend on the order that the threads actually acquire the lock, which is subject to various effects that are difficult to model.

Aspnes and Ruppert [6] therefore propose an *adversarial* model: imagine there is a scheduler which can compare the nodes each thread is trying to insert, and then gets to choose which thread goes next, with the goal of maximizing the average depth of nodes in the tree[3]. They show that the expected average depth is $O(c + \log n)$. Of course in reality, the scheduler is not actually trying to maximize the average depth, but the point is to do the analysis under very conservative assumptions.

In their analysis, Aspnes and Ruppert [6] do not consider the actual code or algorithms for concurrent binary trees, but rather phrase the problem as a kind of game involving numbered cards, where the number of threads c corresponds to the number of cards in the hand of the player. This abstraction lets them focus on the relevant probabilistic aspects of the problem without considering the concrete details of these algorithms. As we will see in the rest of this section, the process of abstracting away from the concurrent code to a more mathematical model is very common in the analysis of concurrent randomized algorithms.

1.1.2 Approximate Counters

Approximate counters are another algorithm with renewed relevance in large scale concurrent systems. They were originally proposed by Morris [87] as a way to count a large number of events in a memory constrained setting. Usually, to count up to n with a standard counter, one needs $\log_2 n$ bits. Morris's idea was that rather than storing the current count k, one could store $\lfloor \log_2 k \rfloor$. Then, one can count up to n using only $\log_2 \log_2 n$ bits, at the cost of some inaccuracy due to round-off.

The difficulty is that because one is only storing a rounded-off approximation of the current count, when we perform an increment it is not clear what the new value of the counter should be. Morris proposed a randomized strategy for doing increments. The code for this increment routine is shown in Figure 1.1a, written in an ML-like pseudo code. The command flip(p) returns True with probability p and False otherwise. If the current value stored is x, then with probability $\frac{1}{2^x}$ an increment updates the value to $x + 1$ (in effect, doubling our estimate of the count) and with probability $1 - \frac{1}{2^x}$ leaves the value at x. If the counter is initialized with a value of 0, and C_n is the random variable giving the value stored in the counter after n calls of the increment function, then $\mathbb{E}[2^{C_n}] = n + 1$. Hence we can estimate the actual number of increments from the approximate value stored in the counter. Morris proposed a generalization with a parameter that could be tuned to adjust the variance of 2^{C_n} at the cost of being able to store a smaller maximum count. Flajolet [41] gave a very detailed analysis of the distribution of C_n, in which he first observed that the value stored in the counter can be described as a very simple Markov chain, which he then proceeded to analyze using techniques from analytic combinatorics [42].

Morris's counters may seem relatively unimportant today when even cell phones commonly

[3]The depth of a node is the number of edges from the root to the node.

```
incr l ≜
  let k = !l in
  let b = flip(1/2^k) in
  if b then l := k + 1
  else ()

read l ≜ let k = !l in 2^k − 1
```

(a) Sequential approximate counter.

```
incr l ≜
  let b = randbits(64) in
  incr_aux l b

incr_aux l b ≜
  let k = !l in
  if lsbZero(b, k) then
    if CAS(l, k, k + 1) then ()
    else incr_aux l b
  else ()
```

(b) Dice et al.'s concurrent counter (simplified).

```
incr l ≜
  let k = min(!l, MAX) in
  let b = flip(1/(k + 1)) in
  if b then (FAA(l, k + 1); ())
  else ()

read l ≜ !l
```

(c) An unbiased concurrent counter.

Figure 1.1: Approximate counting algorithms.

have gigabytes of memory and a 64-bit integer can store numbers larger than 10^{19}. However, in the concurrent setting, multiple threads may be trying to increment some shared counter to keep track of the number of times an event has happened across the system. In order to do so correctly, they need to use expensive atomic instructions like fetch-and-increment or compare-and-swap (CAS) which have synchronization overheads. Dice et al. [32] realized that if one instead uses a concurrent form of the approximate counter, then as the number stored in the counter grows larger, the probability that the value needs to be modified gets smaller and smaller. Thus, the number of actual times a thread needs to perform a concurrent update operation like CAS goes down. In this setting, the probabilistic counter is useful not because it reduces memory use, but because it decreases contention for a shared resource.

Dice et al. [32] propose a number of variants and optimizations for a concurrent approximate counter. For instance, they suggest that one can use an adaptive algorithm that keeps track of the exact count until reaching a certain count, and then switches to the approximate algorithm. This way, for small counts the values are exact, and if the counts are still small, there must not be that much contention yet, so there is no need to be using the approximation scheme.

A simplified version of the non-adaptive increment function for one of their proposals is shown in Figure 1.1b. The code ignores overflow checking for simplicity. The function starts by generating a 64 bit vector uniformly at random, which is then bound to a variable b. It then enters a loop in which it reads the current value of the counter. If the current value is k, it checks whether the first k bits of b are 0, which occurs with probability $\frac{1}{2^k}$. If so, it attempts to increment the counter by atomically updating it to `k+1` using a CAS, and otherwise it returns. If the CAS returns true, this means the CAS has succeeded and no other thread has done an increment in between, so again the code returns. If the CAS returns false, it repeats.

The code does not generate a new random number if the CAS fails. Although Dice et al. [32] do not address this in their work, this raises the possibility for an adversarial scheduler to affect the expected value of the counter, much as the scheduler can affect tree depth in the analysis of concurrent trees by Aspnes and Ruppert [6]. Imagine c threads are attempting to concurrently perform an increment, and the scheduler lets them each generate their random value of randbits and then pauses them. Suppose the current value in the counter is k. Some of the threads may have drawn values for randbits that would cause them to not do an increment, because there is a 1 within the first k bits of their number. Others may have drawn a number where far more than the first k bits will be 0: these threads would have performed an increment even if the value in the counter were larger than k. The scheduler can exploit this fact to maximize the value of the counter by running each thread one after the other in order of how many 0 bits they have at the beginning of their number.

Figure 1.1c presents an unbiased concurrent approximate counter. Unlike Morris's algorithm, it tries to store an estimate of the actual count, not the logarithm of the count, so it uses the standard $\log_2(n)$ bits. But it still has the property that as the count grows larger, the probability of an increment decreases – thus, it would have similar scalability as the biased variant of Dice et al. [32]. The increment routine first reads the current value in the counter. It then takes the minimum of this value and MAX, which is some parameter to the algorithm, and binds the minimum to k. Then, with probability $\frac{1}{k+1}$, it adds $k+1$ to the value in the counter using an atomic fetch and add instruction; otherwise, it leaves the count unchanged.

How can we show that this counter is unbiased? Because addition is commutative and

associative, it does not matter that in between the moments in which a thread reads the value, makes its random choice, and then finally does an increment, another thread may also modify the counter. Therefore, we want to use a similar abstraction as in the analysis of binary search trees by Aspnes and Ruppert: we think of the effects of concurrency as an adversary that merely gets to affect the value of k that is used in each call to increment. And, because no matter what value k is used in a given call to the increment routine, the expected value it will add to the count is 1, because

$$\frac{1}{k+1} \cdot (k+1) + \frac{k}{k+1} \cdot 0 = 1$$

Thus, assuming this adversarial abstraction is correct, we would like to argue that by linearity of expectation, the expected value after performing n increments will be n.

1.1.3 Skiplists

Pugh [100] developed *skip lists*, a data structure that can be used to implement a dynamic set interface for ordered data. A skip list consists of several sorted linked lists, where the nodes in each list contain a key. We visualize each list as running horizontally from left to right, with the different lists stacked vertically above one another (see Figure 1.2). For simplicity we will consider a version only having two lists, and only containing integer keys. The set of keys contained in the top list is a subset of the keys contained in the bottom list, and the node containing a key k in the top list includes a pointer to the corresponding node for k in the list below it. At the beginning and ends of each list, there are sentinel nodes containing the minimum and maximum representable integers (which are written as $-\infty$ and $+\infty$ in Figure 1.2).

We first consider how operations on this data structure are implemented in the sequential case. To check whether a key k is contained in the set, we begin by searching for the key in the top list starting at the left sentinel. If we find a node containing it, we return true. If not, we stop at the largest key $j < k$ in the list, and then follow the pointer in j's node to the copy of j in the bottom list. We then resume searching for k starting at node j in the bottom list. If k is found in the bottom list we return true, otherwise the key is not in the set so we return false. To insert k, we first find the nodes N_t and N_b with the largest keys less than $\leq k$ in the top and bottom list, respectively. If we find that k is already in either list, we stop and return. Otherwise we execute $\mathsf{flip}(p)$, where p is some fixed parameter of the data structure that we can select. If it returns true, we insert new nodes for key k into both the top and bottom lists, after N_t and N_b. Otherwise, if it returns false, we only insert a node in the bottom list after N_b. We call N_t and N_b the "predecessor nodes", because they become the predecessors of k if it is inserted into each list.

If n distinct keys are inserted into the set, then in expectation n/p of them will appear in the top list. Then when searching for a key, we will be able to more quickly descend down the top list, and either find they key there, or if not, only have to examine a few additional nodes in the bottom list. Of course, it is possible (though unlikely, depending on p) that none or all of the nodes are inserted into the top list, in which case we are effectively searching in a regular sorted linked list. More precisely, the number of nodes we will examine in the top list is binomially distributed and the number in the bottom list is geometrically distributed, so from standard properties of these distributions we can derive the expected number of nodes searched.

If there are more than two lists, instead of generating a random bit, we sample from a geometric distribution to obtain a "height" h for the node, and then only insert the node in the bottom h levels. The analysis is more involved, but one approach, found in Pugh's early analysis of the structure, is to show that the number of comparisons is stochastically bounded by the sum of several independent random variables drawn from "standard" well-known distributions (negative-binomial and binomial). Other techniques involve deriving recurrence relations for the expected number of comparisons and then analyzing the asymptotic behavior of these recurrences [95] or relating the skip list to a probabilistic branching process, and then applying results from the theory of such processes [31].

There are several ways to add support for concurrent operations to a skip list. We will consider a simplified implementation[4] inspired by that of Herlihy et al. [52]. We start by adding a lock to each node in the lists. Checking for whether a key is in the set is the same as in the non-concurrent case, and no locks need to be acquired.

To insert a key k, we again search for the predecessor nodes N_t and N_b. When we identify one of these nodes, we acquire its lock and then check that the node after it has not changed in the time between when we examined its successor and when the lock was acquired. If it has, that means another thread may have inserted a new node with key k' such that $N_t < k' \leq k$ or $N_b < k' \leq k$. In that case N_t or N_b is not the appropriate predecessor, so we release the locks and search for the new predecessor (or, in fact, find that k has already been inserted). Otherwise, so long as we hold the locks, we are guaranteed that N_t and N_b will remain the proper predecessors for key k, and that no other thread can insert k. Having acquired both locks, we proceed as in the sequential case by generating a random bit, and on the basis of that bit we insert new nodes for k into either both lists or just the bottom list. We then release the locks and return.

What effect does concurrency have on the number of nodes that must be examined to find a key? The answer is none, so long as there are no concurrent insertions happening while searching. The reason is that in the implementation we have just described, the random choice is made *after* acquiring the locks for insertion. Thus, at the point the random choice is made, the ordering of operations by threads cannot affect where the node will be inserted.

However, an eager programmer might consider the following "optimization" of the concurrent insertion routine: If we generate the random bit *before* acquiring locks for the predecessors, and the resulting bit says we will only insert the node in the bottom list, then we only need to acquire the lock for the bottom predecessor. More generally, if there were more than two levels in the list, we could just acquire the locks up to the height we are going to insert the node into, which might be even more beneficial.

How does this optimization affect the probabilistic analysis? Now the distribution is no longer equivalent to the classical sequential version. To see why, imagine two threads are trying to concurrently insert key k into the list. Suppose that the outcome of the first thread's random bit generation indicates that it will insert the node only into the bottom list, but the second thread will try to insert into both lists. Then the scheduler can influence the distribution by pausing the second thread and letting the first thread finish; when the second thread is eventually allowed to run, it will find that k is already in the list and so it will return without

[4]Some additional subtleties arise if one wants to support deletion from the list.

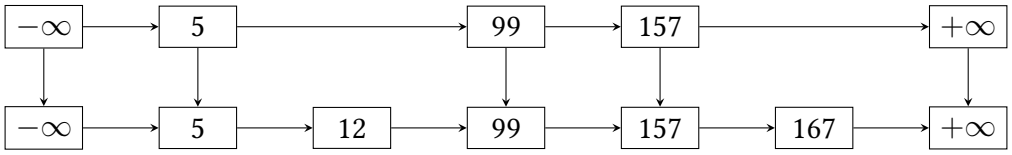

Figure 1.2: Diagram for 2-level concurrent skip list.

doing anything.

Although I have once more used adversarial language to describe the scheduler, in this case it is clearer how behavior like this could arise without having to imagine any malice. Because the first thread only has to acquire a single lock, it really is plausible that it might tend to finish before the second thread.

In this section, we have examined several algorithms that use both concurrency and probabilistic choice, and briefly covered how their probabilistic behaviors are analyzed, or at least how sequential analogues are analyzed. What conclusions can we draw from these examples? A common pattern is that when analyzing the probabilistic behavior, we want to abstract away from implementation details. For example, it is more helpful to model the behavior of the scheduler in terms of adversarial choices over, say, the order nodes are inserted into a search structure, rather than over the set of possible interleavings of the discrete steps involved in *performing* these insertions. Going further, we can sometimes argue that concurrency does not affect the probabilistic quantity we are interested in (*e.g.,* for the expected value of the unbiased counter, or the skip list without the "optimization"). Finally, when deriving quantitative results, one often shows that the probabilistic quantity of interest obeys some recurrence relation, or that it can be related to "standard" probability distributions. Once such relations are established, analysis can proceed without reference to the original algorithm at all. As will become clear later, these observations have guided the design of the logic presented in this dissertation.

1.2 Related Work on Program Logics

A recurring theme in the previous section is that the analysis of concurrent randomized algorithms usually involves a process of abstraction: the code is modeled by some simpler stochastic process and then various properties of this model are analyzed.

But how do we *prove* that these more abstract models faithfully describe the behavior of the code? More generally, how do we prove even more basic properties of such programs, like showing that they do not dereference null pointers or trigger other kinds of faults?

To carry out such proofs, researchers have developed various program logics. In this part I describe prior work on these logics for reasoning about concurrent programs and randomized algorithms. The focus here is on what I consider to be the core insight or idea underlying each logic.

1.2.1 Hoare Logic

Most program logics for imperative programs are extensions to or otherwise based on Hoare logic [55]. Recall that in traditional Hoare logic, one establishes judgments of the form:

$$\{P\}\, e\, \{Q\}$$

where e is a program and P and Q are predicates on program states, which we call assertions. This judgment, called a "triple", says that if e is executed in a state satisfying P, then execution of e does not trigger faults, and after e terminates, Q holds. In addition to rules for all of the basic commands of the language, Hoare logic has two important structural rules:

$$\textsc{Ht-csq}\quad \frac{P \Rightarrow P' \qquad \{P'\}\, e\, \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\, e\, \{Q\}} \qquad\qquad \textsc{Ht-seq}\quad \frac{\{P\}\, e_1\, \{Q\} \qquad \{Q\}\, e_2\, \{R\}}{\{P\}\, e_1; e_2\, \{R\}}$$

The first, called the rule of consequence, is a kind of weakening rule: from a derivation of $\{P'\}\, e\, \{Q'\}$ we can weaken the postcondition Q' to Q and (contravariantly) strengthen the precondition to P to conclude $\{P\}\, e\, \{Q\}$. The second rule, called the sequencing rule, lets us reason about the program $e_1; e_2$, which first executes e_1 and then e_2, by proving triples about each expression separately. The postcondition we prove for e_1 must match the precondition needed by e_2.

Subsequently, Benton [21] observed that Hoare logic could be extended to do relational reasoning about two programs. Instead of the triples from Hoare logic, Benton's logic featured a "quadruple" judgment about pairs of programs:

$$\{P\}\, e_1 \sim e_2\, \{Q\}$$

where now the assertions P and Q are *relations* on pairs of program states, and the judgment means that if e_1 and e_2 execute starting from states related by P, afterward their states will be related by Q. Benton showed that this logic was useful by using it to verify a number of compiler transformations.

An alternative formulation of Hoare's logic, advocated by Dijkstra, is the "weakest precondition calculus" [33]. Instead of Hoare's triples, there is a predicate $\mathsf{wp}\ e\ \{Q\}$, which holds for a given program state S if when e is executed from S, it will not fault, and if it terminates, the resulting state will satisfy Q. Then the triple $\{P\}\, e\, \{Q\}$ can be encoded as $P \Rightarrow \mathsf{wp}\ e\ \{Q\}$. Benton's quadruples can be similarly presented in this style. In the rest of this introduction, I will generally present things using Hoare-style judgments, but some of the logics I mention are actually based on the weakest precondition calculus, and the Hoare triple is defined using an encoding like the above. Later on, we will see why working directly with weakest preconditions can be preferable.

One final variation I will use throughout is that when reasoning about languages where programs are expressions that evaluate to values (as in ML-like languages), it is more natural to consider triples of the form:

$$\{P\}\, e\, \{x.\, Q\}$$

where x is a variable that may appear in Q, and the judgment now means that if the precondition holds and e terminates with some value v, then $[v/x]Q$ holds. When x does not appear in Q, we can omit the binder so that it resembles the traditional Hoare triple. The sequencing rule then becomes a "let" rule:

$$\textsc{Ht-let}\quad \frac{\{P\}\ e_1\ \{x.\, Q\} \qquad \forall v.\, \{[v/x]Q\}\ [v/x]e_2\ \{y.\, R\}}{\{P\}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \{y.\, R\}}$$

At first glance, Hoare logic may seem to be restricted to establishing *functional* correctness properties, *i.e.,* proving properties about the return value or final state of a program after execution. However, it can also be used to reason about the complexity of a program. This can be done in several ways. For example, one can "instrument" a program with "ghost code" that counts the number of operations performed, and then one can bound the value computed by this ghost code. Alternatively, the operational semantics of the language can be modified to track the total number of steps in a designated part of state. Similar encodings can be used with the other logics that will be discussed throughout this dissertation.

1.2.2 Separation Logic

The language considered by Hoare in his original work was rather limited. It did not have references or pointers between memory cells, which are needed for representing various mutable data structures such as linked lists and trees.

Unfortunately, adding pointers poses several challenges. To see why, imagine we extend the logic with a primitive assertion $p \mapsto v$, which says that p is a pointer to a memory cell containing the value v. Natural rules for writing and reading from pointers would then be:

$$\{p \mapsto v\}\ p := v'\ \{x.\, x = () \wedge p \mapsto v'\} \qquad\qquad \{p \mapsto v\}\ !p\ \{x.\, x = v \wedge p \mapsto v\}$$

We can now specify various data structures by linking together these pointer assertions. For example, we can define a predicate $\mathsf{list}(p, l)$ which says that p points to a linked list of nodes containing the values in the abstract list l. If we have a function $\mathsf{append}(p_1, p_2)$, which takes two pointers p_1 and p_2 to linked lists, and appends the second list to the end of the first. We might hope to prove that:

$$\{\mathsf{list}(p_1, l_1) \wedge \mathsf{list}(p_2, l_2)\}\ \mathsf{append}(p_1, p_2)\ \{\mathsf{list}(p_1, l_1 +\!\!+\, l_2)\}$$

where $+\!\!+$ is the append operation for abstract lists. Unfortunately, this is very likely to not be true! The problem is that the precondition does not rule out the possibility that p_1 and p_2 are equal, and if they are, $\mathsf{append}(p_1, p_2)$ might produce a cycle in the list p_1, so that the tail points back to the head.

One solution is to define a predicate $\mathsf{disjoint}(p_1, p_2)$ which states that two linked lists are disjoint, and then modify the precondition to add this as an additional assumption:

$$\{\mathsf{list}(p_1, l_1) \wedge \mathsf{list}(p_2, l_2) \wedge \mathsf{disjoint}(p_1, p_2)\}\ \mathsf{append}(p_1, p_2)\ \{\mathsf{list}(p_1, l_1 +\!\!+\, l_2)\} \tag{1.1}$$

Although this works, it soon leads to proofs that are cluttered with all of these disjointness assumptions. However, an even more fundamental problem is that having proved the above triple, we soon find it is not very re-usable when verifying programs that use the append function. For example, suppose we wanted to prove:

$$\{\mathsf{list}(p_{11}, l_{11}) \wedge \mathsf{list}(p_{12}, l_{12}) \wedge \mathsf{list}(p_{21}, l_{21}) \wedge \mathsf{list}(p_{22}, l_{22}) \wedge (\text{disjointness assumptions})\}$$
$$\quad \mathsf{append}(p_{11}, p_{12}); \mathsf{append}(p_{21}, p_{22})$$
$$\{\mathsf{list}(p_{11}, l_{11} +\!\!+ l_{12}) \wedge \mathsf{list}(p_{21}, l_{21} +\!\!+ l_{22})\}$$

To carry out this proof, we would like to use the sequencing rule and then apply our earlier proof about append twice. The problem is that now our precondition mentions the other linked lists, and so does not match the precondition of the triple in 1.1. We might try to use the rule of consequence to fix this, because the precondition here certainly implies the precondition of 1.1. The problem is that in so doing we would "forget" about the other pair of linked lists, p_{21} and p_{22}, and so the postcondition we would derive for $\mathsf{append}(p_{11}, p_{12})$ would not imply the precondition needed for $\mathsf{append}(p_{21}, p_{22})$.

What is needed is something like the following additional structural rule:

$$\frac{\{P\}\, e\, \{Q\}}{\{P \wedge R\}\, e\, \{Q \wedge R\}}$$

Taking P to be the assumptions about the lists p_{11} and p_{12}, and letting R be the assumptions about the other lists, this would let us temporarily "forget" about the second pair of lists while we derive a triple about $\mathsf{append}(p_{11}, p_{12})$. Then, in the postcondition we again recover R, the facts about the second list.

Unfortunately, this rule is not sound. The problem is that in general R might mention state that is subsequently modified by e in a way that makes R no longer hold. For example, we would be able to derive:

$$\frac{\{p \mapsto v\}\, p := v'\, \{p \mapsto v'\}}{\{p \mapsto v \wedge p \mapsto v\}\, p := v'\, \{p \mapsto v' \wedge p \mapsto v\}}$$

where now p points to two possibly different values.

The most complete and satisfying resolution to this problem was developed by Reynolds, O'Hearn, Ishtiaq, and Yang in an extension to Hoare logic called *separation logic* [103], in which the assertions are a form of the substructural logic of Bunched Implications developed by O'Hearn and Pym [93]. Separation logic introduces a new connective $P * Q$, called separating conjunction. This assertion holds if the program state (the "heap") can be split into two disjoint pieces, which satisfy P and Q respectively. To make this notion of "splitting" the heap precise, one observes that the heap can be thought of as a finite partial function from locations to values. The set of heaps then forms a *partial commutative monoid* (PCM) where the monoid operation is disjoint union of the sets representing these heap functions, and the identity is the empty heap. Then heap h satisfies $P * Q$ if there exists heaps h_1 and h_2 such that $h = h_1 \cdot h_2$, h_1 satisfies P, and h_2 satisfies Q.

This removes the need for the $\mathsf{disjoint}(p_1, p_2)$ assertions, because from $\mathsf{list}(p_1, l_1) * \mathsf{list}(p_2, l_2)$ we can conclude the two lists are disjoint as they must reside in separate pieces of the heap. In addition, this form of conjunction validates the structural rule we wanted before:

$$\textsc{Ht-frame}\ \frac{\{P\}\ e\ \{Q\}}{\{P * R\}\ e\ \{Q * R\}}$$

Intuitively, the meaning and soundness of this rule is clear, since if e is only operating on the part of the heap described by P, it will not modify the part described by R, so that part will continue to satisfy R after e executes.

Unlike normal conjunction, $P \not\vdash P * P$, because a heap satisfying P may not be decomposable into two heaps each satisfying P. As a result, it is best to interpret propositions not as facts, but as descriptions of "resources" (heap pieces) which can be used and transformed, but not duplicated.

1.2.3 Concurrency Logics

O'Hearn [94] realized that separation logic's interpretation of heaps as resources was also useful for verification of concurrent programs. He observed that these programs often essentially divide the heap into pieces which they operate on separately, relying on synchronization primitives like locks to transfer "ownership" between themselves. Writing $e_1 \mid\mid e_2$ for the concurrent composition of e_1 and e_2, he proposed the following rule:

$$\frac{\{P_1\}\ e_1\ \{Q_1\} \qquad \{P_2\}\ e_2\ \{Q_2\}}{\{P_1 * P_2\}\ e_1 \mid\mid e_2\ \{Q_1 * Q_2\}}$$

Intuitively, the soundness of this rule is justified by the fact that the premise about e_1 suggests it only uses the part of the heap satisfied by P_1, and analogously for e_2 and P_2, so when we run them concurrently they will not interfere.

Of course, if we could only compose threads that never interacted at all, this would exclude many important uses of concurrency. O'Hearn's logic was for a programming language equipped with a monitor-like synchronization primitive (a scoped lock with a condition variable), which threads could use to control access to shared pieces of state. Therefore, O'Hearn also proposed a way to associate assertions called *invariants* with these synchronization primitives. Then, the logic had a rule that allowed threads to access the resources described by those assertions upon entering a critical section guarded by the synchronization primitive, so long as they re-established the invariant and relinquished the resources upon exiting the critical section.

Brookes [26] proved the soundness of O'Hearn's logic by giving a semantics of the concurrent language which formalizes this ownership transfer between threads via synchronization primitives.

But what if we want to verify the correctness of the synchronization primitives themselves? Brookes and O'Hearn took as given that the language provided these primitives, but in some programming languages locks and other synchronization primitives are instead implemented

from even more basic atomic operations. Moreover, some concurrent data structures, like the ones discussed in §1.1, use these basic atomic operations directly instead of higher-level primitives like locks.

An alternative called *rely-guarantee logic* [63], which was developed by Jones long before concurrent separation logic, has some advantages for reasoning in certain situations like these. The idea behind rely-guarantee logic is to specify how threads may interfere with one another. These specifications are given in the form of *rely* and *guarantee* assertions. When verifying a thread, the rely assertion describes what the effects of *other* threads can be (hence, what the thread under consideration can "rely" on), and the guarantee assertion describes the effects of the thread itself (that is, what it "guarantees" to other threads). Naturally, for this kind of reasoning to be sound, the rules of the logic must ensure that these assertions are coherent: what one thread relies on must indeed be guaranteed by the others. This style of reasoning is natural when one wants to model fine-grained interactions between concurrent threads that may not necessarily protect access to an entire data structure with a lock. However, a disadvantage of rely-guarantee logic relative to concurrent separation logic is that one cannot reason about these rely/guarantee conditions locally, because they make reference to the whole global state of the program.

A logic combining the benefits of rely-guarantee reasoning and concurrent separation logic was thus a natural goal, and Vafeiadis and Parkinson [124] and Feng et al. [39] independently proposed such combinations. By now, a number of concurrency logics have been developed incorporating or extending these and other ideas for reasoning about fine-grained uses of concurrency (e.g. [34, 44, 61, 90], among many others).

Parkinson [96] suggested that the trend of having new logics for each new aspect of concurrency was unsatisfying, and that what was needed was a single logic expressive enough to handle all of these use cases. In part, he argued that this would be beneficial because it would remove the need to produce new soundness theorems for each logic, which often required substantial work.

Since then, attempts have been made at exactly this kind of unification with a simple foundation. Among these, Dinsdale-Young et al. [35] noted that many reasoning patterns from concurrency logics could be encoded in a logic that provided (1) a means of specifying an abstract notion of a resource that could be owned by threads, represented as elements of a monoid, (2) assertions to encode how threads can manipulate and modify these resources, and (3) a way to connect these abstract resources to the actual physical state of the program. This was pushed further by Jung et al. [64] in a logic called Iris [65, 72], which again provided the ability to specify resources using monoids, but offered a more flexible way to encode the relationship between abstract resources and physical state.

Using this idea of partial commutative monoids as "resources", it becomes possible to *encode* relational reasoning in the logic. The idea, originally developed by Turon et al. [122] and subsequently used and extended in [75, 115] in the setting of Iris, is to treat threads from one program (the "source") as a kind of resource that can be owned by threads of another program (the "target"). One starts by defining an assertion $\mathsf{source}(i, e)$ which says that in the source program execution, thread identifier i is executing the expression e. Then there are assertions like $p \mapsto_{\mathsf{s}} v$ which describe the source program's heap. Finally, one has rules for simulating

steps of that source program, such as:

$$(\mathsf{source}(i, l := w; e) * l \mapsto_{\mathsf{s}} v) \Rrightarrow (\mathsf{source}(i, e) * l \mapsto_{\mathsf{s}} w)$$

which executes an assignment in the source program, where the $\Rrightarrow$ connective, called a "view-shift", can be thought of as a kind of implication in which one may transform these abstract resources representing threads. Then the following triple:

$$\{\mathsf{source}(i, e_1)\}\ e_2\ \{v.\ \mathsf{source}(i, v)\}$$

implies that if e_2 terminates with value v, there is an execution in which e_1 terminates with v.

The advantage of having these source threads as assertions rather than the four-place judgment of Benton's relational Hoare logic is that in the concurrent setting, one target program thread may simulate multiple source program threads, or the relationship between threads may change over time, so we want the ability to transfer these threads just as we can transfer ownership of physical resources like a memory location in the heap.

1.2.4 Probabilistic Logics

Given the importance of randomized algorithms, a number of program logics have been developed for reasoning about programs with the ability to make probabilistic choices.

Ramshaw [101] presented a system like Hoare logic extended with a primitive assertion of the form $\mathsf{Fr}(P) = p$ which says that P holds with "frequency" p. This can be thought of as a kind of assertion about the probability that P holds, except that unlike probabilities, these frequencies are not required to sum to 1. Several program logics featuring explicit probabilistic assertions have subsequently been developed [17, 30, 102].

Kozen [71] argued that instead of having separate assertions for specifying the probabilities, *all* assertions should be interpreted probabilistically. That is, rather than interpreting assertions as being true or false, they should denote a value p representing the probability that they are true. He developed a variant of dynamic logic called probabilistic propositional dynamic logic based on this idea. This "quantitative" approach was developed further by Morgan et al. [86] who describe a probabilistic variant of Dijkstra's weakest precondition calculus. There, they consider a language with a probabilistic choice operator, $e_1 \oplus_p e_2$. This is a program that with probability p continues as e_1 and with probability $1 - p$ behaves as e_2. Then, reasoning about this kind of choice is done using the following rule for weakest preconditions:

$$\mathsf{wp}\ e_1 \oplus_p e_2\ \{P\} = p \cdot \mathsf{wp}\ e_1\ \{P\} + (1 - p) \cdot \mathsf{wp}\ e_2\ \{P\}$$

Many extensions and variants of this kind of quantitative approach have been suggested. For example, Kaminski et al. [68] extend this logic with the ability to count the number of steps taken by the program in order to derive expected time bounds.

An alternative approach is to not introduce probabilities at the level of assertions at all. Barthe et al. [13] describe a logic where the *judgment* for the Hoare triple is indexed by a probability. They write $\vdash_p \{P\}\ e\ \{Q\}$ for a judgment which says that if e is executed in a state satisfying P, upon termination Q will fail to hold with probability *at most* p. They argue that although this system may be less expressive, it is easier to use for certain examples.

Probabilistic Relational Hoare Logic (pRHL) [10, 14] takes another approach that avoids using probabilistic assertions. This logic is an extension of Benton's relational Hoare logic to programs with probabilistic choice. In Benton's work, the assertions P and Q in the judgment $\{P\}\ e_1 \sim e_2\ \{Q\}$ are relations on the start and end states of the two programs. In the probabilistic variant, each of the e_i will terminate in a *distribution* of states, so the relation Q is *lifted* to a relation on distributions. In particular, let S_1 and S_2 be sets of states for e_1 and e_2 respectively, and let $D(S_i)$ be the set of all distributions on S_i. Then, given a relation $Q \subseteq S_1 \times S_2$, the lifting $L(Q) \subseteq D(S_1) \times D(S_2)$ is a relation such that $(\mu_1, \mu_2) \in L(Q)$ if there exists $\mu \in D(S_1 \times S_2)$ for which:

1. If $\mu(a, b) > 0$ then $(a, b) \in Q$.
2. $\mu_1(a) = \sum_b \mu(a, b)$
3. $\mu_2(b) = \sum_a \mu(a, b)$

Then $\{P\}\ e_1 \sim e_2\ \{Q\}$ holds in pRHL if whenever e_1 and e_2 are executed in states related by P and terminate with a distribution of states μ_1 and μ_2, respectively, then $(\mu_1, \mu_2) \in L(Q)$.

As Barthe et al. [16] noted, a joint distribution μ satisfying the conditions above is what is known as a *coupling* between the distributions of states after executing e_1 and e_2. Couplings are a well-known proof technique in probability theory [80], and the existence of a coupling μ can often tell us useful information about the two distributions μ_1 and μ_2. For instance, suppose x and y are integer assignables in the states of the programs e_1 and e_2, and let X and Y be random variables for the values of x and y after the programs execute. Then if the quadruple $\{\mathsf{True}\}\ e_1 \sim e_2\ \{x \geq y\}$ holds, we can conclude that X *stochastically dominates* Y, that is:

$$\forall r, \Pr[X \geq r] \geq \Pr[Y \geq r]$$

In some applications for cryptography and security, establishing this kind of stochastic dominance (or related results), is precisely the desired specification. In other cases, this is useful because if we want to bound $\Pr[Y \geq r]$, it suffices to bound $\Pr[X \geq r]$. As usual with relational reasoning, ideally e_1 is simpler to reason about than e_2.

The important difference between pRHL and some of the logics described above is that assertions are not interpreted probabilistically during the verification, so explicit reasoning about probability is minimized. The key rule where probabilistic reasoning enters is when the two programs e_1 and e_2 in fact make a probabilistic choice[5]:

$$\frac{f \blacktriangleleft \mu_1, \mu_2}{\{P\}\ \mathsf{draw}(\mu_1) \sim \mathsf{draw}(\mu_2)\ \{(x, y).\ P \wedge y = f(x) \wedge x \in \mathsf{support}(\mu_1)\}}$$

where μ_1 and μ_2 are probability distributions on sets A_1 and A_2, respectively, $\mathsf{draw}(\mu)$ is the expression for sampling from a distribution, $f : A_1 \to A_2$ is a bijection, and $f \blacktriangleleft \mu_1, \mu_2$ holds if $\forall x \in A_1.\ \mu_1(x) = \mu_2(f(x))$.

[5]The version of this rule in the work by Barthe et al. [10] is different because their draw command mutates an assignable rather than returning a value. For consistency with the rest of this section, what is shown here is a simplified rule for an expression based language.

1.2.5 Combinations

Recently, some work has proposed logics that combine features from the different kinds of logics mentioned above.

McIver et al. [84] present a probabilistic version of rely-guarantee logic [63]. They use their logic to verify a "faulty" concurrent Sieve of Eratosthenes, in which threads remove numbers from a list to identify primes, with thread i removing multiples of $(i+1)$ – however, each thread only probabilistically removes the elements it is supposed to, and the goal is to give a lower bound on the probability that the resulting list only contains primes. Like the original rely-guarantee logic, this logic does not permit local reasoning: one must check stability against rely-guarantee conditions that refer to the global state of the program. This makes it hard to verify a data structure and provide an abstract specification that can be used by a client.

Independently of the work described in this dissertation, Batz et al. [19] developed a version of (non-concurrent) separation logic for reasoning about sequential probabilistic programs with dynamic memory allocation. They verify an example of a program which probabilistically appends nodes to a list (so that the length of the list is geometrically distributed), a tree deletion procedure which only probabilistically deletes nodes, and an algorithm that shuffles an array. Their logic is in the style of the other "quantitative" logics mentioned above, where assertions denote functions from program states to probabilities/expected values, and rules are given for computing and bounding these probabilities. Their logic features an analog of the frame rule, which says (under certain side conditions)

$$\mathsf{wp}\ e\ \{P\} * Q \leq \mathsf{wp}\ e\ \{P * Q\}$$

where the ordering here is the pointwise ordering of functions from states to $\mathbb{R}$. They argue that this is the natural analogue of the frame rule, because the ordering $\leq$ plays the role of the ordering given by entailment in non-quantitative separation logics.

1.2.6 Alternatives

The work discussed above fits, broadly speaking, into the tradition of Hoare logic. I now discuss some alternative formalisms and proof techniques outside this tradition.

Probabilistic Automata. One approach to reasoning about concurrent and distributed systems starts by modeling them as non-deterministic automata. Having formally specified the possible states and transitions of an automaton, one can reason about its behavior by establishing invariants that hold throughout execution. In addition, one can use a form of relational reasoning by exhibiting a *simulation relation* between the automata and a simpler one. See Lynch [81] for a very thorough explanation of this approach.

The thesis of Segala describes how to adapt this to *probabilistic* automata so as to model randomized distributed systems. Besides probabilistic simulation relations [107], an important proof technique developed in this work is the application of *coin lemmas* (which is elaborated on further in [106]). At a high level, the idea behind coin lemmas is that in the analysis of randomized distributed algorithms, one is often in the following situation: We want to determine a lower bound on the probability that the algorithm will "succeed" or enter a "good" state,

and we can identify a collection of probabilistic choices (called "experiments" in [106]), which, in the event they all occur, will completely determine whether the algorithm will succeed. If we could guarantee that regardless of the effects of non-determinism these experiments would all take place, then we could reduce the analysis of the algorithm to a stochastic process in which the outcomes of the experiments are random variables $X_1, \ldots, X_n$, and the success of the algorithm is described by a predicate R on their outcome. Then, we would just need a lower bound on $\Pr[R(X_1, \ldots, X_n)]$. However, suppose as a consequence of the effects of non-determinism, some of these probabilistic choices will not in fact occur in certain executions. Then $\Pr[R(X_1, \ldots, X_n)]$ is not necessarily a valid bound. However, it does remain a valid bound so long as we can show that for each execution where only some subset of the experiments take place, if it is possible to *assign* values to the other experiments so that R holds for the combined collection of outcomes, then the algorithm must have succeeded in that execution. The coin lemmas of [105] identify common examples of predicates R and experiments for which this is indeed the case.

Although approaches based on probabilistic automata have been used to analyse several sophisticated distributed algorithms, there is a gap between code written in a programming language and a model of an algorithm expressed as an automaton. How do we show that the code actually implements the algorithm described by the automaton? A relational program logic provides a way to do so.

Temporal Logic. An alternative family of logics based on *temporal logic* have been developed for verifying concurrent systems [78, 82, 83]. In these logics, assertions are understood as predicates on states of some system which varies over time. There are then various modalities for describing at which moments of time an assertion holds. For example, $\Box P$ says that P always holds. One can then define $\Diamond P \triangleq \neg \Box \neg P$, which means (classically) that P eventually holds. A plethora of additional modalities can be considered, some depending on whether time is viewed as a discrete sequence of steps or a continuum. In contrast to approaches based on separation logic, additional emphasis is put on specifying and proving *liveness* properties, that is, showing the program eventually performs certain actions, as opposed to merely proving *safety properties* that show the program never enters an invalid state. These logics are widely used in automated model checking systems [28]. Some of these logics have been extended with probabilistic assertions [51].

However, when it comes to reasoning about shared memory concurrency, temporal logic has the same shortcomings of Hoare logic that motivated the development of separation logic: we must manually specify that various memory cells are disjoint, and specifications have to mention that they leave particular locations unchanged. This makes it hard to give compositional specifications and reason locally about programs[6].

Linearizability. Instead of Hoare-style specifications, the principal correctness criterion considered in the community of concurrent algorithm designers is *linearizability* [53]. Roughly speaking, a concurrent object or data structure is linearizable if we can regard the execution of its operations as if they were atomic steps. Besides its intuitive appeal, this property is useful

[6]However, not all advocates of temporal logics see this as a pressing problem [77].

because it seems to suggest that clients using a data structure cannot tell the difference between a complex, efficient linearizable implementation and a slower or (practically unimplementable) version in which the operations truly are atomic.

One way to define this idea of indistinguishability formally is known as *contextual equivalence* [99]: We first define what it means for two complete programs e_1 and e_2 to be *observationally equivalent*, which is usually defined in terms of what values they can both reduce to or their termination behavior. Then, we imagine a client C as a "program with a hole", called a context. Given an expression e implementing the data structure, we define the operation of filling in the hole in C with e to obtain a complete program, which is written as $C[e]$. Two implementations e_1 and e_2 are said to be contextually equivalent if for all contexts C, $C[e_1]$ and $C[e_2]$ are observationally equivalent[7]. When considering non-deterministic systems, it is more natural to instead work with *observational refinement*, which says that the set of behaviors of one program is a subset of behaviors of the other, and then define an analogous notion of *contextual refinement*.

As Filipovic et al. [40] note, it had long been informally understood that linearizability seemed to imply contextual refinement between an implementation and an abstract version of the data structure in which all operations are atomic. Filipovic et al. [40] gave the first formal proof that this was indeed the case. Naturally, their proof is with respect to a particular programming language, since the semantics of the language affects what the contexts C can observe about implementations and what it means for programs to be equivalent. As we add features to the language considered by Filipovic et al. [40], there is no *a priori* guarantee that their result will continue to hold in the extended language.

Indeed, Golab et al. [49] show that if clients can make randomized choices, the distribution of values they return *can* differ when using a linearizable implementation of a data structure as compared to a truly atomic version. Because any reasonable notion of observational equivalence in the presence of randomization ought to take into account the distribution of values returned, this means that linearizability may not imply contextual refinement in languages with probabilistic choice. Golab et al. [49] propose instead *strong linearizability* and prove that under certain assumptions, this alternative notion suffices to ensure that the distribution of behaviors is once again indistinguishable[8]. However, in this result they do not permit *implementations* of a data structure to have randomized behavior, only *clients* can make randomized choices. In light of these issues, neither linearizability nor strong linearizability seem like appropriate correctness criteria for the kind of data structures described in §1.1.

1.3 Design Choices and Outline

In the previous sections, I have described some concurrent algorithms and surveyed a number of program logics. However, I argue that none of the logics discussed are expressive enough to

[7]Generally, one assigns types to contexts and expressions, and then in the definition of contextual equivalence, we only quantify over contexts C for which the combination of $C[e]$ will be well typed.

[8]The formalism considered by Golab et al. does not use the notion of contextual refinement or consider a concrete programming language. Rather, they describe concurrent systems abstractly in terms of sets of sequences of operations called "histories", which represent a partial execution.

verify the probabilistic properties of these example algorithms. All but one of them does not address the combination of concurrent and probabilistic reasoning, and the one exception [84] lacks the local reasoning features of modern concurrency logics.

In this section, I describe several choices I have made about the design of Polaris, the logic that will be presented in this dissertation. As mentioned above, this design is summarized by the following thesis:

> Separation logic, extended with support for probabilistic relational reasoning, provides a foundation for the verification of concurrent randomized programs.

I now offer some rationale for the choices implicit in this statement.

Why separation logic? In order to reason about probabilistic properties of concurrent randomized algorithms, one first needs to prove the same kinds of intermediate properties that are used to establish functional correctness. For example, one must show that synchronization primitives are used appropriately to maintain assorted invariants of data structures. I think the effectiveness of separation logic for reasoning about a wide range of concurrent programs is clear now, and there is a trend toward some degree of stabilization and unification of ideas from these logics. Moreover, it is worthwhile to build on a "state of the art" concurrency logic to ensure that we have the features needed to reason about fine-grained concurrent data structures. To that end, Polaris is an extension of Iris [64], and the extensions are done in a way to ensure that all the original proof rules of Iris remain sound.

The choice of how to do probabilistic reasoning is less obvious. As we have seen in the previous section, there are many Hoare-like program logics for reasoning about probability with a variety of features and approaches. I believe the most appropriate choice is the relational style, because I think we want to use a program logic only for the purposes of abstracting away from low level details of the program, and then use whatever tools from probability theory are needed to analyze the higher level model of the problem. This choice is preferable for several reasons:

1. **Relational reasoning is closer to the style used in pencil-and-paper proofs.**

   As I have already stressed, the common approach to analyzing these algorithms is to consider a mathematical abstraction of the algorithm's behavior. Therefore, it seems desirable to have a logic where we can prove that there is a relation between a concrete implementation and such an abstraction.

2. **Relational reasoning is appropriate when we cannot formulate a single specification that captures all aspects of an algorithm.**

   When considering the behavior of probabilistic algorithms, there are many properties of interest: expected values, variances, tail-bounds, rates of convergence to asymptotic distributions, and so on. Therefore, we will almost certainly want to carry out multiple proofs about a given algorithm. However, we do not want to re-prove basic facts about the correctness of the algorithm each time, such as showing that a particular pointer is not null or that there will be no data-race when accessing some field. If we prove once and for all that the concurrent algorithm is modeled by some more abstract "pure" stochastic

process, then subsequent mathematical analysis only needs to consider this stochastic process.

3. **Probability theory is too diverse to embed synthetically in a logic.**

   In analogy to synthetic differential geometry or synthetic homotopy theory, I consider the union bound logic of Barthe et al. [13] as a kind of "synthetic" approach to a fragment of probability theory: the rules of the logic codify a common use of the union bound in probability theory, with minimal mention of explicit probabilities.

   Although appealing in some ways, I think this approach is susceptible to the criticisms Parkinson [96] raised about the proliferation of concurrent separation logics. That is, one can envision dozens of specialized logics each trying to encode some commonly used technique from probability theory: Chernoff bounds, Doob's optional stopping theorem, Wald's lemma, etc. Unfortunately, it seems the analysis of randomized algorithms is too diverse to encapsulate in some single synthetic logic.

   The approach taken in [17] is to "embed" several specialized synthetic logics in some more expressive general logic with probabilistic assertions. This seems promising, but I believe the right setting for such a logic would be in reasoning about more abstract representations of a stochastic program which does not involve low-level details like pointers or concurrent interleavings. That is, we should first use relational reasoning techniques to simplify the program under consideration.

Of course, a full argument in favor of my thesis will be contained in the rest of this dissertation, which is as follows.

Because I have argued in favor of a relational style, we first need some way to express the more mathematically abstract version of programs we want to reason about. As the purpose of doing this is to end up with something that is easier to reason about, we do not necessarily want to express this abstract version in the same programming language we started with. Instead, we will write the abstract program in a monadic style, using a monad for the combination of probabilistic and non-deterministic choice proposed by Varacca and Winskel. I describe this monadic construction in Chapter 2.

As mentioned above, Polaris is an extension of the Iris logic. Therefore, before the extensions can be explained, I must first give some background on Iris. As will become apparent, Iris provides language-generic components that can be used to derive program logics for specific languages. Chapter 3 provides a brief introduction to Iris by presenting a particular instantiation of the framework with a concurrent ML-like language. Then, Chapter 4 describes the more general set-up and discusses how soundness of the logic is proved.

Next, I describe in Chapter 5 the new probabilistic extensions constituting Polaris. The purpose of relational reasoning in the extended logic is to establish a connection between the behavior of a concrete program and a monadic computation expressed using the monad described in Chapter 2. Once again, Polaris is parameterized by the semantics of a probabilistic concurrent language, about which few assumptions are made. In fact, specifying the semantics of a language with a combination of concurrency and probabilistic choice involves some subtleties. The formulation I give has some restrictions, but it suffices for the examples we are

interested in here. After describing the semantics, I turn to the new proof rules and soundness statement for the probabilistic extension. The generic framework is instantiated with a concurrent ML-like language, this time with primitives for generating random booleans.

Chapter 6 describes how to use the logic, instantiated with this language, to reason about two of the examples presented in §1.1: approximate counters and skip lists. For each example, we follow the pattern of first writing down a monadic model of the algorithm, using the program logic to establish a relationship between the concrete program and the monadic model, and then analyzing the behavior of the monadic model.

Finally, Chapter 7 summarizes the development and suggests some directions for future work.

1.4 Verification and Foundations

All of the new results herein have been formally verified in the Coq theorem prover [117]. This includes not only the soundness of Polaris, but also the example programs that will be verified using the logic. The only things not verified are the simple examples considered in the tutorial introduction to Iris. The proofs use the Coq standard library axiomatization of the reals, along with two axioms for classical reasoning: the law of the excluded middle, and the axiom of constructive indefinite description, which is a choice principle similar to Hilbert's epsilon operator. These axioms are used to reason about probabilities.

With the hope of minimizing discrepancies between the machine checked proofs and the written versions in this dissertation, I will write the latter in the style of an "informal type theory" with classical reasoning principles. Thus, I will speak of mathematical objects as having types, as opposed to being elements of sets. This informal type theory is intended to correspond to the Calculus of Inductive Constructions (CiC) [97], the formal type theory underlying Coq, extended with the axioms mentioned above. In addition to a hierarchy of predicative types $\mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$, there is an additional impredicative type Prop of propositions. Given $P : \mathsf{Prop}$, we say that P holds if there is a term of type P. The type $\mathsf{Unit} : \mathsf{Type}_0$ consists of a single term $()$.

In the Calculus of Inductive Constructions, one generally cannot eliminate existentials in Prop when defining terms whose type does not belong to Prop. That is, in standard CiC, knowing that $\forall n : \mathbb{N}.\, \exists n : \mathbb{N}.\, P(n)$ holds does not provide any means to define a corresponding *function* $f : \mathbb{N} \to \mathbb{N}$ with the property that for all n, $P(f(n))$ holds. However, the above mentioned axiom of constructive indefinite description extends CiC with just such a principle. Namely, it says that given a type T, a predicate $P : T \to \mathsf{Prop}$, and a proof that $\exists t : T.\, P(t)$, we may derive a term of type T for which the predicate holds.

When I refer to a "set of terms of type T", I mean a predicate $A : T \to \mathsf{Prop}$, where the terms t such that $A(t)$ holds are thought of as being "in" the set. Then, union of such sets is defined to be disjunction of the predicates, intersection is conjunction, and so on.

Throughout this dissertation, I will use the "horizontal bar" inference rule notation. When I write something like

$$\frac{A_1 \qquad \ldots \qquad A_n}{B}$$

I mean that there is a term of type $A_1 \to \cdots \to A_n \to B$. Hence, one can conclude B by showing all of A_1 through A_n. Often collections of such rules will occur in a figure, where all of the types below the horizontal lines will have similar schematic form. Such a list of rules should *not* be construed as an inductive definition, unless otherwise stated. Rather, the figure simply means that there are terms corresponding to all of the rules occurring therein.

Chapter 2

Monadic Representation

A common approach to reasoning about effectful programs is to model effects using a suitable monad M. One represents an effectful program that returns a value of type T as a term of type $M(T)$. Next, one usually proves a series of equational rules for simplifying terms of type $M(T)$, and other lemmas for reasoning about such terms. This approach has been used for reasoning about a number of effects, including: state [89, 113], non-termination [27], non-determinism [46], probabilistic choice [7, 46, 98, 126], and even the combination of non-deterministic and probabilistic choice [46].

What is nice about using this representation in a dependently typed proof assistant is that, except at points where effectful operations are performed, such terms are composed of "just" pure terms of the appropriate type, which we can write and reason about using all of the standard facilities of the theorem prover. And, when effects are used, we (ideally) have a clean equational theory for reasoning about them. Thus, such a representation makes an ideal candidate for the more abstract way of expressing concurrent probabilistic algorithms which I motivated in Chapter 1. We need then a monad for representing the combination of non-deterministic choice (to model the effects of concurrency) and probabilistic choice.

This turns out to be challenging to obtain, for reasons we begin with. After explaining the difficulties, we discuss the monad of indexed valuations, due to Varacca and Winskel [128], which can be used to obtain a monad for both probabilistic and non-deterministic choice. In their original presentation of the monad, Varacca and Winskel focused on its equational properties and on using it to give an adequate denotational model for a certain programming language. However, because our eventual goal is to be able to derive quantitative bounds on things like probabilities and expected values, I define such notions for these monadic computations and develop some rules for calculating and bounding them. We then consider how to generalize the notion of coupling (alluded to in Chapter 1) to this setting. Finally, some alternative denotational models combining non-deterministic and probabilistic choice are described.

2.1 Background

Let us start by recalling common monadic encodings for non-deterministic and probabilistic choice (separately).

2.1.1 Non-deterministic Choice

For non-determinism, we can define $M_{\mathsf{N}}(T)$ as the type consisting of predicates $A : T \to \text{Prop}$ for which there exists at least some $t : T$ such that $A(t)$ holds. We think of these predicates as non-empty sets of terms of type T, where each element of the set represents one of the different non-deterministic outcomes. We say two terms A and B of type $M_{\mathsf{N}}(T)$ are equivalent, written $A \equiv B$, if their sets of elements are the same: for all x, $x \in A \leftrightarrow x \in B$. In addition to the standard monadic operations (bind and return), we can represent non-deterministic choice between two computations A and B as the union $A \cup B$ of the two sets, defined by:

$$A \cup B \equiv \lambda t.\, A(t) \vee B(t)$$

This operation satisfies a number of natural rules:

$$A \cup B \equiv B \cup A \qquad A \cup (B \cup C) \equiv (A \cup B) \cup C \qquad A \cup A \equiv A$$

These, along with the usual monad laws, can be used to prove that one non-deterministic computation is equivalent to another.

2.1.2 Probabilistic Choice

We can represent a (discrete) probabilistic computation of type T as a function $f : T \to [0, 1]$, mapping values of type T to the probabilities that they occur. The *support* of f, written $\mathsf{supp}(f)$ is the set of t such that $f(t) > 0$. Naturally, we want the sum of all the probabilities for the values in $\mathsf{supp}(f)$ to be equal to 1. In order to make sense of such an infinite sum, we require the support to be countable. We define $M_{\mathsf{P}}(T)$ to be the type of all functions $f : T \to [0, 1]$ such that $\mathsf{supp}(f)$ is countable and

$$\sum_{x \in \mathsf{supp}(f)} f(x) = 1$$

Given $A, B : M_{\mathsf{P}}(T)$, we say $A \equiv B$ if for all x, $A(x) = B(x)$. We can define an operation which selects between a computation A with probability p and another computation B of the same type with probability $(1 - p)$:

$$A \oplus_p B \triangleq \lambda x.\, p \cdot A(x) + (1 - p) \cdot B(x)$$

This operation satisfies equational rules such as:

$$A \oplus_p B \equiv B \oplus_{1-p} A \qquad A \oplus_p A \equiv A$$

2.1.3 Obstructions to Combination

In order to reason about programs that use both probability and non-determinism, we would like some way to *combine* the monads we have just examined. We might try to represent computations of type T combining both effects as terms of type $M_{\mathsf{N}}(M_{\mathsf{P}}(T))$, *i.e.,* non-empty sets of probability distributions.

But how do we define the monad operations for this combination? One way to derive the monad operations for a combination of two monads is to specify a *distributive law* [20]. In so doing, we can specify how the two effects interact. For example, the following equational rule:

$$A \oplus_p (B \cup C) \equiv (A \cup B) \oplus_p (A \cup C)$$

says that probabilistic choice distributes over non-deterministic choice. We can interpret this as saying that it does not matter whether we resolve the non-deterministic choice between B and C before or after the outcome of the probabilistic choice between the alternative A. Such a property seems natural, if we adopt the "adversarial" perspective outlined in §1.1: to an adversary trying to maximize/minimize some probability or expected value by non-deterministically selecting between B and C, the preferable alternative should not depend on the outcome of the probabilistic choice.

However, Varacca and Winskel [128] have given a proof (based on an idea they attribute to Plotkin) that no distributive law exists between the monads[1] M_{N} and M_{P}. For our purposes, it is not necessary to understand this impossibility proof. Instead, we can consider whether by changing the monads, we might be able to obtain a distributive law. Varacca and Winskel show that a result by Gautam [45] implies that the most natural way of combining these two monads cannot work so long as we expect the following equational law to hold:

$$A \oplus_p A \equiv A$$

Thus, perhaps the solution is to come up with a monad for probabilistic choice in which this equivalence does not hold. But might we not be giving up too much? At first this equivalence seems like something we want to retain: if in either case we choose A, then the probabilistic choice was irrelevant. However, when we later add in the effect of non-determinism, the absence of this law becomes more justifiable, because it allows us to account for the fact that subsequent non-determinism in the computation can be *resolved differently* on the basis of this seemingly irrelevant probabilistic choice. For example, a scheduler could observe the outcome of this internal probabilistic choice and use it as a basis for ordering subsequent operations in a larger computation.

2.2 Indexed Valuations

Using their observations about impossibility results, Varacca and Winskel describe an alternative way of representing probabilistic choice, which they call the *indexed valuation monad*, in which the problematic equivalence $A \oplus_p A \equiv A$ does not hold. They then describe a distributive law between M_{N} and the monad of indexed valuations to obtain a monad combining both effects.

Definition 2.1. An *indexed valuation* $\mathbb{I}$ of type T is a tuple (I, d, v), where

[1] More precisely, they consider the case where M_{N} is the monad of *finite* non-empty sets of terms of type T, and M_{P} consists of *finite* distributions, instead of countable ones. However, the impossibility of a distributive law in the finitary case precludes one for the non-finitary versions we have defined.

- I is a countable type[2] whose terms are called *indices*,
- d is a function of type $I \to T$, and
- v is a function of type $I \to \mathbb{R}^{\geq 0}$ such that[3]:

$$\sum_{i:I} v(i) = 1$$

Informally, we can think of the indices as a collection of "codes" or identifiers, the v function gives the probability of a particular index occurring, and d maps these codes to elements of type T. Importantly, the d function is not required to be injective, so that different codes can lead to the same observable result. We write $M_{\mathsf{I}}(T)$ for the type of indexed valuations of type T and define the following projections for the components of an indexed valuation:

$$\begin{aligned}\pi_{\mathsf{idx}}\,(I, d, v) &= I \\ \pi_{\mathsf{dec}}\,(I, d, v) &= d \\ \pi_{\mathsf{val}}\,(I, d, v) &= v\end{aligned}$$

The *indicial support*[4] of a valuation $\mathbb{I}$, notated $\mathsf{isupp}(\mathbb{I})$, is the set of indices i for which

$$\pi_{\mathsf{val}}(\mathbb{I})(i) > 0$$

We say $\mathbb{I}_1 \equiv \mathbb{I}_2$ if there exists a bijection $h : \mathsf{isupp}(\mathbb{I}_1) \to \mathsf{isupp}(\mathbb{I}_2)$ such that for all $i \in \mathsf{isupp}(\mathbb{I}_1)$:

$$\begin{aligned}\pi_{\mathsf{val}}(\mathbb{I}_1)(i) &= \pi_{\mathsf{val}}(\mathbb{I}_2)(h(i)), \text{ and} \\ \pi_{\mathsf{dec}}(\mathbb{I}_1)(i) &= \pi_{\mathsf{dec}}(\mathbb{I}_2)(h(i))\end{aligned}$$

That is, the bijection can only "relabel" indices in a way that preserves their probabilities and what they decode to.

There is a map H which takes indexed valuations of type T to elements of $M_{\mathsf{P}}(T)$:

$$H(I, d, v) = \lambda x. \sum_{i \in d^{-1}(\{x\})} v(i)$$

In other words, the probability of x in the resulting distribution is the sum of the probabilities of indices that decode to x. It is clear that if $\mathbb{I}_1$ and $\mathbb{I}_2$ are two indexed valuations such that

[2] A type T is said to be countable if there are functions $g : T \to \mathbb{N}$ and $f : \mathbb{N} \to \mathsf{Option}\ T$ such that for all t, $f(g(t)) = \mathsf{Some}\ t$.

[3] In fact, Varacca and Winskel first define a more general structure in which the sums of $v(i)$ do not have to equal 1, and the indices need not be countable. After working out some of the theory of these more general objects, they restrict to the subcategory where the indices are finite sets and the probabilities sum to 1. We will not restrict to finite sets of indices, since by letting them be countable we can model sampling from arbitrary discrete distributions.

[4] Varacca and Winskel call this simply the "support" of the valuation, however I prefer to use that term for something different, defined below.

$\mathbb{I}_1 \equiv \mathbb{I}_2$, then $H(\mathbb{I}_1) \equiv H(\mathbb{I}_2)$. However, the converse is not true because $\mathbb{I}_1$ and $\mathbb{I}_2$ could have indicial supports with different cardinalities.

Given a term t of type T, the indexed valuation ret t is defined to be $(\mathsf{Unit}, \lambda x.\, t, \lambda x.\, 1)$. That is, the type of codes is a singleton whose sole element decodes to t and occurs with probability 1.

If $\mathbb{I}$ is an indexed valuation of type T_1 and f is a function from T_1 to indexed valuations of type T_2, then we define bind $f\ \mathbb{I}$ to be the indexed valuation (I, d, v), where:

$$I = \left\{ (i_1, i_2) \mid i_1 \in \pi_{\mathsf{idx}}(I_1) \wedge i_2 \in \pi_{\mathsf{idx}}\Big(f\big(\pi_{\mathsf{dec}}(\mathbb{I})(i_1)\big)\Big) \right\}$$
$$d = \lambda(i_1, i_2).\, \pi_{\mathsf{dec}}\Big(f\big(\pi_{\mathsf{dec}}(\mathbb{I})(i_1)\big)\Big)(i_2)$$
$$v = \lambda(i_1, i_2).\, \pi_{\mathsf{val}}(\mathbb{I})(i_1) \cdot \pi_{\mathsf{val}}\Big(f\big(\pi_{\mathsf{dec}}(\mathbb{I})(i_1)\big)\Big)(i_2)$$

Intuitively, the idea behind these definitions is that they represent first sampling a code i_1 from $\mathbb{I}$, decoding it to get some term t, and then sampling a code i_2 from the indexed valuation $f(t)$, with the final value returned being whatever i_2 decodes to. Thus the type of indices consists of dependent pairs (i_1, i_2), where i_1 is an index from $\mathbb{I}$, and i_2 is an index of $f\big(\pi_{\mathsf{dec}}(\mathbb{I})(i_1)\big)$. The probability of obtaining the code (i_1, i_2) is the product of the probabilities of obtaining i_1 and i_2. As usual, the notation $x \leftarrow \mathbb{I};\ f(x)$ is defined to be bind $f\ \mathbb{I}$.

Given an indexed valuation $\mathbb{I} : M_{\mathsf{I}}(T)$, we define $\mathsf{idxOf}(\mathbb{I}) : M_{\mathsf{I}}(\pi_{\mathsf{idx}}(T))$ as

$$(\pi_{\mathsf{idx}}(\mathbb{I}), \lambda x.\, x, \pi_{\mathsf{val}}(\mathbb{I}))$$

This indexed valuation behaves like $\mathbb{I}$, except that rather than decoding indices, it simply returns them.

The probabilistic choice between two indexed valuations is then defined by:

$$(I_1, d_1, v_1) \oplus_p (I_2, d_2, v_2) \triangleq (I_1 + I_2, d', v')$$

where:

$$d'(i) = \begin{cases} d_1(i') & \text{if } i = \mathsf{inl}(i') \\ d_2(i') & \text{if } i = \mathsf{inr}(i') \end{cases}$$
$$v'(i) = \begin{cases} p \cdot v_1(i') & \text{if } i = \mathsf{inl}(i') \\ (1-p) \cdot v_2(i') & \text{if } i = \mathsf{inr}(i') \end{cases}$$

When we construct an indexed valuation using these operations, the indices record a kind of history or trace of the execution, logging the intermediate outcomes used to derive the final value. For example, if we consider the computation

$$\big(\mathsf{ret\ true} \oplus_p \mathsf{ret\ false}\big) \oplus_p \big(\mathsf{ret\ true} \oplus_p \mathsf{ret\ false}\big)$$

the type of indices will be $(\mathsf{Unit} + \mathsf{Unit}) + (\mathsf{Unit} + \mathsf{Unit})$. For this computation, the index $\mathsf{inl}(\mathsf{inr}(()))$ will decode to false, and represents an execution in which the outermost probabilistic choice yields the left operand, and then the probabilistic choice within that sub-computation yields its right operand.

One can show that for all indexed valuations $\mathbb{I}_1$ and $\mathbb{I}_2$ and $0 \leq p \leq 1$, we have $\mathbb{I}_1 \oplus_p \mathbb{I}_2 \equiv \mathbb{I}_2 \oplus_{1-p} \mathbb{I}_1$. However, unlike the original probabilistic choice monad discussed above, $\mathbb{I} \oplus_p \mathbb{I} \not\equiv \mathbb{I}$, unless $p = 0$ or $p = 1$. The reason is that when p is neither 0 nor 1, the indicial support of $\mathbb{I} \oplus_p \mathbb{I}$ will have a larger cardinality than the indicial support of $\mathbb{I}$, so there can be no bijection between them. Recall that we do *not* want this equivalence to hold, because it rules out the existence of the distributive law we want.

Indeed, since this rule does not hold, it is possible to define appropriate monad operations on $M_{\mathsf{N}} \circ M_{\mathsf{I}}$, and we write M_{NI} for this composition. The approach followed by Varacca and Winskel is to define a distributive law, from which the corresponding monad is derived from general theorems about distributive laws. However, we will instead specify the monad operations directly.

First, we define a notion of equivalence for elements of $M_{\mathsf{NI}}(T)$. Given two non-empty sets of indexed valuations, $\mathcal{I}_1$ and $\mathcal{I}_2$, we say $\mathcal{I}_1 \equiv \mathcal{I}_2$ if for each $\mathbb{I}_1 \in \mathcal{I}_1$, there exists some $\mathbb{I}_2 \in \mathcal{I}_2$ such that $\mathbb{I}_1 \equiv \mathbb{I}_2$, and vice versa.

Let $\mathcal{I}$ be a non-empty set of indexed valuations of type T_1, and let f have type $T_1 \to M_{\mathsf{NI}}(T_2)$. Then bind $f\ \mathcal{I}$ is the set of all $\mathbb{I} : M_{\mathsf{I}}(T_2)$ for which there exists $\mathbb{I}_0 \in \mathcal{I}$ and $h : \pi_{\mathsf{idx}}(\mathbb{I}_0) \to M_{\mathsf{NI}}(T_2)$ such that:

1. $\mathbb{I} \equiv x \leftarrow \mathsf{idxOf}(\mathbb{I}_0)\,;\ h(x)$, and
2. For all $i \in \mathsf{isupp}(\mathbb{I}_0)$, the indexed valuation $h(i)$ is in $\pi_{\mathsf{idx}}\Big(f\big(\pi_{\mathsf{dec}}(\mathbb{I}_0)(i)\big)\Big)$

In other words, the elements of bind $f\ \mathcal{I}$ are equivalent to indexed valuations which first sample some index i from an element $\mathbb{I}_0$ of $\mathcal{I}$ and then select an element from $f(\pi_{\mathsf{dec}}(\mathbb{I}_0)(i))$ and sample from that. The decision about which element of $f(\pi_{\mathsf{dec}}(\mathcal{I}_0)(i))$ is chosen is determined by a function h. We can think of this function h as representing the strategy of the adversary that this non-deterministic choice is modeling. Crucially, h is a function from the indices of $\mathbb{I}_0$, *not* just what they decode to.

For a term t of type T, we define ret $t : M_{\mathsf{NI}}(T)$ to be the singleton set containing just the indexed valuation $(\{()\}, \lambda x.\, t, \lambda x.\, 1)$.

Given $\mathcal{I}_1$ and $\mathcal{I}_2$ of type $M_{\mathsf{NI}}(T)$, the probabilistic choice operation $\mathcal{I}_1 \oplus_p \mathcal{I}_2$ is defined by taking the pairwise probabilistic choice of each indexed valuation in the respective sets:

$$\mathcal{I}_1 \oplus_p \mathcal{I}_2 \equiv \{\mathbb{I}_1 \oplus_p \mathbb{I}_2 \mid \mathbb{I}_1 \in \mathcal{I}_1, \mathbb{I}_2 \in \mathcal{I}_2\}$$

and non-deterministic choice between $\mathcal{I}_1$ and $\mathcal{I}_2$ is simply the union $\mathcal{I}_1 \cup \mathcal{I}_2$ of the two sets.

Given an indexed valuation $\mathbb{I} : M_{\mathsf{I}}(T)$, the singleton set $\{\mathbb{I}\}$ has type $M_{\mathsf{NI}}(T)$. Taking a singleton set of an indexed valuation commutes with the operations on $M_{\mathsf{I}}(T)$. For example, $\{\mathbb{I}_1\} \oplus_p \{\mathbb{I}_2\} \equiv \{\mathbb{I}_1 \oplus_p \mathbb{I}_2\}$, and similarly for bind and return. We say that a computation $\mathcal{I} : M_{\mathsf{NI}}(T)$ is a singleton when there exists $\mathbb{I}$ such that $\mathcal{I} \equiv \{\mathbb{I}\}$.

The choice operations and the monad operations respect the equivalence relation we defined above. A selection of additional equational rules are shown in Figure 2.1 (the standard monad laws are omitted).

Example 2.2 (Modeling approximate counters)**.** In Figure 2.2 we show how to model the approximate counter code from Figure 1.1c using this monad. The approxIncr computation first

$$\mathcal{I}_1 \oplus_p \mathcal{I}_2 \equiv \mathcal{I}_2 \oplus_{1-p} \mathcal{I}_1 \qquad \mathcal{I}_1 \oplus_1 \mathcal{I}_2 \equiv \mathcal{I}_1 \qquad \mathcal{I} \cup \mathcal{I} \equiv \mathcal{I} \qquad \mathcal{I}_1 \cup \mathcal{I}_2 \equiv \mathcal{I}_2 \cup \mathcal{I}_1$$

$$\mathcal{I}_1 \cup (\mathcal{I}_2 \cup \mathcal{I}_3) \equiv (\mathcal{I}_1 \cup \mathcal{I}_2) \cup \mathcal{I}_3 \qquad \mathcal{I}_1 \oplus_p (\mathcal{I}_2 \cup \mathcal{I}_3) \equiv (\mathcal{I}_1 \oplus_p \mathcal{I}_2) \cup (\mathcal{I}_1 \oplus_p \mathcal{I}_3)$$

$$x \leftarrow \mathcal{I}_1 \cup \mathcal{I}_2 \,;\; F(x) \equiv (x \leftarrow \mathcal{I}_1 \,;\; F(x)) \cup (x \leftarrow \mathcal{I}_2 \,;\; F(x))$$

$$x \leftarrow \mathcal{I}_1 \oplus_p \mathcal{I}_2 \,;\; F(x) \equiv (x \leftarrow \mathcal{I}_1 \,;\; F(x)) \oplus_p (x \leftarrow \mathcal{I}_2 \,;\; F(x))$$

$$x \leftarrow \{\mathbb{I}_1\} \,;\; y \leftarrow \{\mathbb{I}_2\} \,;\; F(x,y) \equiv y \leftarrow \{\mathbb{I}_2\} \,;\; x \leftarrow \{\mathbb{I}_1\} \,;\; F(x,y)$$

Figure 2.1: Equational laws for $M_{\mathsf{N}} \circ M_{\mathsf{I}}$ monad.

$$\begin{array}{l|l}
\mathsf{approxIncr} \triangleq & \mathsf{approxN}\ 0\ z \triangleq \mathsf{ret}\ z \\
\quad k \leftarrow \mathsf{ret}\ 0 \cup \cdots \cup\ \mathsf{ret}\ \mathsf{MAX}\,; & \mathsf{approxN}\ (n+1)\ z \triangleq \\
\quad \mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0 & \quad k \leftarrow \mathsf{approxIncr}\,; \\
 & \quad \mathsf{approxN}\ n\ (z+k)
\end{array}$$

Figure 2.2: Monadic encoding of approximate counter algorithm from Figure 1.1c.

non-deterministically selects a number k up to MAX – this models the process of taking the minimum of the value in l and MAX in the code. The non-determinism accounts for the fact that the value that will be read depends on what other threads do. The monadic encoding then makes a probabilistic choice, returning $k+1$ with probability $\frac{1}{k+1}$ and 0 otherwise, which represents the probabilistic choice that the code will make about whether to do the fetch-and-add.

Finally, the process of repeatedly incrementing the counter n times is modeled by approxN. The first argument n tracks the number of pending increments to perform, and the second argument l accumulates the sum of the values returned by the calls to approxIncr. Note that this model *does not* try to represent multiple threads in the middle of an increment each waiting to add its value to the shared count – rather, it is *as if* the actual calls to `incr` all happened atomically in sequential order, with the effects of concurrency captured by the non-determinism in the approxIncr computation.

Of course, we need to show that this model accurately captures the behavior of the code from Figure 1.1c – this is what the program logic we describe in Chapter 5 will do.

2.3 Expected Values

With what we have described so far, we can express computations with randomness and non-determinism and derive equivalences between them, but we do not yet have a way to talk about the standard concerns of probability theory (*e.g.,* expected values, variances, tail bounds).

Given an indexed valuation $\mathbb{I} = (I, d, v)$ of type T and a function $f : T \to \mathbb{R}$, we can define

$$\frac{\mathbb{I}_1 \equiv \mathbb{I}_2}{\mathbb{E}_f[\mathbb{I}_1] = \mathbb{E}_f[\mathbb{I}_2]} \qquad \mathbb{E}_f[\mathsf{ret}\ v] = f(v) \qquad \textsc{Ex-Linear}\ \frac{k \geq 0}{\mathbb{E}_{(\lambda x.k \cdot f(x)+c)}[\mathbb{I}] = k \cdot \mathbb{E}_f[\mathbb{I}] + c}$$

$$\mathbb{E}_f[\mathbb{I}_1 \oplus_p \mathbb{I}_2] = p \cdot \mathbb{E}_f[\mathbb{I}_1] + (1-p) \cdot \mathbb{E}_f[\mathbb{I}_2] \qquad \textsc{Ex-Comp}\ \mathbb{E}_{g \circ f}[\mathbb{I}] = \mathbb{E}_g[x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)]$$

$$\frac{\forall x.\, k_1 \leq \mathbb{E}_f[F(x)] \leq k_2}{k_1 \leq \mathbb{E}_f[x \leftarrow \mathbb{I}_1;\ F(x)] \leq k_2} \qquad \textsc{Ex-Mono}\ \frac{\forall x.\, f_1(x) \leq f_2(x)}{\mathbb{E}_{f_1}[\mathbb{I}] \leq \mathbb{E}_{f_2}[\mathbb{I}]}$$

Figure 2.3: Selection of rules for calculating expected values.

the *expected value* of f on $\mathbb{I}$ as:

$$\mathbb{E}_f[\mathbb{I}] \triangleq \sum_{i:I} f(d(i)) \cdot v(i)$$

(this coincides with the usual notion of expected value of a random variable if we interpret the indexed valuation as a distribution using the map H defined above). Because I is a countable type, the above series may not necessarily converge[5]. We say that the expected value of f on $\mathbb{I}$ exists if the above series converges absolutely. Throughout this dissertation, when expected values are mentioned in rules and derivations, we will implicitly assume side conditions stating that all the relevant expected values exist. A selection of rules for calculating expected values are shown in Figure 2.3.

Given a predicate $P : T \to \mathsf{Prop}$, we define $[P]$ as the indicator function

$$[P](x) = \begin{cases} 1 & \text{if } P(x) \text{ is true} \\ 0 & \text{if } P(x) \text{ is not true} \end{cases}$$

Then $\mathbb{E}_{[P]}[\mathbb{I}]$ is equal to the probability that P holds of the value returned by $\mathbb{I}$, so we define:

$$\Pr_P[\mathbb{I}] \triangleq \mathbb{E}_{[P]}[\mathbb{I}]$$

Because an $\mathcal{I}$ of type $M_{\mathsf{NI}}(T)$ is just a non-empty set of indexed valuations, we can apply $\mathbb{E}_f[-]$ to each $\mathbb{I} \in \mathcal{I}$ to get the set of expected values that can arise depending on how non-deterministic choices are resolved. Generally speaking, we will be interested in bounding the smallest or largest possible value that these expected values can take. We can define the *minimal* and *maximal* expected value of f on $\mathcal{I}$ as:

$$\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}] \triangleq \inf_{\mathbb{I} \in \mathcal{I}} \mathbb{E}_f[\mathbb{I}] \qquad \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}] \triangleq \sup_{\mathbb{I} \in \mathcal{I}} \mathbb{E}_f[\mathbb{I}]$$

[5]In the Coq formalization, I use the Coquelicot library developed by Boldo et al. [23] to reason about such infinite series.

$$\mathbb{E}_f^{\mathsf{min}}[\mathsf{ret}\ v] = f(v) \qquad \textsc{Extrema-Linear}\ \frac{k \geq 0}{\mathbb{E}^{\mathsf{min}}_{(\lambda x.k\cdot f(x)+c)}[\mathcal{I}] = k \cdot \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}] + c}$$

$$\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_1 \oplus_p \mathcal{I}_2] = p \cdot \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_1] + (1-p) \cdot \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_2] \qquad \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_1 \cup \mathcal{I}_2] = \min(\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_1], \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_2])$$

$$\textsc{Extrema-Comp}\ \mathbb{E}^{\mathsf{min}}_{g\circ f}[\mathcal{I}] = \mathbb{E}^{\mathsf{min}}_{g}[x \leftarrow \mathcal{I}\,;\ \mathsf{ret}\ f(x)] \qquad \textsc{Extrema-Bind-Case}\ \frac{\forall x.\, k_1 \leq \mathbb{E}_f^{\mathsf{min}}[F(x)] \leq k_2}{k_1 \leq \mathbb{E}_f^{\mathsf{min}}[x \leftarrow \mathcal{I}_1\,;\ F(x)] \leq k_2} \qquad \textsc{Extrema-Mono}\ \frac{\forall x.\, f_1(x) \leq f_2(x)}{\mathbb{E}^{\mathsf{min}}_{f_1}[\mathcal{I}] \leq \mathbb{E}^{\mathsf{min}}_{f_2}[\mathcal{I}]}$$

Figure 2.4: Selection of rules for calculating extrema of expected values (analogous rules for $\mathbb{E}_f^{\mathsf{max}}[-]$ omitted).

We say that these extrema exist if for all $\mathbb{I} \in \mathcal{I}$, $\mathbb{E}_f[\mathbb{I}]$ exists. Because $\mathcal{I}$ may be an infinite set, $\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}]$ and $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}]$ can be $-\infty$ and $+\infty$ respectively. $\Pr_P^{\mathsf{max}}[\mathcal{I}]$ and $\Pr_P^{\mathsf{min}}[\mathcal{I}]$ are defined in the analogous way.

Rules for calculating these values are given in Figure 2.4. These rules are derived by using the corresponding rules for expected values from Figure 2.3. As before, we implicitly assume that all of the stated extrema exist and are finite.

To help reason about these extrema, we introduce a partial order on terms of type $M_{\mathsf{NI}}(T)$: We say $\mathcal{I}_1 \subseteq \mathcal{I}_2$ if for each $\mathbb{I}_1 \in \mathcal{I}_1$, there exists some $\mathbb{I}_2 \in \mathcal{I}_2$ such that $\mathbb{I}_1 \equiv \mathbb{I}_2$. If $\mathcal{I}_1 \subseteq \mathcal{I}_2$ then $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}_1] \leq \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}_2]$ and $\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_2] \leq \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}_1]$. Thus, we can bound $\mathcal{I}_1$'s extrema by first finding some $\mathcal{I}_2$ such that $\mathcal{I}_1 \subseteq \mathcal{I}_2$, and then bounding the latter's extrema.

Although we have omitted side-conditions on the existence of expected values and extrema here, they must be dealt with in formal proofs. One way to discharge these side conditions is to show that the functions we are computing expected values of are suitably bounded. We first define the *support* of $\mathbb{I}$ as the set of all values that occur with non-zero probability:

$$\mathsf{supp}(\mathbb{I}) \triangleq \{v \mid \exists i \in \mathbb{I}.\, d(i) = v \wedge v(i) > 0\}$$

The support of a set of indexed valuations, $\mathcal{I}$ is then the union of their supports:

$$\mathsf{supp}(\mathcal{I}) \triangleq \bigcup_{\mathbb{I}\in\mathcal{I}} \mathsf{supp}(\mathbb{I})$$

We say that f is bounded on the support of $\mathcal{I}$ if there exists some c such that $|f(v)| \leq c$ for all $c \in \mathsf{supp}(\mathcal{I})$. If this holds, then $\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}]$ and $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}]$ exist and are finite.

Example 2.3 (Expected value of approximate counter.)**.** Using the above rules, we can show that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ n\ 0] = \mathbb{E}^{\mathsf{max}}_{\mathsf{id}}[\mathsf{approxN}\ n\ 0] = n$, which implies that no matter how the non-determinism is resolved in our model of the counter, the expected value of the result will be the number of increments. Let us just consider the case for the minimum, because the maximum is the same. The proof proceeds by induction on n, after first strengthening the induction hypothesis to the claim that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ n\ l] = n + l$. The key step of the proof is to show that

$$\mathbb{I} \equiv_{\mathsf{p}} \mathbb{I} \qquad \frac{\mathbb{I}_1 \equiv \mathbb{I}_1' \qquad \mathbb{I}_2 \equiv \mathbb{I}_2' \qquad \mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_2}{\mathbb{I}_1' \equiv_{\mathsf{p}} \mathbb{I}_2'} \qquad \frac{\mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_2 \qquad \mathbb{I}_2 \equiv_{\mathsf{p}} \mathbb{I}_3}{\mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_3}$$

$$\frac{\mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_2 \qquad \forall x.\, F_1(x) \equiv_{\mathsf{p}} F_2(x)}{x \leftarrow \mathbb{I}_1 ;\ F_1(x) \equiv_{\mathsf{p}} x \leftarrow \mathbb{I}_2 ;\ F_2(x)} \qquad \frac{\mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_2 \qquad \mathbb{I}_1' \equiv_{\mathsf{p}} \mathbb{I}_2'}{\mathbb{I}_1 \oplus_p \mathbb{I}_1' \equiv_{\mathsf{p}} \mathbb{I}_2 \oplus_p \mathbb{I}_2'} \qquad (x \leftarrow \mathbb{I}_1 ;\ \mathbb{I}_2) \equiv_{\mathsf{p}} \mathbb{I}_2$$

Figure 2.5: Rules for the $\equiv_{\mathsf{p}}$ relation on indexed valuations.

$\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxIncr}] = 1$, *i.e.,* each increment contributes 1 to the expected value. By Extrema-Bind-Case, it suffices to show that whatever value of k is non-deterministically selected, the resulting expected value will be 1. We have that for all k:

$$\begin{aligned}
&\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}\left[\mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0\right] \\
&= \left(\frac{1}{k+1}\right) \cdot (k+1) + \left(1 - \frac{1}{k+1}\right) \cdot 0 \\
&= 1
\end{aligned}$$

Let us summarize the discussion so far. Because the non-determinism monad M_{N} could not be combined with the standard probabilistic choice monad M_{P}, we replaced the latter with the monad of indexed valuations, M_{I}. The distinction between M_{I} and M_{P} is that indexed valuations carry additional data (the indices) and a finer notion of equivalence. This additional data was used to define the bind operation in the combined monad M_{NI}, in which we had functions h that depend on the indices themselves, rather than just what they decode to.

We then re-developed the notions of expected values and probabilities for M_{I} and defined corresponding extrema of expected values for M_{NI}. Since these definitions respect the equivalence relations on M_{I} and M_{NI}, we could bound the extrema of some $\mathcal{I}$ by first finding $\mathcal{I}'$ such that $\mathcal{I} \equiv \mathcal{I}'$, and then bounding the extrema of $\mathcal{I}'$. More generally, we also had the $\subseteq$ relation, so that similar bounds could be obtained solely by showing $\mathcal{I} \subseteq \mathcal{I}'$.

However, the relations $\equiv$ and $\subseteq$ above are finer than necessary if our goal is to use them to translate bounds on extrema of $\mathcal{I}'$ to bounds on $\mathcal{I}$, and similarly so for translating bounds on expected values of one indexed valuation to another. For example, if f is a bounded function, then $\mathbb{E}_f[\mathbb{I}] = \mathbb{E}_f[\mathbb{I} \oplus_p \mathbb{I}]$, yet we know that $\mathbb{I} \not\equiv \mathbb{I} \oplus_p \mathbb{I}$.

Because our goal is to do relational reasoning in order to bound expected values, it is useful to define the coarsest relations that suffice for this purpose, and derive some properties about them. We define $\mathbb{I} \equiv_{\mathsf{p}} \mathbb{I}'$ to hold if for all bounded[6] functions f, $\mathbb{E}_f[\mathbb{I}] = \mathbb{E}_f[\mathbb{I}']$. Because indicator functions are bounded, observe that if $\mathbb{I} \equiv_{\mathsf{p}} \mathbb{I}'$, then $\Pr_A[\mathbb{I}] = \Pr_A[\mathbb{I}']$ for all A. In other words, this notion of equivalence is the same as saying that the probability distributions $H(\mathbb{I})$ and $H(\mathbb{I}')$ are equal. Rules for this relation are shown in Figure 2.5. Crucially, it is a congruence with respect to all the monad operations.

[6]The reason for quantifying over bounded functions is to ensure that the two expected values exist.

$$\mathcal{I} \subseteq_{\mathsf{p}} \mathcal{I} \qquad \frac{\mathcal{I}_1 \subseteq_{\mathsf{p}} \mathcal{I}_2 \qquad \mathcal{I}_2 \subseteq_{\mathsf{p}} \mathcal{I}_3}{\mathcal{I}1 \subseteq_{\mathsf{p}} \mathcal{I}_3} \qquad \frac{\mathcal{I}_1' \subseteq \mathcal{I}_1 \qquad \mathcal{I}_2 \subseteq \mathcal{I}_2' \qquad \mathcal{I}_1 \subseteq_{\mathsf{p}} \mathcal{I}_2}{\mathcal{I}_1' \subseteq_{\mathsf{p}} \mathcal{I}_2'}$$

$$\frac{\mathcal{I}_1 \subseteq_{\mathsf{p}} \mathcal{I}_2 \qquad \forall x.\, F_1(x) \subseteq_{\mathsf{p}} F_2(x)}{x \leftarrow \mathcal{I}_1\,;\ F_1(x) \subseteq_{\mathsf{p}} x \leftarrow \mathcal{I}_2\,;\ F_2(x)} \qquad \frac{\mathcal{I}_1 \subseteq_{\mathsf{p}} \mathcal{I}_2 \qquad \mathcal{I}_1' \subseteq_{\mathsf{p}} \mathcal{I}_2'}{\mathcal{I}_1 \oplus_p \mathcal{I}_1' \equiv_{\mathsf{p}} \mathcal{I}_2 \oplus_p \mathcal{I}_2'} \qquad (x \leftarrow \mathcal{I}_1\,;\ \mathcal{I}_2) \subseteq_{\mathsf{p}} \mathcal{I}_2$$

Figure 2.6: Rules for $\subseteq_{\mathsf{p}}$ relation.

Analogously, we define $\mathcal{I} \subseteq_{\mathsf{p}} \mathcal{I}'$ to hold if for all bounded functions f, $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}] \leq \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}']$. Thus, if this relation holds, one can bound the maxima of $\mathcal{I}$ by bounding the maxima of $\mathcal{I}'$. Since the negation of f is bounded if and only if f is, this also implies that $\mathbb{E}_f^{\mathsf{min}}[\mathcal{I}'] \leq \mathbb{E}_f^{\mathsf{min}}[\mathcal{I}]$ for all bounded f. Some rules for this relation are shown in Figure 2.6.

The following lemma shows that the definition of $\subseteq_{\mathsf{p}}$ generalizes to a larger class of functions than just the bounded ones:

Lemma 2.4. If $\mathcal{I} \subseteq_{\mathsf{p}} \mathcal{I}'$ and f is bounded on the support of $\mathcal{I}'$, then f is bounded on the support of $\mathcal{I}$ and $\mathbb{E}_f^{\mathsf{max}}[\mathcal{I}] \leq \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}']$.

Proof. If $x \in \mathsf{supp}(\mathcal{I})$, then there exists some $\mathbb{I} \in \mathcal{I}$ such that $\Pr_{\lambda y.\, y=x}[\mathbb{I}] > 0$. Hence, $\Pr^{\mathsf{max}}_{\lambda y.\, y=x}[\mathcal{I}] > 0$, and therefore $\Pr^{\mathsf{max}}_{\lambda y.\, y=x}[\mathcal{I}'] > 0$. Thus, there exists some $\mathbb{I}' \in \mathcal{I}'$ such that $\Pr_{\lambda y.\, y=x}[\mathbb{I}'] > 0$, so $x \in \mathsf{supp}(\mathcal{I}')$. This means $\mathsf{supp}(\mathcal{I}) \subseteq \mathsf{supp}(\mathcal{I}')$ so f is bounded on the support of $\mathcal{I}$.

We then consider the function

$$g(x) = \begin{cases} f(x) & \text{if } x \in \mathsf{supp}(\mathcal{I}') \\ 0 & \text{otherwise} \end{cases}$$

Then $\mathbb{E}_g^{\mathsf{max}}[\mathcal{I}] = \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}]$ and $\mathbb{E}_g^{\mathsf{max}}[\mathcal{I}'] = \mathbb{E}_f^{\mathsf{max}}[\mathcal{I}]$, since g only differs from f outside the support of $\mathcal{I}$ and $\mathcal{I}'$. Moreover, g is bounded, so we have $\mathbb{E}_g^{\mathsf{max}}[\mathcal{I}] \leq \mathbb{E}_g^{\mathsf{max}}[\mathcal{I}']$ from the definition of $\subseteq_{\mathsf{p}}$. □

2.4 Analogues of Classical Inequalities

An important part of probability theory is the extensive number of inequalities that can be used to bound probabilities and expected values. These inequalities are frequently used in the analysis of algorithms. How do these generalize to the extrema of expected values of the form we have described in the previous section?

2.4.1 Markov's Inequality

One of the most fundamental inequalities in probability theory is Markov's inequality, which says that if X is a non-negative random variable such that $\mathbb{E}[X]$ exists, then for all $a > 0$

$$\Pr[X > a] \leq \frac{\mathbb{E}[X]}{a}$$

More generally, if X is an arbitrary random variable and $h : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ is a monotone function for which $h(a) > 0$ and $\mathbb{E}[h(|X|)]$ exists, then:

$$\Pr[X > a] \leq \frac{\mathbb{E}[h(|X|)]}{h(a)}$$

The standard proof of this result generalizes to the following formulation for $\mathbb{E}^{\mathsf{max}}$:

Lemma 2.5. Let $\mathcal{I}$ be a non-empty set of indexed valuations of type T, and let f be a real-valued function on T. If $h : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ is monotone, $h(a) > 0$, and $\mathbb{E}^{\mathsf{max}}_{\lambda x.\, h(|f(x)|)}[\mathcal{I}]$ exists and is finite, then

$$\Pr^{\mathsf{max}}_{\lambda x.\, f(x) > a}[\mathcal{I}] \leq \frac{\mathbb{E}^{\mathsf{max}}_{\lambda x.\, h(|f(x)|)}[\mathcal{I}]}{h(a)}$$

Proof. Unfolding definitions, we have $\Pr^{\mathsf{max}}_{\lambda x.\, f(x) > a}[\mathcal{I}] = \mathbb{E}^{\mathsf{max}}_{[\lambda x.\, f(x) > a]}[\mathbb{I}]$. Moreover, using the analogue of Extrema-Linear for maxima, we can simplify the right hand side of the inequality:

$$\frac{\mathbb{E}^{\mathsf{max}}_{\lambda x.\, h(|f(x)|)}[\mathcal{I}]}{h(a)} = \mathbb{E}^{\mathsf{max}}_{\lambda x.\, h(|f(x)|)/h(a)}[\mathcal{I}]$$

It suffices then by Extrema-Mono to show that for all x,

$$[f(x) > a] \leq h(|f(x)|)/h(a)$$

First, consider the case where $f(x) \leq a$. Then the left hand side is 0, and since h is non-negative, the inequality is immediate. If $f(x) > a$, then the left hand side is 1, and we just need to show that $h(a) \leq h(|f(x)|)$, which follows from the fact that h is monotone.

□

2.4.2 Chebyshev's Inequality

An important instance of the generalized form of Markov's inequality is known as Chebyshev's inequality. In the standard setting this says that if X is a random variable whose expected value and variance exist, then for all $k > 0$,

$$\Pr[|X - \mathbb{E}[X]| > k] \leq \frac{\mathbb{V}[X]}{k^2}$$

Recall that $\mathbb{V}[X]$, the variance of a random variable, is defined to be $\mathbb{E}[(X - \mathbb{E}[X])^2]$. Generalizing slightly, we have bounds for deviations around not just the expected value, but any constant c:

$$\Pr[|X - c| > k] \leq \frac{\mathbb{E}[(X-c)^2]}{k^2}$$

This version has the following analogue for $\mathbb{E}^{\mathsf{max}}$:

Theorem 2.6. Let $\mathcal{I}$ be a non-empty set of indexed valuations of type T, and let f be a real-valued function on T. If $\mathbb{E}^{\mathsf{max}}_{\lambda x.\,(f(x)-c)^2}[\mathcal{I}]$ exists and is finite, then

$$\Pr^{\mathsf{max}}_{\lambda x.\,|f(x)-c|>k}[\mathcal{I}] \leq \frac{\mathbb{E}^{\mathsf{max}}_{\lambda x.\,(f(x)-c)^2}[\mathcal{I}]}{k^2}$$

Directly computing $\mathbb{E}^{\mathsf{max}}_{\lambda x.\,(f(x)-c)^2}[\mathcal{I}]$ to use in this inequality can be difficult. However, we can obtain a bound using linearity of expectation:

Lemma 2.7. If $c \geq 0$, then

$$\mathbb{E}^{\mathsf{max}}_{\lambda x.\,(f(x)-c)^2}[\mathcal{I}] \leq \mathbb{E}^{\mathsf{max}}_{\lambda x.\,f(x)^2}[\mathcal{I}] - 2c\mathbb{E}^{\mathsf{min}}_{f}[\mathcal{I}] + c^2$$

Proof. For each $\mathbb{I} \in \mathcal{I}$, we have:

$$\mathbb{E}_{\lambda x.\,(f(x)-c)^2}[\mathbb{I}] = \mathbb{E}_{\lambda x.\,f(x)^2}[\mathbb{I}] - 2c\mathbb{E}_{f}[\mathbb{I}] + c^2$$

Because $\mathbb{E}_{\lambda x.\,f(x)^2}[\mathbb{I}] \leq \mathbb{E}^{\mathsf{max}}_{\lambda x.\,f(x)^2}[\mathcal{I}]$ and $\mathbb{E}^{\mathsf{min}}_{f}[\mathcal{I}] \leq \mathbb{E}_{f}[\mathbb{I}]$, it follows that

$$\mathbb{E}_{\lambda x.\,(f(x)-c)^2}[\mathbb{I}] \leq \mathbb{E}^{\mathsf{max}}_{\lambda x.\,f(x)^2}[\mathcal{I}] - 2c\mathbb{E}^{\mathsf{min}}_{f}[\mathcal{I}] + c^2$$

□

2.5 Couplings

As mentioned in Chapter 1, recent work by Barthe et al. [11, 14, 16] has shown that the notion of *coupling* [80] is fundamental for relational reasoning in probabilistic program logics. Adapting the definition from Chapter 1 to the monad M_{P}, we have that given two distributions $A : M_{\mathsf{P}}(T_A)$ and $B : M_{\mathsf{P}}(T_B)$, a coupling between A and B is a distribution $C : M_{\mathsf{P}}(T_A \times T_B)$ such that:

1. $\forall x : T_A.\, A(x) = \sum_y C(x, y)$
2. $\forall y : T_B.\, B(y) = \sum_x C(x, y)$

That is, C is a joint distribution whose marginals equal A and B. These two conditions are equivalent to requiring that:

1. $A \equiv ((x, y) \leftarrow C\,;\ \mathsf{ret}\ x)$
2. $B \equiv ((x, y) \leftarrow C\,;\ \mathsf{ret}\ y)$

Given a predicate $P : A \times B \to \mathsf{Prop}$, we say that C is a P-coupling, if, in addition to the above, we have:

$$\forall x, y.\, C(x, y) > 0 \to P(x, y)$$

i.e., all pairs (x, y) in the support of the distribution C satisfy P. The existence of a P-coupling can tell us important things about the two distributions. For example, if $P(x, y) = (x = y)$, then the existence of a P-coupling tells us the two distributions are equivalent. Moreover, there are rules for systematically constructing couplings between distributions. We will explain some of these rules once we have described how to adapt couplings to the monad M_{NI}.

Using the monadic formulation of the coupling conditions, it is straightforward to define an analogous idea for M_{I}. Given $\mathbb{I}_1 : M_{\mathsf{I}}(T_1)$ and $\mathbb{I}_2 : M_{\mathsf{I}}(T_2)$, a coupling between $\mathbb{I}_1$ and $\mathbb{I}_2$ is an $\mathbb{I} : M_{\mathsf{I}}(T_1 \times T_2)$ such that:

1. $\mathbb{I}_1 \equiv_{\mathsf{p}} ((x, y) \leftarrow \mathbb{I};\ \mathsf{ret}\ x)$

2. $\mathbb{I}_2 \equiv_{\mathsf{p}} ((x, y) \leftarrow \mathbb{I};\ \mathsf{ret}\ y)$

where we use the coarser equivalence $\equiv_{\mathsf{p}}$ instead of $\equiv$ because $\equiv_{\mathsf{p}}$ corresponds to equivalence of the indexed valuations interpreted as probability distributions. Further, we say $\mathbb{I} = (I, d, v)$ is a P-coupling if for all i such that $v(i) > 0$, $P(d(i))$ holds. As before, if P is the equality predicate, then the existence of a P-coupling between $\mathbb{I}_1$ and $\mathbb{I}_2$ implies $\mathbb{I}_1 \equiv_{\mathsf{p}} \mathbb{I}_2$.

We can lift this to a relation between a single indexed valuation $\mathbb{I}$ and a set of indexed valuations $\mathcal{I}$: We say[7] there is a *non-deterministic P-coupling* between $\mathbb{I}$ and $\mathcal{I}$ if there exists some $\mathbb{I}'$ such that $\{\mathbb{I}'\} \subseteq_{\mathsf{p}} \mathcal{I}$ and a P-coupling between $\mathbb{I}$ and $\mathbb{I}'$. We write $\mathbb{I} \sim \mathcal{I} : P$ to denote the existence of such a coupling. The word non-deterministic will be omitted when it is clear from context which kind of coupling we mean.

Rules for constructing these couplings are shown in Figure 2.7. If we interpret the P in $\mathbb{I} \sim \mathcal{I} : P$ as a kind of "post-condition" for the execution of the computations $\mathbb{I}$ and $\mathcal{I}$, then these coupling rules have the structure of a Hoare-like relational logic [21], as in the work of Barthe et al. [11]: *e.g.,* the rule Bind is analogous to the usual sequencing rule in Hoare logic.

The rule P-Choice lets us couple probabilistic choices $\mathbb{I} \oplus_p \mathbb{I}'$ and $\mathcal{I} \oplus_p \mathcal{I}'$ with post-condition P by coupling $\mathbb{I}$ to $\mathcal{I}$ and $\mathbb{I}'$ to $\mathcal{I}'$. This is somewhat surprising: we get to reason about these two probabilistic choices as if they both chose the left alternative or both chose the right alternative, rather than considering the full set of four combinations. This counter-intuitive rule is quite useful, as demonstrated in the many examples given in the work of Barthe et al..

The following theorem lets us use the existence of a non-deterministic coupling to bound expected values:

Theorem 2.8. Let g be bounded on $\mathsf{supp}(\mathcal{I})$ and let $P(x, y) = (f(x) = g(y))$. If $\mathbb{I} \sim \mathcal{I} : P$, then $\mathbb{E}_f[\mathbb{I}]$ exists and

$$\mathbb{E}_g^{\mathsf{min}}[\mathcal{I}] \leq \mathbb{E}_f[\mathbb{I}] \leq \mathbb{E}_g^{\mathsf{max}}[\mathcal{I}]$$

[7] Barthe et al. [11] use "non-deterministic coupling" to refer to a particular kind of coupling which is unrelated to adversarial non-deterministic choice.

$$\textsc{Ret}\ \frac{P(a,b)}{\mathsf{ret}\ a \sim \mathsf{ret}\ b : P} \qquad \textsc{Equiv}\ \frac{\mathbb{I} \equiv \mathbb{I}' \qquad \mathcal{I} \subseteq \mathcal{I}' \qquad \mathbb{I} \sim \mathcal{I} : P}{\mathbb{I}' \sim \mathcal{I}' : P}$$

$$\textsc{Conseq}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \to P'(x,y)}{\mathbb{I} \sim \mathcal{I} : P'}$$

$$\textsc{Bind}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \forall x, y.\, P(x,y) \to F(x) \sim F'(y) : Q}{(x \leftarrow \mathbb{I};\ F(x)) \sim (y \leftarrow \mathcal{I};\ F'(y)) : Q} \qquad \textsc{P-Choice}\ \frac{\mathbb{I} \sim \mathcal{I} : P \qquad \mathbb{I}' \sim \mathcal{I}' : P}{\mathbb{I} \oplus_p \mathbb{I}' \sim \mathcal{I} \oplus_p \mathcal{I}' : P}$$

$$\textsc{Trivial}\ \mathbb{I} \sim \mathcal{I} : \mathsf{True}$$

Figure 2.7: Rules for constructing non-deterministic couplings.

Proof. We will just show that $\mathbb{E}_g^{\mathsf{min}}[\mathcal{I}] \leq \mathbb{E}_f[\mathbb{I}]$, as the case for the upper bound is similar. Using Ex-Comp and Extrema-Comp it suffices to show that

$$\mathbb{E}_{\mathsf{id}}^{\mathsf{min}}[y \leftarrow \mathcal{I};\ \mathsf{ret}\ g(y)] \leq \mathbb{E}_{\mathsf{id}}[x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)]$$

and to establish that the expected value on the right exists. From Bind we have that

$$\big(x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)\big) \sim \big(y \leftarrow \mathcal{I};\ \mathsf{ret}\ g(y)\big) : \big(\lambda(x,y).\, x = y\big)$$

Thus there exists some $\mathbb{I}'$ such that $\{\mathbb{I}'\} \subseteq_{\mathsf{p}} y \leftarrow \mathcal{I};\ \mathsf{ret}\ g(y)$ and a $(\lambda(x,y).x = y)$-coupling between $x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)$ and $\mathbb{I}'$. Hence, $x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x) \equiv_{\mathsf{p}} \mathbb{I}'$. Because g is bounded on $\mathsf{supp}(\mathcal{I})$, we have that $\mathbb{E}_{\mathsf{id}}[\mathbb{I}']$ exists. This means that $\mathbb{E}_{\mathsf{id}}[x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)]$ exists as well, and moreover, we have:

$$\begin{aligned}\mathbb{E}_{\mathsf{id}}[x \leftarrow \mathbb{I};\ \mathsf{ret}\ f(x)] &= \mathbb{E}_{\mathsf{id}}[\mathbb{I}'] \\ &\geq \mathbb{E}_{\mathsf{id}}^{\mathsf{min}}[y \leftarrow \mathcal{I};\ \mathsf{ret}\ g(y)]\end{aligned}$$

where the last inequality follows from the fact that $\{\mathbb{I}'\} \subseteq_{\mathsf{p}} (y \leftarrow \mathcal{I};\ \mathsf{ret}\ g(y))$. □

2.6 Alternatives

A number of denotational models combining probabilistic and non-deterministic choice have been developed [50, 85, 119, 127]. Many of these are presented as monads on certain categories of domains, but one can instead consider analogues on the category SET, as done by Varacca and Winskel. To make the comparison with indexed valuations clearer, I will do so in the descriptions below.

In an alternative developed by Mislove [85] and Tix et al. [119], the law $A \oplus_p A \equiv A$ holds again, but the role of non-empty sets for modeling non-determinism is instead fulfilled by *convex* sets. Such sets are closed under convex combinations, in the following sense: if A and B belong to the set, then $A \oplus_p B$ does as well, for all p. Operationally, we can think of this as saying that whenever the adversarial scheduler can pick between two alternatives A and B, it also has the power to flip a (weighted) coin and use the outcome to select between the alternatives. Indeed, Varacca and Winskel use this monad to give an adequate denotational semantics for a language with such a probabilistic scheduler, and use the monad of finite sets of indexed valuations to give an adequate semantics for a variant in which the scheduler cannot make probabilistic choices. Gibbons and Hinze [46] also used this monad based on convex powersets to model and reason about several variants of the classic Monty Hall problem. It would be interesting to consider using this alternative monad. I suspect that natural analogues of many of the results presented in this chapter could be obtained.

Chapter 3

Iris: A Brief Tutorial

As explained in Chapter 1, Polaris is an extension of Iris, a recent concurrency logic with many expressive features. In order to explain these extensions, some background must first be given on Iris. In this chapter and the following one, I explain the aspects of Iris needed to understand Polaris and the examples verified using it in Chapter 6. Rather than being tied to a particular programming language, Iris provides a more general framework that can be instantiated to obtain logics for different languages. The present chapter gives a brief introduction to how Iris is used by describing an instantiation of Iris for a concurrent ML-like language. Chapter 4 describes the more general framework and explains the adequacy proof for Iris.

This chapter summarizes material from the Iris papers and manual [64, 65, 66, 72, 116], and interested readers are referred to these original sources for a full account. The lecture notes by Birkedal and Bizjak [22] provide a more thorough introduction to Iris. Readers familiar with Iris via some of these sources can skip this chapter.

3.1 Concurrent ML-like Language

The syntax and per-thread semantics of the example language are given in Figure 3.1. We use syntactic evaluation contexts to structure the operational semantics. The values of the language are recursive functions, pairs and sums of values, unit $()$, integers (for which we use the meta-variable z), booleans (with metavariable b), and addresses in the heap (meta-variable l), which are represented concretely as positive natural numbers. We write $\lambda x.\, e$ as notation for a recursive function $\mathsf{rec}\ f\ x.\ e$ when f does not occur free in e. Similarly, $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is notation for the expression $(\lambda x.\, e_2)\ e_1$. There are (partial) coercions $\mathrm{expr\_to\_val}$ and $\mathrm{val\_to\_expr}$ between expressions and values of the language. We specify head step reductions using the judgment $e, \sigma \to_{\mathsf{h}} e', \sigma', T$, which means that a thread executing an expression whose head is e in a state σ can take a step to e', updating the state to σ' and creating new threads for each expression in the list T (which may be empty). These are then lifted to the per-thread step relation $\to$ using the rule Ctx-Step in the standard way.

The language has several special concurrent operations. The expression $\mathsf{fork}\{e\}$ forks a new thread running e and returns $()$ in the parent thread. The compare and swap expression $\mathsf{CAS}(l, v_1, v_2)$ checks whether the value stored in address l is equal to v_1: if so, it replaces it

with v_2 and returns true; otherwise it leaves l unchanged and returns false. The fetch-and-add command $\mathsf{FAA}(l, z)$ adds z to the integer stored at l (if l is not allocated or does not contain an integer, the command gets stuck), writes the summed value back to l, and returns the value that was originally stored at l.

We define:

$$\mathrm{red}(e, \sigma) \triangleq \exists e', \sigma', T.\, (e; \sigma \to e', \sigma', T)$$

and say that e is reducible in σ if this holds. An expression is said to be *atomic* if whenever it can take a step, the resulting expression is not reducible:

$$\mathrm{atomic}(e) \triangleq \forall \sigma, e', \sigma', T.\, (e; \sigma \to e'; \sigma'; T) \Rightarrow \neg \mathrm{red}(e', \sigma')$$

The single-thread semantics is then lifted into a concurrent semantics. A *configuration* is a non-empty list of threads (called the "thread pool") and a state. Again, using T as a meta-variable for a list of threads, we write $+\!\!+$ for the operation of appending two lists together and $[e]$ for the singleton list consisting of e. The following reduction relation specifies how concurrent steps are taken:

$$\frac{e; \sigma \to e'; \sigma'; T_f}{T_1 +\!\!+ [e] +\!\!+ T_2; \sigma \to T_1 +\!\!+ [e'] +\!\!+ T_2 +\!\!+ T_f; \sigma'}$$

In the above, some thread in the pool takes a step according to the per-thread reduction relation, generating some list of new threads T_f which are added to the end of the thread pool.

3.2 Resource Algebras

In addition to being parameterized by a language, Iris is also parameterized by a type of *resources*. Recall from Chapter 1 that in separation logic, one thinks of propositions as asserting ownership of resources. These resources represent both the "real" state of the program one is verifying, as well as auxiliary "ghost state" that is used to track additional information needed for verification.

In Iris, these resources[1] are terms from structures called resource algebras:

Definition 3.1. A *resource algebra* (RA) is a type M with an operation $\cdot : M \times M \to M$, a predicate $\mathcal{V} : M \to \mathsf{Prop}$, and a function $|{-}| : M \to \mathsf{Option}\ M$ such that for all $a, b, c : M$,

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$
$$a \cdot b = b \cdot a$$
$$\mathcal{V}(a \cdot b) \Rightarrow \mathcal{V}(a)$$

[1] Iris in fact supports a more general algebraic structure for resources called a "camera". However, we will not need to use this more general structure directly in our examples. Thus, we will restrict attention to the more limited notion of resource algebras above. Jung et al. [66] make a similar simplification when initially presenting Iris.

Syntax:

Val	v	::=	$\mathsf{rec}\, f\, x.\, e \mid (v_1, v_2) \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid () \mid z \mid b \mid l \mid \ldots$
Expr	e	::=	$e_1\, e_2 \mid \mathsf{ref}\, e \mid !e \mid e_1 \coloneqq e_2 \mid \mathsf{fork}\{e\} \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \mathsf{FAA}(e_1, e_2) \mid \ldots$
Ctx	K	::=	$[\,] \mid K\, e \mid v\, K \mid (K, e) \mid (v, K) \mid !K \mid K \coloneqq e \mid v \coloneqq K \mid \ldots$
Loc	l	:	$\mathbb{N}^+$
State	σ	:	$\mathsf{Loc} \to \mathsf{Val}$
ThreadPool	T	:	$\mathsf{List}\ \mathsf{Expr}$
Config	ρ	:	$\{T : \mathsf{ThreadPool} \mid T \neq \emptyset\} \times \mathsf{State}$

Head Reduction: $e; \sigma \to_{\mathsf{h}} e'; \sigma'; T$

$$\frac{l = \min(\mathrm{dom}(\sigma)) + 1}{\mathsf{ref}\, v; \sigma \to_{\mathsf{h}} l; \sigma[l := v]; \mathsf{nil}} \qquad \frac{l \in \mathrm{dom}(\sigma)}{!l; \sigma \to_{\mathsf{h}} \sigma(l); \sigma; \mathsf{nil}} \qquad \frac{l \in \mathrm{dom}(\sigma)}{l \coloneqq v; \sigma \to_{\mathsf{h}} (); \sigma[l := v]; \mathsf{nil}}$$

$$\mathsf{fork}\{e\}; \sigma \to_{\mathsf{h}} (); \sigma; e \qquad \frac{l \in \mathrm{dom}(\sigma) \qquad \sigma(l) \neq v_1}{\mathsf{CAS}(l, v_1, v_2); \sigma \to_{\mathsf{h}} \mathsf{false}; \sigma; \mathsf{nil}}$$

$$\frac{\sigma(l) = v_1}{\mathsf{CAS}(l, v_1, v_2); \sigma \to_{\mathsf{h}} \mathsf{true}; \sigma[l := v_2]; \mathsf{nil}} \qquad \frac{\sigma(l) = z_1}{\mathsf{FAA}(l, z_2); \sigma \to_{\mathsf{h}} z_1; \sigma[l := z_1 + z_2]; \mathsf{nil}}$$

(standard rules omitted)

Per-Thread Reduction: $e; \sigma \to e'; \sigma'; T$

$$\textsc{Ctx-Step}\ \frac{e; \sigma \to_{\mathsf{h}} e'; \sigma'; T}{K[e]; \sigma \to_{\mathsf{h}} K[e']; \sigma'; T}$$

Figure 3.1: ML-like Language.

and for each a, if there exists b such that $|a| = \mathsf{Some}\ b$ then,

$$
\begin{aligned}
&a = a \cdot b \\
&|b| = \mathsf{Some}\ b \\
&a \preccurlyeq a' \Rightarrow (\exists b'.\ |a'| = \mathsf{Some}\ b' \wedge b \preccurlyeq b') \qquad \text{where } a \preccurlyeq b \triangleq \exists c : M.\ a \cdot c = b
\end{aligned}
$$

We call terms of type M *resources*, and write RA for the type of all resource algebras. Given $a, b : M$, the product $a \cdot b$ represents the composition of the two resources a and b. This composition operation is associative and commutative. The predicate $\mathcal{V}$ indicates which resources are "well formed". If $\mathcal{V}(a)$ holds we say that a is *valid*. If the composition $a \cdot b$ is valid, then a and b are also valid. When $|a| = \mathsf{Some}\ b$, we say that b is the *core* of a. If b is the core of a, then $a = a \cdot b$, so we think of the core b as being a resource which we can create unlimited copies of once we own a. Given a resource algebra M, if there exists a term $\varepsilon : M$ such that $\mathcal{V}(\varepsilon)$ holds, $\varepsilon \cdot a = a$ for all a, and $|\varepsilon| = \mathsf{Some}\ \varepsilon$, then ε is said to be a *unit* of M. If an algebra M has a unit, then for all $a : M$, $\varepsilon \preccurlyeq a$, so the rules for the core operation ensure that $|a| \neq \mathsf{None}$.

We give some examples of such resource algebras:

Example 3.2. Let T be a type. The *exclusive RA of* T, written $\mathrm{Ex}(T)$, has resources of the form $\lightning$ and $\mathsf{ex}(t)$ for all $t : M$, and operations defined by:

$$
\begin{aligned}
a \cdot b &= \lightning \\
|a| &= \mathsf{None} \\
\mathcal{V}(a) &= (a \neq \lightning)
\end{aligned}
$$

Example 3.3. Let T be a type. The *agreement RA of* T, written $\textsc{Ag}(T)$, has resources of the form $\lightning$ and $\mathsf{ag}(t)$ for all $t : M$. For all $a, b : \textsc{Ag}(T)$, we define

$$
a \cdot b = \begin{cases} a & \text{if } a = b \\ \lightning & \text{otherwise} \end{cases}
$$

We set $|\mathsf{ag}(t)| = \mathsf{Some}\ \mathsf{ag}(t)$ and $|\lightning| = \mathsf{None}$. Finally, $\mathcal{V}(a)$ is defined to hold if and only if $a \neq \lightning$.

Example 3.4. Let M be a resource algebra. The *option RA of* M, written $\textsc{Opt}(M)$, has resources of the form $\bot$ and a_{s} for all $a \in M$. Composition is defined as:

$$
\begin{aligned}
\bot \cdot \bot &= \bot \\
a_{\mathsf{s}} \cdot \bot &= a_{\mathsf{s}} \\
\bot \cdot a_{\mathsf{s}} &= a_{\mathsf{s}} \\
a_{\mathsf{s}} \cdot b_{\mathsf{s}} &= (a \cdot b)_{\mathsf{s}}
\end{aligned}
$$

Validity is defined by:

$$
\begin{aligned}
\mathcal{V}(\bot) &= \mathsf{True} \\
\mathcal{V}(a_{\mathsf{s}}) &= \mathcal{V}(a)
\end{aligned}
$$

If $|a| = \mathsf{Some}\ b$, then $|a_{\mathsf{s}}| = \mathsf{Some}\ b_{\mathsf{s}}$, otherwise $|a_{\mathsf{s}}| = \mathsf{Some}\ \bot$. Finally, $|\bot| = \mathsf{Some}\ \bot$.

Example 3.5. The resource algebra NAT has as resources the natural numbers, with composition given by addition. We define $\mathcal{V}(n) = \mathsf{True}$ and $|n| = \mathsf{Some}\ 0$ for all n.

The previous example is an instance of the following more general construction:

Example 3.6. Let M be a monoid, that is, a type equipped with a commutative, associative operation $+ : M \times M \to M$ and a term $\epsilon : M$ such that $\epsilon + a = a$ for all $a : M$. Then M can be made into a resource algebra by setting $a \cdot b = a + b$, $\mathcal{V}(n) = \mathsf{True}$, and $|n| = \mathsf{Some}\ \epsilon$.

Example 3.7. The resource algebra MAXNAT again has as resources the natural numbers, but with composition defined by $n \cdot m = \max(n, m)$. We define $\mathcal{V}(n) = \mathsf{True}$ and $|n| = \mathsf{Some}\ n$.

Notice that even though the natural numbers form a monoid under the $\max$ operation, the MAXNAT algebra is different from what we would obtain by applying the construction from Example 3.6 to this monoid, because in the latter we would have $|n| = \mathsf{Some}\ 0$ instead of $|n| = \mathsf{Some}\ n$.

Example 3.8. The *fraction RA* FRAC has terms of the form $\mathsf{frac}(q)$ for each positive rational number q, and operations defined by

$$\begin{aligned} \mathsf{frac}(q_1) \cdot \mathsf{frac}(q_2) &= \mathsf{frac}(q_1 + q_2) \\ \mathcal{V}(\mathsf{frac}(q)) &= (q \leq 1) \\ |\mathsf{frac}(q)| &= \mathsf{None} \end{aligned}$$

Example 3.9. Given two resource algebras M_1 and M_2, the *product RA* $M_1 \times M_2$ consists of pairs $(a_1, a_2) : M_1 \times M_2$. Composition is defined componentwise. Coring is defined by:

$$|(a_1, a_2)| = \begin{cases} \mathsf{Some}\ (b_1, b_2) & \text{if } |a_1| = \mathsf{Some}\ b_1 \text{ and } |a_2| = \mathsf{Some}\ b_2 \\ \mathsf{None} & \text{if } |a_1| = \mathsf{None} \text{ or } |a_2| = \mathsf{None} \end{cases}$$

A pair is valid just when both of the components are valid, that is

$$\mathcal{V}((a_1, a_2)) = \mathcal{V}(a_1) \wedge \mathcal{V}(a_2)$$

Example 3.10. Let M be a resource algebra and K a countable set. The *finite map RA*, written $\textsc{FinMap}(K, M)$, consists of partial functions of type $K \to M$ with finite domains. Given such a map f, we write $\mathrm{dom}(f)$ for its domain. The composition of two maps f_1 and f_2 has domain $\mathrm{dom}(f_1) \cup \mathrm{dom}(f_2)$, and for all $i \in \mathrm{dom}(f_1) \cup \mathrm{dom}(f_2)$, we define

$$(f_1 \cdot f_2)(i) = \begin{cases} f_1(i) \cdot f_2(i) & \text{if } i \in \mathrm{dom}(f_1) \wedge i \in \mathrm{dom}(f_2) \\ f_1(i) & \text{if } i \in \mathrm{dom}(f_1) \wedge i \notin \mathrm{dom}(f_2) \\ f_2(i) & \text{if } i \notin \mathrm{dom}(f_1) \wedge i \in \mathrm{dom}(f_2) \end{cases}$$

Validity is defined pointwise. If f is a resource and there exists $i \in \mathrm{dom}(f)$ such that $|f(i)| = \mathsf{None}$, then $|f| = \mathsf{None}$. Otherwise, if $\forall i \in \mathrm{dom}(f)$, the core of $f(i)$ exists, then $|f| = \mathsf{Some}\ g$,

where g is a partial function mapping each $i \in \mathrm{dom}(f)$ to the core of $f(i)$. We write $\{i \mapsto a\}$ for the singleton map sending i to a, and $f\,[i \mapsto a]$ for the map

$$\lambda x. \begin{cases} a & \text{if } x = i \\ f(x) & \text{if } x \neq i \end{cases}$$

which sends i to a and maps all other elements according to f.

Further examples of resource algebras will be given later on.

3.3 Basic Propositions and Semantic Entailment

As mentioned above, the Iris logic is parameterized by a family of resource algebras. The idea is that the user of Iris selects some resource algebras that they will need to model ghost state for the program they want to verify. Formally, Jung et al. [66] define a term

$$\mathsf{iProp} : \prod_{(I:\mathsf{Type})} (I \to \mathsf{RA}) \to \mathsf{Type}$$

which is the parameterized type of Iris propositions. The type I here is used to index a family of resource algebras, while the second parameter is a function mapping indices to the corresponding algebras.

The situation here is different from what the reader may be familiar with from other logics. Traditionally, one presents a logic by inductively defining the syntax of propositions and collections of rules used to prove entailments. Besides constructing derivations using these rules, one may study the meta-theory of the logic using proof-theoretic techniques, or by constructing and analyzing models.

However, in Iris, propositions and entailments are not inductively defined[2]. Rather, the type $\mathsf{iProp}\ I\ f$ is in fact the (canonical) *semantic model*. Jung et al. [66] define various connectives as functions whose codomains are the type of propositions. For example, a term

$$\mathsf{and} : \prod_{(I:\mathsf{Type})} \prod_{(f:I\to\mathsf{RA})} \mathsf{iProp}\ I\ f \to \mathsf{iProp}\ I\ f \to \mathsf{iProp}\ I\ f$$

is defined, which represents intuitionistic conjunction. Then, a relation

$$\mathsf{entailment} : \prod_{(I:\mathsf{Type})} \prod_{(f:I\to\mathsf{RA})} \mathsf{iProp}\ I\ f \to \mathsf{iProp}\ I\ f \to \mathsf{Prop}$$

is defined, which represents semantic entailment. If P and Q are two Iris propositions, we write $P \vdash Q$ as shorthand for this entailment relation[3]. This relation is shown to be reflexive

[2]On paper, Jung et al. [66] first present a syntactic system with inductive definitions, which they justify by the semantic model construction that I mention, but in the machine checked proofs only the semantic model is constructed, and one works directly with the connectives defined in this model.

[3]It is traditional in the study of logic to use $\vDash$ for this kind of a semantic entailment, however we will use $\vdash$ to stay closer to the notation used in most presentations of Iris.

Name	Type	Notation
and	iProp $\to$ iProp $\to$ iProp	$P \wedge Q$
or	iProp $\to$ iProp $\to$ iProp	$P \vee Q$
implication	iProp $\to$ iProp $\to$ iProp	$P \Rightarrow Q$
exists	$\prod_{\tau:\mathsf{Type}} (\tau \to \mathsf{iProp}) \to \mathsf{iProp}$	$\exists x : \tau.\, P$
forall	$\prod_{\tau:\mathsf{Type}} (\tau \to \mathsf{iProp}) \to \mathsf{iProp}$	$\forall x : \tau.\, P$
pure	Prop $\to$ iProp	$\ulcorner \phi : \mathsf{Prop} \urcorner$
separating conjunction	iProp $\to$ iProp $\to$ iProp	$P * Q$
separating implication	iProp $\to$ iProp $\to$ iProp	$P \mathrel{-\!\!*} Q$
later	iProp $\to$ iProp	$\triangleright P$
ownership	GName $\to \prod_{i:I} f(i) \to$ iProp $f\ I$	$\boxed{a : f(i)}^{\gamma}$
validity	$\prod_{i:I} f(i) \to$ iProp $f\ I$	$\mathcal{V}(a : f(i))$
persistently	iProp $\to$ iProp	$\Box P$
weakest precondition	Expr $\to$ Mask $\to$ (Val $\to$ iProp) $\to$ iProp	$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$
invariant	InvName $\to$ iProp $\to$ iProp	$\boxed{P}^{\iota}$
update	Mask $\to$ Mask $\to$ iProp $\to$ iProp	${}^{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\mid^{\mathcal{E}_2} [P]$

Table 3.1: Connectives defined in the Iris model.

and transitive, and various lemmas are proved about entailment for the different connectives by appealing to the semantic model. At a high level, the model is a Kripke style semantics, in which propositions represent step-indexed sets [4] of resources, and $P \vdash Q$ holds when each resource in P is also in Q. However, a user of Iris does not need to understand the underlying model and can simply use the basic lemmas about entailment proved by Jung et al. [66]. For that reason, we will not discuss the definition of the model.

While using the logic, we will fix a particular collection of resource algebras to use. It is tedious to keep writing explicitly that connectives are parameterized by such a collection, so subsequently we will assume that some collection of resource algebras has been selected, and just write iProp for the type of Iris propositions using that implicit collection, unless we need to explicitly refer to the indexing type or associated family of resource algebras. Similarly, we will just write the type of a connective like and as iProp $\to$ iProp $\to$ iProp.

Table 3.1 lists some of the connectives that Jung et al. [66] define in the semantic model of Iris. Table 3.2 describes several types mentioned in the signatures of these connectives. As we will see, some connectives can be defined directly in terms of the others.

We now describe these connectives and rules for them. Since the connectives and entailment are defined in the model, rather than being defined inductively by rules, the rules we list are simply *lemmas* about these terms that Jung et al. [66] have proven to hold in the model. The first several entries in Table 3.1 are the standard connectives of intuitionistic logic, and they

Name	Definition	Meta-variable
GName	$\mathbb{N}$	γ
InvName	$\mathbb{N}$	ι
Mask	$\mathbb{N} \to \mathsf{Prop}$	$\mathcal{E}$

Table 3.2: Additional types used in the Iris connectives. Jung et al. [66] define Mask as arbitrary subsets of natural numbers, as we do here, but in their machine checked proofs they use an alternate representation that contains finite sets, cofinite sets, and is closed under union, intersection, and complement operations.

$$P \land Q \vdash P \qquad P \land Q \vdash Q \qquad \frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \land R} \qquad P \vdash P \lor Q \qquad Q \vdash P \lor Q$$

$$\frac{P \vdash R \qquad Q \vdash R}{P \lor Q \vdash R} \qquad \frac{P \land Q \vdash R}{P \vdash Q \Rightarrow R} \qquad \frac{P \vdash Q \Rightarrow R}{P \land Q \vdash R} \qquad \frac{\forall t : \tau.\, (P \vdash [t/x]Q)}{P \vdash (\forall x : \tau.\, Q)}$$

$$\frac{t : \tau}{(\forall x : \tau.\, P) \vdash [t/x]P} \qquad \frac{\forall t : \tau.\, ([t/x]P \vdash Q)}{(\exists x : \tau.\, P) \vdash Q} \qquad \frac{t : \tau}{[t/x]P \vdash \exists x : \tau.\, P} \qquad \frac{\phi}{P \vdash \ulcorner \phi \urcorner}$$

$$\frac{\phi \to (\mathsf{True} \vdash P)}{\ulcorner \phi \urcorner \vdash P} \qquad \forall x : \tau.\, \ulcorner \phi \urcorner \vdash \ulcorner \forall x : \tau.\, \phi \urcorner$$

Figure 3.2: Rules for intuitionistic connectives.

behave as in that setting. Rules for the intuitionistic connectives are given in Figure 3.2. Note that the quantification variable x for the quantifiers $\exists$ and $\forall$ can range over an arbitrary type τ, including the type iProp itself, making the logic higher-order. The $\ulcorner \cdot \urcorner$ connective embeds meta-propositions of type Prop into Iris. We call embeddings $\ulcorner \phi \urcorner$ *pure* assertions. We can use this to define True and False as the embeddings of their meta-equivalents:

$$\mathsf{False} \triangleq \ulcorner \mathsf{False} \urcorner \qquad\qquad \mathsf{True} \triangleq \ulcorner \mathsf{True} \urcorner$$

In addition, we have the separating conjunction $*$ and separating implication $-\!\!*$ (also called "wand") of separation logic. Some motivation for the notion of separating conjunction has already been given in Chapter 1. The important point is that a resource satisfies $P * Q$ when it can be split into two pieces, one that satisfies P and one that satisfies Q. Unlike intuitionistic conjunction, in general, $P \not\vdash P * P$. Separating implication $-\!\!*$ is the right adjoint of separating conjunction (just as intuitionistic implication $\Rightarrow$ is to intuitionistic conjunction $\land$). A resource satisfies $P -\!\!* Q$ if when combined with an additional resource satisfying P, the result satisfies Q. Rules for these connectives are shown in Figure 3.3.

$$P * Q \vdash P \qquad P * Q \vdash Q \qquad \frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'} \qquad P \vdash \mathsf{True} * P \qquad P * Q \vdash Q * P$$

$$(P * Q) * R \vdash P * (Q * R) \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R} \qquad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

Figure 3.3: Rules for spatial connectives.

$$\textsc{Later-Mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{Later-Intro}\ P \vdash \triangleright P \qquad \textsc{Later-Sep}\ \triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \forall x : \tau.\ \triangleright P \vdash \triangleright \forall x : \tau.\ P$$

$$\triangleright \exists x : \tau.\ P \vdash \triangleright \mathsf{False} \vee (\exists x : \tau.\ \triangleright P) \qquad \textsc{Löb}\ (\triangleright P \Rightarrow P) \vdash P$$

Figure 3.4: Rules for later modality.

Next, we have the later modality $\triangleright$ [88]. For intuition, one can think of the logic as being stratified over "steps" of time (later, these steps will be linked to steps of program execution). Then, $\triangleright P$ holds at the present step if P holds in the next step of time. The purpose of introducing this stratification is that it allows us to take fixed points using the guarded fixed point operator $\mu\, x : \tau.\ P$, so long as occurrences of x in P appear underneath a $\triangleright$ (such occurrences are then said to be "guarded"). See Figure 3.4 for rules about these connectives. The rule Löb permits a form of inductive reasoning called Löb induction, in which to show that P holds it suffices to show P under the additional assumption that it holds one time step later (*i.e.,* $\triangleright P$). We often have occasion to repeat a modality like $\triangleright$ several times, which we will indicate by annotating the modality with a superscripted natural number. For example, $\triangleright^n$ is the modality obtained by repeating $\triangleright$ a total of n times.

Ownership of a resource a from a resource algebra M is represented by an assertion $\boxed{a : M}^{\gamma}$, where γ is a "ghost" name used to distinguish two different "instances" of the algebra M. The reason for introducing names is that the proof for two code modules may use the same kind of resource, but the resources used in each proof are distinct and not meant to interact. The algebra M must belong to the family used to instantiate the type of Iris propositions. We will simply write $\boxed{a}^{\gamma}$ when the type of a is clear. Validity of a resource a is also internalized in the logic as a proposition $\mathcal{V}(a)$. Rules for ownership and validity are shown in Figure 3.5. The rule Own-Sep indicates that ownership of the composition $a \cdot b$ is equivalent to separately owning a and b. Next, Own-Val ensures that if a resource is owned, it is valid. We can use this with Val-Elim to prove that ownership of certain resource combinations is impossible. For example, recall the exclusive resource algebra $\mathrm{Ex}(T)$ from Example 3.2. Using the rules from Figure 3.5,

Own-Sep
$\boxed{a \cdot b : M}^\gamma \dashv\vdash \boxed{a : M}^\gamma * \boxed{b : M}^\gamma$

Own-Val
$\boxed{a : M}^\gamma \vdash \mathcal{V}(a)$

Val-Comp
$\mathcal{V}(a \cdot b) \vdash \mathcal{V}(a)$

Val-Elim
$$\frac{\neg\mathcal{V}(a)}{\mathcal{V}(a) \vdash \mathsf{False}}$$

Figure 3.5: Rules for ownership. The resource algebra M must belong to the family of algebras used to instantiate the logic.

for any $t_1 : T$ and $t_2 : T$ we have:

$$\begin{aligned}\boxed{\mathsf{ex}(t_1)}^\gamma * \boxed{\mathsf{ex}(t_2)}^\gamma &\vdash \boxed{\mathsf{ex}(t_1) \cdot \mathsf{ex}(t_2)}^\gamma \\ &\vdash \boxed{\lightning}^\gamma \\ &\vdash \mathcal{V}(\lightning) \\ &\vdash \mathsf{False}\end{aligned}$$

hence, only one resource from this algebra can be owned at a time, explaining the name "exclusive".

As mentioned above, the separating conjunction is substructural. In particular, $P \not\vdash P * P$ for arbitrary P because assertions represent ownership of resources. Concretely, we can see that if we take P to be $\boxed{a}^\gamma$ then we only expect $\boxed{a}^\gamma \vdash \boxed{a}^\gamma * \boxed{a}^\gamma$ to hold if $a = a \cdot a$. If a is the core of some other resource, then we know that $a = a \cdot a$ does indeed hold, so ownership assertions for this resource can be duplicated. More generally, if the resources that satisfy an assertion P are cores, then ownership of P can be duplicated. The property that an assertion holds for resources which are cores is internalized in the logic with a modality $\Box$, which is called "persistently" or "intuitionistically" because $\Box P$ can be duplicated and behaves like a fully-structural intuitionistic assertion[4]. Rules for this modality are shown in Figure 3.6. To see how this modality is used, we will show that $\boxed{a}^\gamma \vdash \boxed{a}^\gamma * \mathcal{V}(a)$, that is, from ownership of a resource, we can continue to own it and (separately) know it is valid. We have:

$$\begin{aligned}\boxed{a}^\gamma &\vdash \boxed{a}^\gamma \wedge \boxed{a}^\gamma \\ &\vdash \boxed{a}^\gamma \wedge \mathcal{V}(a) \\ &\vdash \boxed{a}^\gamma \wedge \Box\mathcal{V}(a) \\ &\vdash \boxed{a}^\gamma * \Box\mathcal{V}(a) \\ &\vdash \boxed{a}^\gamma * \mathcal{V}(a)\end{aligned}$$

where the second to last line follows from Pers-Sep.

3.4 Weakest Preconditions

We now discuss the connective that lets us use the logic to state and prove properties of programs. In Iris, the *weakest precondition* assertion has the form $\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$. Recall from the

[4]This modality is similar in some ways to the exponential modality ! from linear logic [47].

$$\frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad \Box P \vdash P \qquad \Box P \vdash \Box\Box P \qquad \Box \forall x : \tau.\, P \dashv\vdash \forall x : \tau.\, \Box P$$

$$\Box \exists x : \tau.\, P \dashv\vdash \exists x : \tau.\, \Box P \qquad \textsc{Pers-Sep}\ \Box P \wedge Q \vdash \Box P * Q \qquad \textsc{Pers-Dup}\ \Box P \vdash \Box P * \Box P$$

$$\frac{|a| = \mathsf{Some}\ b}{\boxed{a : M}^{\gamma} \vdash \Box \boxed{b : M}^{\gamma}} \qquad \mathcal{V}(a) \vdash \Box \mathcal{V}(a)$$

Figure 3.6: Rules for $\Box$ modality.

introduction that the notion of weakest precondition was introduced by Dijkstra [33] as an alternative to Hoare triples. In the context of Iris, a resource satisfies $\mathsf{wp}_{\mathcal{E}}\ e\ \{x.\, P\}$ if from ownership of that resource, we can ensure that e will not fault, and if it terminates with a value v, then $[v/x]P$ will be satisfied. Moreover, any threads forked by e will not fault either. We will state this guarantee more precisely when we discuss the soundness theorem of Iris. The subscripted $\mathcal{E}$ is a set of natural numbers, which we call a *mask*. When $\mathcal{E}$ is the full set $\mathbb{N}$, we omit writing it. We also use $\top$ as an alternative notation for the full mask $\mathbb{N}$. For now, we will ignore the role of the mask.

Hoare triples can be encoded using weakest preconditions as follows:

$$\{P\}\, e\, \{x.\, Q\} \triangleq \Box\,(P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{x.\, Q\})$$

i.e., given resources satisfying P, we can prove a corresponding weakest precondition whose post-condition matches that of the triple. The separating implication is wrapped in the $\Box$ modality to ensure that the only non-duplicable resources needed for verification of e are those obtained in P. Although the Hoare triple notation may be more familiar to some, it is more convenient to work with the weakest precondition form, both when constructing derivations in the logic and when doing meta-theory[5]. For the instantiation of Iris with the ML-like language we are considering in this section, one can also define a points-to assertion $l \mapsto v$, which asserts ownership of the heap location l and indicates that it contains the value v, as usual in separation logic. In particular, $l \mapsto v * l \mapsto v' \vdash \mathsf{False}$, since we cannot have separate ownership of the same location l.

Generic structural rules for weakest precondition that hold for any language used with Iris are shown in Figure 3.7, and specialized rules for the ML-like language using the points-to assertion are given in Figure 3.8. The rule WP-Frame is a version of the frame rule for weakest precondition. Intuitively, it says that if Q is disjoint from whatever resources are used to establish the weakest precondition of e, then they must not be changed during execution of e, so Q will hold in the postcondition. WP-Value says that the weakest precondition of a value v

[5]Observe that the weakest precondition is a kind of modality, indexed by expressions. From this perspective, the preference for working directly with weakest precondition rather than Hoare triples is similar to how modal logicians work with a primitive modality $\Box P$ rather than a compound proposition equivalent to $Q \Rightarrow \Box P$.

WP-Frame
$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P * Q\}$

WP-Value
$[v/x]P \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{x.\, P\}$

WP-Mono
$(\forall v.\, \Phi(v) \mathrel{-\!\!*} \Psi(v)) * \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \Phi(x)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \Psi(x)\}$

Step-Later-Frame
$$\frac{\mathrm{expr\_to\_val}(e) = \mathsf{None}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} * \triangleright Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P * Q\}}$$

WP-Bind
$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, K[x]\, \{x'.\, P\}\} \dashv\vdash \mathsf{wp}_{\mathcal{E}}\, K[e]\, \{x'.\, P\}$

Figure 3.7: Generic structural weakest precondition rules.

can be established by showing that the postcondition holds for that value. WP-Mono is a form of the rule of consequence for weakest preconditions. The rule Step-Later-Frame is a strengthening of the frame rule, where we finally have a connection between steps of execution and the later modality $\triangleright$: if e is not a value, then at the point the postcondition holds, at least one step will have elapsed and so we can convert $\triangleright Q$ into just Q there. Finally, WP-Bind lets us decompose the program into a subexpression e and an evaluation context K, and then prove weakest preconditions for e and K filled with whatever e may return.

The language-specific rules in Figure 3.8 are written in a "continuation passing style", in which the post-condition of the weakest precondition is some arbitrary predicate Φ, and the left of the entailment is divided into an assertion needed to execute the command and an implication showing that the post-condition will hold after execution. For example, ml-store says that if we have $l \mapsto v$, and that $l \mapsto w \mathrel{-\!\!*} \Phi(())$ then we can deduce $\mathsf{wp}\ l := w\ \{\Phi\}$. Moreover, we only need to have the implication at one time step later, because it only needs to hold after we are done executing the store. This is what one might expect, because to do the store we need permission to write to the location (represented by $l \mapsto v$), and after the store is completed, l will point to w and the return value will be $()$, so we should get back $l \mapsto w$ and need to prove the post-condition. An alternative way of presenting this rule would be:

$$\triangleright\, l \mapsto v \vdash \mathsf{wp}_{\mathcal{E}}\, l := w\, \{x.\, \ulcorner x = () \urcorner * l \mapsto w\} \tag{3.1}$$

However, the style in Figure 3.8 is easier to use when doing fully formal proofs, because it can be applied no matter what the shape of the postcondition is, whereas the "direct" alternative requires explicitly using the monotonicity and framing rules when one is trying to prove a weakest precondition that does not match the format of (3.1).

The rest of the rules in Figure 3.8 can be understood similarly to ml-store, by comparing the rule with the operational semantics of the corresponding command. The interesting case is ml-fork, where we must prove that the post condition Φ of the parent thread holds in one step and separately must prove a weakest precondition for the newly forked thread running e, where

ML-ALLOC
$$\triangleright (\forall l.\, l \mapsto v \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{ref}\ v\ \{\Phi\}$$

ML-LOAD
$$\triangleright\, l \mapsto v * \triangleright (l \mapsto v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}_{\mathcal{E}}\ !l\ \{\Phi\}$$

ML-STORE
$$\triangleright\, l \mapsto v * \triangleright (l \mapsto w \mathrel{-\!\!*} \Phi(())) \vdash \mathsf{wp}_{\mathcal{E}}\ l := w\ \{\Phi\}$$

ML-FAA
$$\triangleright\, l \mapsto z_1 * \triangleright (l \mapsto (z_1 + z_2) \mathrel{-\!\!*} \Phi(z_1)) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{FAA}(l, k)\ \{\Phi\}$$

ML-CAS-FAIL
$$\triangleright\, l \mapsto w * \ulcorner w \neq v_1 \urcorner * \triangleright (l \mapsto w \mathrel{-\!\!*} \Phi(\mathsf{false})) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{CAS}(l, v_1, v_2)\ \{\Phi\}$$

ML-CAS-SUCCESS
$$\triangleright\, l \mapsto v_1 * \triangleright (l \mapsto v_2 \mathrel{-\!\!*} \Phi(\mathsf{true})) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{CAS}(l, v_1, v_2)\ \{\Phi\}$$

ML-REC
$$\triangleright\, \mathsf{wp}_{\mathcal{E}}\left([v/x][(\mathsf{rec}\ f\ x.\ e)/f]e\right)\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\left((\mathsf{rec}\ f\ x.\ e)\ v\right)\{\Phi\}$$

ML-FORK
$$\triangleright\, \Phi(()) * \triangleright\, \mathsf{wp}_{\top}\ e\ \{\mathsf{True}\} \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{fork}\{e\}\ \{\Phi\}$$

Figure 3.8: Specific weakest precondition rules for the ML-like language.

the postcondition is simply True. This postcondition suffices because we merely want to ensure that e does not get stuck, and do not care about the value it returns. To better understand the ML-FORK rule, consider the following derived rule that is obtained by combining ML-FORK with WP-BIND:

$$\triangleright\, \mathsf{wp}\left(K[()]\right)\{\Phi\} * \triangleright\, \mathsf{wp}\ e\ \{\mathsf{True}\} \vdash \mathsf{wp}\left(K[\mathsf{fork}\{e\}]\right)\{\Phi\}$$

Recalling the resource interpretation of separation logic connectives, this says that to verify a program which forks an expression e in an evaluation context K, it suffices to establish, using separate resources, weakest preconditions for the continuation of the parent thread and the newly forked child thread e. This is therefore analogous to the reading of O'Hearn's concurrent composition rule discussed at the beginning of §1.2.3.

To see how these weakest precondition rules are used, consider the following very simple program:

$$\mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fork}\{!l\}$$

The program allocates a new location with initial value 0, then forks a thread which simply reads from that location. We will establish an entailment involving a weakest precondition whose meaning implies that the program does not get stuck:

$$\mathsf{True} \vdash \mathsf{wp}\left(\mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fork}\{!l\}\right)\{x.\ \mathsf{True}\} \tag{3.2}$$

We will start by giving a fully formal proof using the rules explained so far. The proof proceeds by a form of backward reasoning. We will repeatedly apply the weakest precondition rules,

along with transitivity of entailment, to "simplify" the right hand side of the entailment. To start and illustrate the general pattern, observe that by WP-Bind,

$$\mathsf{wp}\ \mathsf{ref}\ 0\ \{x.\ \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}\} \vdash \mathsf{wp}\ (\mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}$$

hence by transitivity of entailment, to establish (3.2) it suffices to show

$$\mathsf{True} \vdash \mathsf{wp}\ \mathsf{ref}\ 0\ \{x.\ \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}\}$$

In the following, we will omit indicating that we are using transitivity of entailment, simply writing

$$\frac{A}{B}\ \mathrm{H}$$

to indicate that to prove the entailment B, it suffices to show A, where the optional annotation H indicates the primary rule used to justify this step. Continuing, we have then:

$$\dfrac{\dfrac{\mathsf{True} \vdash \forall x.\ x \mapsto 0 \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}}{\mathsf{True} \vdash \triangleright (\forall x.\ x \mapsto 0 \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\})}\ \textsc{Later-Intro}}{\mathsf{True} \vdash \mathsf{wp}\ \mathsf{ref}\ 0\ \{x.\ \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}\}}\ \textsc{ml-alloc}$$

Recall from Figure 3.2 to that to introduce a universal quantifier in the logic, it suffices to prove the corresponding statement quantified at the meta-level. Hence, let x be an arbitrary program location (*i.e.,* an arbitrary positive natural number). Then, from what we have established so far, it suffices to show $\mathsf{True} \vdash x \mapsto 0 \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}$, for which we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{True} \vdash \triangleright\,\mathsf{True}}\ \textsc{Later-Intro} \quad \dfrac{\dfrac{\dfrac{}{x \mapsto 0 \vdash \triangleright\, x \mapsto 0}\ \textsc{Later-Intro} \quad \dfrac{}{x \mapsto 0 \vdash \mathsf{True}}}{x \mapsto 0 \vdash \triangleright\,(x \mapsto 0 \mathrel{-\!\!*} \mathsf{True})}}{x \mapsto 0 \vdash \triangleright\, x \mapsto 0 * \triangleright\,(x \mapsto 0 \mathrel{-\!\!*} \mathsf{True})}}{x \mapsto 0 \vdash (\mathsf{wp}\ !x\ \{\mathsf{True}\})}\ \textsc{ml-load}}{x \mapsto 0 \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})}\ \textsc{Later-Intro}}{\mathsf{True} * x \mapsto 0 \vdash \triangleright\,\mathsf{True} * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})}}{x \mapsto 0 \vdash \triangleright\,\mathsf{True} * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})}}{x \mapsto 0 \vdash \mathsf{wp}\ \mathsf{fork}\{!x\}\ \{\mathsf{True}\}}\ \textsc{ml-fork}}{x \mapsto 0 \vdash \triangleright\,(\mathsf{wp}\ \mathsf{fork}\{!x\}\ \{\mathsf{True}\})}\ \textsc{Later-Intro}}{x \mapsto 0 \vdash \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}}\ \textsc{ml-rec}}{\mathsf{True} \vdash x \mapsto 0 \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{let}\ l = x\ \mathsf{in}\ \mathsf{fork}\{!l\})\ \{\mathsf{True}\}}$$

3.5 Invariants and Updates for Concurrency

Although the example we just considered involved forking a new thread, there was no subsequent interaction between the parent thread and the child thread. Consider the slightly different

program

$$\begin{array}{l}\mathsf{let}\ l = \mathsf{ref}\ 0\ \mathsf{in}\\ \mathsf{let}\ \_ = \mathsf{fork}\{!l\}\ \mathsf{in}\\ !l\end{array}$$

in which the parent thread now reads l after forking the child. If we try to adapt the proof for the first example to this version, we get stuck when we reach the part of the derivation after invoking ML-FORK. Instead of having to show

$$x \mapsto 0 \vdash \triangleright\,\mathsf{True} * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

we now must show

$$x \mapsto 0 \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\}) * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

where we have to prove a weakest precondition for the parent and child thread, which are now both going to read from l. However, the mechanisms of Iris that we have discussed so far do not suffice to proceed, because we would need to duplicate the assumption $x \mapsto 0$ to prove the separating conjunction, which we cannot do.

In general, to reason about the interaction of threads, we need to state the conventions that each is supposed to follow (*e.g.,* an analogue of the *rely* and *guarantee* conditions explained in Chapter 1). The mechanism for doing this in Iris is to use a combination of special resources and an assertion called an *invariant*. The proposition $\boxed{P}^{\iota}$ indicates that P has been established as an invariant, and the invariant has been assigned the name ι, which is a natural number. These names are used just as a book-keeping device to keep track of the status of various invariants. The idea is that once an invariant $\boxed{P}^{\iota}$ is established, it can be temporarily "opened" to obtain resources satisfying P, and then later it can be "closed" by giving up (possibly different) resources satisfying P. While open, we say the invariant is disabled, and when closed it becomes enabled again. Once an invariant is established, the various threads must ensure that it remains enabled after they complete a step, but correspondingly can expect that it will be enabled at the start of each step, so that they may temporarily open it. The assertion $\boxed{P}^{\iota}$ just represents knowledge that a certain invariant exists, so it can be duplicated, and we have $\boxed{P}^{\iota} \vdash \Box\boxed{P}^{\iota}$.

The "opening" and "closing" manipulations of invariants are done using the *update modality*[6] ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$, where $\mathcal{E}_1$ and $\mathcal{E}_2$ are sets of invariant names. A resource satisfies this assertion if, by opening invariants with names in $\mathcal{E}_1$, performing certain transformations of resources, and then closing invariants, we end up with a resource satisfying P and all of the invariants in $\mathcal{E}_2$ are enabled. We write $\top$ for the full set of all invariant names, and we write $\Rrightarrow_{\mathcal{E}}$ as an abbreviation for ${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}}$. The kinds of transformations of resources permitted are those which preserve validity of composition with other resources that can validly "co-exist" with a. Formally, given a set of resources B, we define $a \rightsquigarrow B$ to hold if for all a' such that $\mathcal{V}(a \cdot a')$, there exists b in B such that $\mathcal{V}(b \cdot a')$ holds. If $a \rightsquigarrow B$ holds, we say that a can be (non-deterministically) updated to an element of B.

[6] Jung et al. [66] call this the "fancy update" modality, in contradistinction to a more primitive "basic" modality. However, the latter will not be described here, so we will not use the word 'fancy'.

Rules for the update modality are shown in Figure 3.9. The rule Res-Update lets us transform a resource within an ownership assertion. The rule Res-Alloc lets us create or "allocate" an initial resource for some fresh ghost name γ, which is non-deterministically selected.

Invariants are manipulated using the rules InvAlloc and InvOpen. The first lets us allocate a new invariant for P if we can prove that $\triangleright P$ holds[7]. We can select some infinite set $\mathcal{E}'$ and are guaranteed that the new ι for the invariant belongs to this set. InvOpen lets us do an update to open an invariant $\boxed{P}^{\iota}$, so long as ι is in the set of enabled invariants for the update. Underneath the update modality, we obtain $\triangleright P$ and a separating implication that lets us close the invariant by giving up $\triangleright P$ again.

Rule Upd-Intro lets us introduce the update modality. If we have resources satisfying P, then they continue to satisfy P if in between we open an invariant and then later close it. Upd-Trans is a kind of transitivity property that lets us collapse nested updates, so long as the invariants needed for the inner modality are precisely the ones enabled at the end of the outer modality. The update modality has a frame rule for propositions (Upd-Frame) and for additional invariants (Upd-Mask-Frame). The latter is justified by the fact that if we did not rely on the fact that an invariant in $\mathcal{E}_f$ was enabled to establish ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$, then we could not have disabled that invariant.

The rule Upd-Timeless says that we can also replace a later modality with an update modality when the proposition under the modality belongs to a class of propositions which are called *timeless*. Figure 3.10 lists various timeless propositions and closure properties for this class of propositions. In particular, both the points-to assertion and resource ownership assertions are timeless[8], and timelessness is closed under the basic connectives of separation logic. When working with timeless propositions, the following rule, which can be derived from Upd-Timeless, Upd-Mono, and Upd-Trans, is useful:

$$\textsc{Upd-Timeless'}\quad \frac{\mathsf{timeless}(P)}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} \triangleright P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P}$$

Finally, we have rules that let us use the update modality while proving weakest preconditions, and we can now explain the meaning of the mask $\mathcal{E}$ in the weakest precondition $\mathsf{wp}_{\mathcal{E}}\ e\ \{x.\ P\}$: it represents the set of invariants that are enabled for use while verifying e. The rules Upd-WP and WP-Upd let us perform updates before a weakest precondition or in the postcondition. Atomic says that when reasoning about an atomic expression e we can temporarily open up invariants (via the outer ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$) so long as we close them in the postcondition (which is done by the ${}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1}$). This is sound because e is atomic, so it will take a single step, and therefore no other thread can observe that the invariant was disabled "during" the execution of e.

Let us now see how invariants can be used to solve the problem we encountered when we had two threads that needed to read from the same memory location. Recall that we needed to

[7]Recall that $P \vdash \triangleright P$.

[8]When using the more general notion of *camera* in Iris instead of the simplified resource algebras we are considering here, ownership of ghost resources is not necessarily timeless.

RES-ALLOC
$$\frac{G \text{ is finite} \qquad \mathcal{V}(a)}{P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \exists \gamma \notin G.\ \boxed{a : M}^{\gamma}}$$

RES-UPDATE
$$\frac{a \rightsquigarrow A}{\boxed{a}^{\gamma} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \exists a' \in A.\ \boxed{a'}^{\gamma}}$$

RES-UNIT-ALLOC
$$\frac{M \text{ has a unit}}{P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \boxed{\varepsilon : M}^{\gamma}}$$

INVALLOC
$$\frac{\mathcal{E}' \text{ infinite}}{\triangleright P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \exists \iota.\ulcorner \iota \in \mathcal{E}' \urcorner * \boxed{P}^{\iota}}$$

INVOPEN
$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\{\iota\}} (\triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}))}$$

UPD-INTRO
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1} P}$$

UPD-TRANS
$${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} P$$

UPD-MONO
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}$$

UPD-FRAME
$$({}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P) * Q \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} (P * Q)$$

UPD-MASK-FRAME
$$\frac{\mathcal{E}_1 \cap \mathcal{E}_f = \emptyset}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \cup \mathcal{E}_f}\!\Rightarrow^{\mathcal{E}_2 \cup \mathcal{E}_f} P}$$

INV-PERS
$$\boxed{P}^{\iota} \vdash \square \boxed{P}^{\iota}$$

UPD-TIMELESS
$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P}$$

UPD-WP
$${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$$

WP-UPD
$$\mathsf{wp}_{\mathcal{E}}\, e \left\{x.\ {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P\right\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\ P\}$$

WP-MONO-UPD
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{(\forall v.\, \Phi(v) \mathrel{-\!\!*} \Rightarrow_{\mathcal{E}_2} \Psi(v)) * \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, \Phi(x)\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\, \Psi(x)\}}$$

ATOMIC
$$\frac{\mathrm{atomic}(e)}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e \left\{x.\ {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1} P\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\ P\}}$$

Figure 3.9: Rules for invariants and $\Rightarrow$ modality.

$$\mathsf{timeless}(l \mapsto v) \qquad \mathsf{timeless}(\boxed{a}^{\gamma}) \qquad \mathsf{timeless}(\mathcal{V}(a))$$

$$\frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \wedge Q)} \qquad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)}$$

$$\frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \vee Q)} \qquad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \Rightarrow Q)}$$

$$\frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \mathrel{-\!\!*} Q)} \qquad \frac{\forall t : \tau.\, \mathsf{timeless}(\Phi(t))}{\mathsf{timeless}(\forall x : \tau.\, \Phi(x))} \qquad \frac{\forall t : \tau.\, \mathsf{timeless}(\Phi(t))}{\mathsf{timeless}(\exists x : \tau.\, \Phi(x))}$$

Figure 3.10: Timeless assertions.

show

$$x \mapsto 0 \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\}) * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

but we could not duplicate the $x \mapsto 0$ assertion to give to both threads. To handle this, before forking we will create an invariant of the from $\boxed{x \mapsto 0}$. Then, each thread will be able to open the invariant, perform its respective read, and then close the invariant. The following derivation shows how the invariant is created prior to forking:

$$\dfrac{\dfrac{\dfrac{\dfrac{\exists \iota.\, \boxed{x \mapsto 0}^{\iota} \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\}) * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})}{\exists \iota.\, \boxed{x \mapsto 0}^{\iota} \vdash (\mathsf{wp}\ (\mathsf{let}\ \_ = \mathsf{fork}\{!x\}\ \mathsf{in}\ !x)\ \{\mathsf{True}\})}\ \textsc{ml-fork},\textsc{WP-Bind},\textsc{ml-rec}}{\Rrightarrow_{\top} \exists \iota.\, \boxed{x \mapsto 0}^{\iota} \vdash \Rrightarrow_{\top} (\mathsf{wp}\ (\mathsf{let}\ \_ = \mathsf{fork}\{!x\}\ \mathsf{in}\ !x)\ \{\mathsf{True}\})}\ \textsc{Upd-Mono}}{x \mapsto 0 \vdash \Rrightarrow_{\top} (\mathsf{wp}\ (\mathsf{let}\ \_ = \mathsf{fork}\{!x\}\ \mathsf{in}\ !x)\ \{\mathsf{True}\})}\ \textsc{InvAlloc}}{x \mapsto 0 \vdash (\mathsf{wp}\ (\mathsf{let}\ \_ = \mathsf{fork}\{!x\}\ \mathsf{in}\ !x)\ \{\mathsf{True}\})}\ \textsc{WP-Upd}$$

Eliminating the existential, we just have to show that for arbitrary ι,

$$\boxed{x \mapsto 0}^{\iota} \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\}) * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

Because $\boxed{x \mapsto 0}^{\iota} \vdash \Box \boxed{x \mapsto 0}^{\iota}$ we can duplicate the invariant assertion, so that it suffices to show that

$$\boxed{x \mapsto 0}^{\iota} * \boxed{x \mapsto 0}^{\iota} \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\}) * \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

This splits into two identical cases, where in each we must prove

$$\boxed{x \mapsto 0}^{\iota} \vdash \triangleright\,(\mathsf{wp}\ !x\ \{\mathsf{True}\})$$

which is established by the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\triangleright x \mapsto 0 \vdash \triangleright x \mapsto 0} \qquad \dfrac{}{\triangleright x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True} \vdash x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True}}}{\triangleright x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True} \vdash \triangleright (x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True})}}{\triangleright x \mapsto 0 * (\triangleright x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True}) \vdash \triangleright x \mapsto 0 * \triangleright (x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True})}}{\triangleright x \mapsto 0 * (\triangleright x \mapsto 0 \mathrel{-\!\!*} {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True}) \vdash \mathsf{wp}_{\top\setminus\{\iota\}}\, !x \left\{ {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True} \right\}}\ \textsc{ml-load}}{\boxed{x \mapsto 0}^{\iota} \vdash {}^{\top}\!\Rrightarrow^{\top\setminus\{\iota\}} \left(\mathsf{wp}_{\top\setminus\{\iota\}}\, !x \left\{ {}^{\top\setminus\{\iota\}}\!\Rrightarrow^{\top} \mathsf{True} \right\} \right)}\ \textsc{InvOpen}, \textsc{Upd-Mono}}{\boxed{x \mapsto 0}^{\iota} \vdash (\mathsf{wp}\, !x \, \{\mathsf{True}\})}\ \textsc{Atomic}}{\boxed{x \mapsto 0}^{\iota} \vdash \triangleright (\mathsf{wp}\, !x \, \{\mathsf{True}\})}\ \textsc{Later-Intro}$$

The top right of the derivation follows immediately from the fact that $x \mapsto 0 \vdash \triangleright x \mapsto 0$.

Although this use of invariants sufficed for this example, this solution is not completely satisfying. First, once we have put $x \mapsto 0$ into an invariant, no thread can ever change the value of that location. Second, the solution seems slightly ad-hoc: do we really need to create such a special invariant each time we want to share access to such a location?

We can obtain a more flexible solution by defining a simplified form of *fractional permissions* [24, 25]. The idea behind fractional permissions is to replace the points-to assertion $l \mapsto v$ with an indexed version $l \overset{q}{\mapsto} v$, where q is a rational number such that $0 < q \leq 1$. The q represents "fractional" ownership of the location. When $q = 1$, the assertion is said to represent "full ownership", and all of the rules in Figure 3.8 hold with $l \overset{1}{\mapsto} v$ in place of $l \mapsto v$. In particular, the location l can be read and modified. However, if $q < 1$, then the assertion represents only partial ownership, and the location l may only be read but not modified using the permission. That is, when $q < 1$, we only have an analogue of ML-LOAD:

ML-LOAD-FRAC
$$\triangleright l \overset{q}{\mapsto} v * \triangleright (l \overset{q}{\mapsto} v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}\, !l \, \{\Phi\}$$

Permissions can be joined and split using the rule

$$l \overset{q_1}{\mapsto} v_1 * l \overset{q_2}{\mapsto} v_2 \dashv\vdash l \overset{q_1+q_2}{\mapsto} v_1 * \ulcorner q_1 + q_2 \leq 1 \land v_1 = v_2 \urcorner$$

Read from left to right, this says that we can combine two fractional permissions for a location l. After doing so, we will also know that the combined permission must be less than or equal to 1, and that the values pointed to by l in each permission are equal. From right to left, we can split a permission so long as the split fractions add up to the original value.

With these fractional permissions, we can verify the example with a parent and child thread reading from a common location just by splitting a permission $l \overset{1}{\mapsto} v$ into $l \overset{1/2}{\longmapsto} v * l \overset{1/2}{\longmapsto} v$, and then passing each half to the two threads.

The original development of fractional permissions in separation logic [24] required a novel soundness proof with a new semantic model. However, the features of Iris are expressive enough to permit us to define them directly in terms of the original points-to assertion $l \mapsto v$ and the connectives we have already seen. To do so, we have to use a construction called the authoritative resource algebra [64]:

Definition 3.11. Let M be a resource algebra with a unit. The *authoritative RA* $\textsc{Auth}(M)$ has as a set of elements $\textsc{Opt}(\textsf{Ex}(M)) \times M$. Composition is defined component-wise, with validity defined by:

$$\begin{aligned}\mathcal{V}(\textsf{ex}(a)_\textsf{s}, b) &= \exists c.\, b \cdot c = a \wedge \mathcal{V}(a)\\ \mathcal{V}(\lightning_\textsf{s}, b) &= \textsf{False}\\ \mathcal{V}(\bot, b) &= \mathcal{V}(b)\end{aligned}$$

Recall that since M has a unit, its core operation is total. The core of $\textsc{Auth}(M)$ is defined by:

$$|(a, b)| = (\bot, b') \quad \text{where } |b| = \textsf{Some } b'$$

We define the notations $\bullet\, a \triangleq (\textsf{ex}(a)_\textsf{s}, \varepsilon)$ and $\circ\, a \triangleq (\bot, a)$.

The idea behind this construction is that $\bullet\, a$ represents some "authoritative" version of a resource, and $\circ\, a$ represents a "claim" or "stake" in a fragment of the authoritative resource. That is, ownership of $\circ\, a$ is meant to guarantee that the authoritative element can be written as a product $a \cdot b$ for some b.

We will model fractional permissions using the resource algebra

$$\textsc{Auth}(\textsc{FinMap}(\textsf{Loc}, \textsc{Frac} \times \textsc{Ag}(\textsf{Val})))$$

This algebra combines several constructions, making it initially appear complex, so we will describe several of its properties. Valid elements of the algebra $\textsc{FinMap}(\textsf{Loc}, \textsc{Frac} \times \textsc{Ag}(\textsf{Val}))$ consist of finite partial functions mapping locations to pairs of the form $(q, \textsf{ag}(v))$, where q is a fraction and v is a value of the ML-like language. We think of these as representing subsets of the program heap, where each location in the domain is also associated with a fractional value q. We will omit writing the $\textsf{ag}(-)$ wrapper everywhere since any mapping that sends an element to $\lightning$ is invalid. Recall that we use the notation $\{l \mapsto (q, v)\}$ for the partial function whose domain is the singleton l, which is sent to (q, v). Using the definitions of the operations for the various constituent resource algebras, we obtain the following facts:

1. If $\mathcal{V}(\bullet f \cdot \circ \{l \mapsto (q, v)\})$, then $l \in \mathrm{dom}(f)$, $q \leq 1$, and $f(l) = v$.
2. If $\mathcal{V}(\circ \{l \mapsto (q_1, v_1)\} \cdot \circ \{l \mapsto (q_2, v_2)\})$, then

   $$\circ \{l \mapsto (q_1, v_1)\} \cdot \circ \{l \mapsto (q_2, v_2)\} = \circ \{l \mapsto (q_1 + q_2, v_1)\}$$

   and both $q_1 + q_2 \leq 1$, and $v_1 = v_2$.

Using these facts about validity, we have corresponding rules for ownership:

$$\boxed{\bullet f}^\gamma * \boxed{\circ \{l \mapsto (q, v)\}}^\gamma \vdash \ulcorner l \in \mathrm{dom}(f) \wedge f(l) = v \wedge q < 1 \urcorner$$

$$\boxed{\circ \{l \mapsto (q_1, v_1)\}}^\gamma * \boxed{\circ \{l \mapsto (q_2, v_2)\}}^\gamma \dashv\vdash \boxed{\circ \{l \mapsto (q_1 + q_2, v_1)\}}^\gamma * \ulcorner q_1 + q_2 \leq 1 \wedge v_1 = v_2 \urcorner$$

The second is similar to the form of the join/split rule for fractional permissions mentioned above. Indeed, we will now define the fractional permission as ownership of an appropriate singleton resource. Given a ghost name γ, we define

$$l \overset{q}{\mapsto}_\gamma v \triangleq \boxed{\circ \{l \mapsto (q, v)\}}^\gamma$$

The second rule about ownership of these resources then becomes:

$$l \overset{q_1}{\mapsto}_\gamma v_1 * l \overset{q_2}{\mapsto}_\gamma v_2 \dashv\vdash l \overset{q_1+q_2}{\mapsto}_\gamma v_1 * \ulcorner q_1 + q_2 \leq 1 \land v_1 = v_2 \urcorner$$

The only difference relative to regular fractional points-to assertions is that the assertion has to be annotated with the ghost name γ.

Now, we just need to link this resource to the corresponding state of the actual program and show that the weakest precondition rules for fractional permissions hold for this encoding. We will establish this linkage using an invariant. We first define an assertion $\mathsf{FracInv}_\gamma$:

$$\mathsf{FracInv}_\gamma \triangleq \exists f. \boxed{\bullet f}^\gamma * \mathop{\Large *}_{l \in \mathrm{dom}(f)} l \mapsto \mathsf{snd}\,(f(l))$$

The large $\mathop{\Large *}$ above represents an iterated separating conjunction. This states ownership of the full resource for some map f, and that for every location l in the domain of f, there is a points-to assertion mapping l to $\mathsf{snd}\,(f(l))$. We will show that combined with the knowledge that an invariant $\boxed{\mathsf{FracInv}_\gamma}^\iota$ holds, the fractional permission we have defined suffices to obtain the expected weakest precondition rules. We explain the proof for allocation in detail, and briefly describe the argument for loading and storing. We start with the following lemma:

Lemma 3.12. For all γ, we have $\mathsf{timeless}(\mathsf{FracInv}_\gamma)$.

Proof. Unfolding the definition, and using the fact that timelessness is closed under existentials, it suffices to show that for all f we have $\mathsf{timeless}(\boxed{\bullet f}^\gamma * \mathop{\Large *}_{l \in \mathrm{dom}(f)} l \mapsto \mathsf{snd}\,(f(l)))$. This follows by induction on the domain of f, using the fact that timelessness is closed under separating conjunction. □

We now establish a weakest precondition rule for allocation:

Lemma 3.13. Let $\mathcal{E}$ be a mask such that $\iota \in \mathcal{E}$. Then, the entailment

$$\boxed{\mathsf{FracInv}_\gamma}^\iota * \triangleright (\forall l.\, l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{ref}\, v\, \{\Phi\}$$

holds.

Remark. Unlike ML-ALLOC, which works with an arbitrary mask $\mathcal{E}$, we have to assume here that the invariant we are using for fractional permissions is enabled.

Proof. To start we have:

$$\dfrac{\dfrac{\dfrac{\triangleright\left(\boxed{\mathsf{FracInv}_\gamma}^{\iota} * (\forall l.\ (l \overset{1}{\mapsto}_\gamma v) \mathrel{-\!\!*} \Phi(l))\right) \vdash \triangleright(\forall l.\ l \mapsto v \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \Phi(l))}{\boxed{\mathsf{FracInv}_\gamma}^{\iota} * \triangleright(\forall l.\ (l \overset{1}{\mapsto}_\gamma v) \mathrel{-\!\!*} \Phi(l)) \vdash \triangleright(\forall l.\ l \mapsto v \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \Phi(l))}\ \textsc{Later-Intro}, \textsc{Later-Sep}}{\boxed{\mathsf{FracInv}_\gamma}^{\iota} * \triangleright(\forall l.\ (l \overset{1}{\mapsto}_\gamma v) \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{ref}\ v\ \{\Rrightarrow_{\mathcal{E}} \Phi\}}\ \textsc{ml-alloc}}{\boxed{\mathsf{FracInv}_\gamma}^{\iota} * \triangleright(\forall l.\ (\exists \gamma.\ l \overset{1}{\mapsto}_\gamma v) \mathrel{-\!\!*} \Phi(l)) \vdash \mathsf{wp}_{\mathcal{E}}\ \mathsf{ref}\ v\ \{\Phi\}}\ \textsc{WP-Upd}$$

To continue, we apply Later-Mono, introduce the universal quantifier on the right, and eliminate the one on the left, so that it suffices to show that for arbitrary l,

$$\boxed{\mathsf{FracInv}_\gamma}^{\iota} * (l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)) \vdash l \mapsto v \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} \Phi(l)$$

Introducing the wand on the right we have to show

$$l \mapsto v * \boxed{\mathsf{FracInv}_\gamma}^{\iota} * (l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)) \vdash \Rrightarrow_{\mathcal{E}} \Phi(l)$$

From here it suffices to show that $l \mapsto v * \boxed{\mathsf{FracInv}_\gamma}^{\iota} \vdash \Rrightarrow_{\mathcal{E}} l \overset{1}{\mapsto}_\gamma v$, since assuming this we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\Phi(l) \vdash \Phi(l)}{l \overset{1}{\mapsto}_\gamma v * \left(l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)\right) \vdash \Phi(l)}}{\Rrightarrow_{\mathcal{E}}\left(l \overset{1}{\mapsto}_\gamma v * \left(l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)\right)\right) \vdash \Rrightarrow_{\mathcal{E}} \Phi(l)}\ \textsc{Upd-Mono}}{(\Rrightarrow_{\mathcal{E}}(l \overset{1}{\mapsto}_\gamma v) * (l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l)) \vdash \Rrightarrow_{\mathcal{E}} \Phi(l)}\ \textsc{Upd-Frame}}{l \mapsto v * \boxed{\mathsf{FracInv}_\gamma}^{\iota} (l \overset{1}{\mapsto}_\gamma v \mathrel{-\!\!*} \Phi(l) \vdash \Rrightarrow_{\mathcal{E}} \Phi(l)}$$

To complete the last step of showing that $l \mapsto v * \boxed{\mathsf{FracInv}_\gamma}^{\iota} \vdash \Rrightarrow_{\mathcal{E}} l \overset{1}{\mapsto}_\gamma v$, we begin by opening the invariant:

$$\dfrac{\dfrac{l \mapsto v * \triangleright \mathsf{FracInv}_\gamma * \left(\triangleright \mathsf{FracInv}_\gamma \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} \mathsf{True}\right) \vdash {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} l \overset{1}{\mapsto} v}{{}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}}\left(l \mapsto v * \triangleright \mathsf{FracInv}_\gamma * \left(\triangleright \mathsf{FracInv}_\gamma \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right) \vdash \Rrightarrow_{\mathcal{E}} l \overset{1}{\mapsto} v}\ \textsc{Upd-Trans}, \textsc{Upd-Mono}}{l \mapsto v * \boxed{\mathsf{FracInv}_\gamma}^{\iota} \vdash \Rrightarrow_{\mathcal{E}} l \overset{1}{\mapsto} v}\ \textsc{InvOpen}$$

At this point, the full derivation tree becomes very unwieldy, so we will describe the next few steps in prose. Using the fact that $\mathsf{FracInv}_\gamma$ is timeless, we can eliminate the later guarding it. Unfolding the definition, we eliminate the existential to get that for some f, we have ownership of $\boxed{\bullet f}^{\gamma}$ and an iterated conjunction of points-to assertions in f. We know that $l \notin \mathrm{dom}(f)$, because if it were, then a second points-to assertion of the form $l \mapsto \mathsf{snd}\,(f(l))$ would be contained in this iterated points-to assertion. Yet, before opening the invariant we already had $l \mapsto v$, and $l \mapsto v * l \mapsto \mathsf{snd}\,(f(l)) \vdash \mathsf{False}$.

Since $l \notin \mathrm{dom}(f)$, we can transform the resource to update f to now map l to $(1, v)$. That is, we have that $\bullet f \rightsquigarrow \{\bullet f\,[l \mapsto (1,v)] \cdot \circ \{l \mapsto (1,v)\}\}$. To see why, consider all a' for which $\bullet f \cdot a'$ is valid. We must have that a' is $\circ f'$ for some f'. The domain of f' is a subset of f, so it is valid when composed with the extended map $f\,[l \mapsto (1,v)]$ and the singleton fragment $\circ\{l \mapsto (1,v)\}$.

Using Res-Update we therefore have:

$$\boxed{\bullet f}^{\gamma} \vdash \Rrightarrow_{\mathcal{E}\setminus\{\iota\}} \left(\boxed{\bullet f\,[l \mapsto (1,v)]}^{\gamma} * \boxed{\circ\{l \mapsto (1,v)\}}^{\gamma}\right)$$

The right conjunct is equivalent to $l \overset{1}{\mapsto}_{\gamma} v$. We will now establish $\mathsf{FracInv}_{\gamma}$, choosing the extended map $f\,[l \mapsto (1,v)]$ for the existential. To establish $\mathsf{FracInv}_{\gamma}$ for this map, we need all of the points-to assertions for the domain of f (which we obtained when we opened the invariant) as well as $l \mapsto v$, which we also have. Summarizing the above, it remains to show

$$\Rrightarrow_{\mathcal{E}\setminus\{\iota\}} \left(l \overset{1}{\mapsto} v * \mathsf{FracInv}_{\gamma} * (\triangleright\, \mathsf{FracInv}_{\gamma} \mathrel{-\!\!*}\, {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\right) \vdash {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} l \overset{1}{\mapsto} v$$

which follows by eliminating the wand and using the frame and transitivity rules for updates. □

Lemma 3.14. For all γ, if $\iota \in \mathcal{E}$, then the following entailments hold

$$\boxed{\mathsf{FracInv}_{\gamma}}^{\iota} * \triangleright\, l \overset{1}{\mapsto}_{\gamma} v * \triangleright\, (l \overset{1}{\mapsto}_{\gamma} w \mathrel{-\!\!*} \Phi(())) \vdash \mathsf{wp}_{\mathcal{E}}\, l := w\, \{\Phi\}$$

$$\boxed{\mathsf{FracInv}_{\gamma}}^{\iota} * \triangleright\, l \overset{q}{\mapsto} v * \triangleright\, (l \overset{q}{\mapsto} v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}_{\mathcal{E}}\, !l\, \{\Phi\}$$

Proof. The proofs of these rules are similar to the proof for allocation. In both cases, we use the corresponding rule for the non-fractional permission (ml-store or ml-load, respectively) and open the invariant for $\mathsf{FracInv}_{\gamma}$ to obtain access to the non-fractional points-to fact needed for these rules. An argument about the validity of the composition of the resource contained in the invariant and the resource in the fractional permission assertion ensure that the desired points-to fact will indeed exist. We then re-establish and close the invariant. In the case of ml-store this requires first updating the resource contained in the invariant to account for the newly stored value.

□

Finally, we just have to show that the $\mathsf{FracInv}_{\gamma}$ invariant can be proved at the beginning of a derivation:

Lemma 3.15. For all e, the following entailment holds

$$(\forall \iota, \gamma.\, \boxed{\mathsf{FracInv}_{\gamma}}^{\iota} \mathrel{-\!\!*} \mathsf{wp}\; e\, \{\Phi\}) \vdash \mathsf{wp}\; e\, \{\Phi\}$$

Proof. We will establish the invariant by choosing the empty map for the existential in $\mathsf{FracInv}_{\gamma}$. By Upd-WP, we may perform updates under the ${}^{\top}\!\Rrightarrow^{\top}$ modality. We first apply Res-Alloc to obtain ownership of $\boxed{\bullet \emptyset}^{\gamma}$ for some γ, where $\emptyset$ is the empty mapping. This suffices to establish $\mathsf{FracInv}_{\gamma}$, so that by using InvAlloc we have $\boxed{\mathsf{FracInv}_{\gamma}}^{\iota}$ for some ι. It then suffices to show

$${}^{\top}\!\Rrightarrow^{\top} \left(\boxed{\mathsf{FracInv}_{\gamma}}^{\iota} * (\forall \iota', \gamma'.\, \boxed{\mathsf{FracInv}_{\gamma'}}^{\iota'} \mathrel{-\!\!*} \mathsf{wp}\; e\, \{\Phi\})\right) \vdash {}^{\top}\!\Rrightarrow^{\top} \mathsf{wp}\; e\, \{\Phi\}$$

which follows by eliminating the quantifiers and the wand. □

This means that we can always start a verification of a program by assuming that $\boxed{\mathsf{FracInv}_\gamma}^\iota$ holds for some arbitrary ι and γ, and then we will be able to use the derived rules for fractional permissions mentioned above. Since the invariant assertion is duplicable, we can use it as we verify different threads of a program.

This example illustrates a common pattern in Iris, where we can use a combination of resources and invariants to encode coordination between threads. Many more examples can be found in the aforementioned Iris papers and the tutorial of Birkedal and Bizjak [22].

3.6 Style of Written Proofs

Unfortunately, formal derivations like the ones we've seen throughout this chapter are somewhat hard to read and write, and do not really convey the ideas behind the proof. Therefore, in the rest of this dissertation, rather than presenting proofs in Iris fully formally, we will write them in a style closer to the way normal mathematical proofs are written. When considering an entailment of the form $P_1 * \ldots * P_n \vdash Q$, we think of the $P_1, \ldots, P_n$ as a list of assumptions that may be used. Of course, the base logic is substructural, so some care is needed to make sure that these assertions are not "used twice". In order to prove an entailment like

$$P_1 * \ldots * P_n \vdash Q_1 * Q_2$$

it suffices to split the list of assumptions into disjoint sublists Δ_1 and Δ_2 and then show that from Δ_i we can prove Q_i. When proving an entailment like $P_1 * \ldots * P_n \vdash Q \mathrel{-\!\!*} R$ we will say things like "Assume Q, …, thus R", with the idea being that Q is added to this list of assumptions, from which we then prove R. Certain idioms arise from the interpretation of propositions as resources. For instance, when we have the assumption P and we know that $P \vdash Q$, we will sometimes say "we transform P to get Q" or we "give up P and get back Q" to mean that the assumption P is replaced by Q.

Another tedious aspect of formal derivations is the management of modalities like $\triangleright$. Note that in the above, the Later-Intro rule must be repeatedly invoked at many points. Even more bureaucratic manipulation is involved when the later modality appears on the left side of an entailment. For example, suppose we want to prove $P * \triangleright (P \mathrel{-\!\!*} Q) \vdash \triangleright Q$. To do so, we start by showing

$$\begin{aligned} P * \triangleright (P \mathrel{-\!\!*} Q) &\vdash \triangleright P * \triangleright (P \mathrel{-\!\!*} Q) \\ &\vdash \triangleright (P * (P \mathrel{-\!\!*} Q)) \end{aligned}$$

By then appealing to Later-Mono it suffices to show that $P * (P \mathrel{-\!\!*} Q) \vdash Q$, which is immediate. More generally, if we are trying to prove an entailment like:

$$P_1 * \ldots * P_n * \triangleright Q_1 * \ldots \triangleright Q_m \vdash \triangleright R$$

then it suffices to "strip off" a later modality guarding any assumption on the left and the goal on the right, and prove

$$P_1 * \ldots * P_n * Q_1 * \ldots Q_m \vdash R$$

Rather than always writing out the intermediary steps that justify this, we will describe this kind of reduction as "eliminating the $\triangleright$ modality".

Finally, notice that although we reason backwards (starting from the bottom of the derivation and working up), we end up reasoning about the program itself in a *forwards* direction. As we move up the derivation, rules like ML-ALLOC, ML-FORK, and ML-LOAD can be seen as symbolically "executing" steps of the program. It is convenient to speak of executing steps as shorthand for using these rules in a proof. Thus, rather than explicitly applying WP-BIND and then using a rule such as ML-ALLOC, we will instead say we allocate a new reference and obtain $x \mapsto 0$ as an assumption, for some new program location x.

Besides hopefully being more readable, this style is actually closer to how machine checked Iris proofs are done. Formally, this reasoning style is justified by defining a relation $\Gamma \Vdash Q$ where Γ is a list of propositions, representing a context of assumptions. This is defined by setting

$$(P_1, \dots, P_n \Vdash Q) \triangleq (P_1 * \cdots * P_n \vdash Q)$$

Then, various rules for this context formulation are derived from the rules for $\vdash$. We will not describe this further. The interested reader can see the references on the Iris proof mode [73, 74].

Chapter 4

Iris: Generic Framework and Soundness

In the previous chapter, we have seen how Iris can be used to reason about programs written in a concurrent ML-like language. However, as mentioned, Iris can be used to reason about various other programming languages as well. In this chapter, we discuss the more general framework and also show how soundness of the logic is established. Again, this chapter summarizes material from the Iris papers [64, 65, 66, 72, 116]. Readers familiar with these sources can skip this chapter. The only novelty is in §4.2, where the adequacy proof I describe is different from the version outlined by Jung et al. [66].

4.1 Generic Program Logic

With the exception of the weakest precondition, none of the connectives discussed in the previous chapter depend in any way on the semantics of the programming language which one wants to reason about. Indeed, they are all defined without reference to any language, and all the rules we have discussed so far about them are language independent.

Only the weakest precondition is parameterized by the semantics of a language. The assumptions made about this operational semantics are very weak, meaning it can be instantiated (in principle) with a wide range of programming languages. Based on the semantics, weakest precondition is defined in terms of the other Iris connectives. From there, rather generic "structural" rules are derived that will hold for any language that might be used with the framework. Finally, various "lifting" lemmas are developed to make it easier to prove rules specific to a given programming language (such as the ones we saw involving the points-to connective for the example language in the previous chapter).

4.1.1 Program Semantics

The assumptions made about the language used when defining weakest precondition are shown in Figure 4.1. The structure will seem familiar from the language described in the previous chapter. We assume there are three syntactic categories: expressions, values, and states, with partial coercions $\mathrm{expr\_to\_val}$ and $\mathrm{val\_to\_expr}$ between expressions and values. We say an expression

Syntax:

Val	v	:	...
Expr	e	:	...
State	σ	:	...
ThreadPool	T	:	List Expr

Syntactic Operations:

$$\begin{array}{rcl} \text{expr\_to\_val} & : & \mathsf{Expr} \to \mathsf{Option\ Val} \\ \text{val\_to\_expr} & : & \mathsf{Val} \to \mathsf{Expr} \end{array}$$

Per-Thread Reduction: $e; \sigma \to e'; \sigma'; T$

Figure 4.1: Input syntactic categories and judgments of generic concurrent language.

e is a value if

$$\text{expr\_to\_val}(e) \neq \mathsf{None}$$

In addition, there is a small step reduction relation describing how a single thread can take a step: $e, \sigma \to e', \sigma', T$ means that a thread executing e in a state σ can take a step to e', updating the state to σ' and creating new threads for each expression in the list T. It is assumed that values cannot take steps. As with the example language from the previous chapter, we define:

$$\text{red}(e, \sigma) \triangleq \exists e', \sigma', T.\, (e; \sigma \to e', \sigma', T)$$

and say that e is reducible in σ if this holds. An expression is again said to be *atomic* if whenever it can take a step, the resulting expression is not reducible:

$$\text{atomic}(e) \triangleq \forall \sigma, e', \sigma', T.\, (e; \sigma \to e'; \sigma'; T) \Rightarrow \neg \text{red}(e', \sigma')$$

We then lift this single-thread semantics into a concurrent semantics in the same way we did for the ML-like language in the previous chapter. Program configurations are tuples consisting of a non-empty list of threads and a state. Concurrent transitions are specified by the following non-deterministic rule:

$$\frac{e; \sigma \to e'; \sigma'; T_f}{T_1 +\!\!+ [e] +\!\!+ T_2; \sigma \to T_1 +\!\!+ [e'] +\!\!+ T_2 +\!\!+ T_f; \sigma'}$$

4.1.2 Weakest Precondition

Having fixed a language, a notion of weakest precondition is defined out of the other connectives we have seen. The idea is that if the weakest precondition wp $e\, \{x.\, P\}$ holds, then should we execute e in any starting state:

1. it will not reach a "stuck state", and neither will any threads generated during its execution, and

2. if it terminates with some value v, then $[v/x]P$ will hold in the resulting program state.

This will be stated more precisely below when we describe the adequacy statement of the logic.

Notice that so far we have assumed almost nothing about what the "state" of the language is composed of. Therefore, the definition of weakest precondition is also parameterized by a "state interpretation" predicate $S : \mathsf{State} \to \mathsf{iProp}$ which maps states to propositions. In full generality, we define an assertion $\mathsf{wp}^S_{\mathcal{E}}\, e\, \{x.\, P\}$, where the set $\mathcal{E}$ indicates what invariant names are enabled. The definition is as follows:

$$
\begin{aligned}
\mathsf{wp}^S \triangleq\ & \mu\, \mathsf{wp}.\ \lambda\, \mathcal{E}, e, \Phi. \\
& (\exists v.\ \mathrm{expr\_to\_val}(e) = \mathsf{Some}\ v \wedge \Rightarrow_{\mathcal{E}} \Phi(v)) \vee \\
& \Big(\mathrm{expr\_to\_val}(e) = \mathsf{None} \wedge \forall \sigma.\ S(\sigma) \mathrel{-\!\!*} \\
& \quad {}^{\mathcal{E}}\Rightarrow^{\emptyset} \Big(\mathrm{red}(e,\sigma) * \triangleright \forall e', \sigma', T.\ (e,\sigma \to e',\sigma',T) \mathrel{-\!\!*} \\
& \qquad {}^{\emptyset}\Rightarrow^{\mathcal{E}} \Big(S(\sigma') * \mathsf{wp}(\mathcal{E}, e', \Phi) * \textstyle\bigstar_{e'' \in T} \mathsf{wp}(\top, e'', \lambda\, \_.\ \mathsf{True})\Big)\Big)\Big)
\end{aligned}
$$

This is a guarded fixed point, consisting of a disjunction. The first disjunct says that when e is a value, we should be able to prove that the post-condition holds of the value after doing an update. The second disjunct says that if e is not a value, then for any possible state σ, if we are given $S(\sigma)$ then we have to show that e is reducible in σ, and for each thing it can reduce to, we have to prove (1) the state interpretation for the new state (σ'), (2) the weakest precondition for the resulting expression (e'), and (3) weakest preconditions for all of the threads it may fork. Along the way, we may open up invariants using a update ${}^{\mathcal{E}}\Rightarrow^{\emptyset}$, so long as we close them at the end via ${}^{\emptyset}\Rightarrow^{\mathcal{E}}$. Moreover, the recursive occurrences of wp in this definition all occur under a later modality $\triangleright$, ensuring that the fixed point indeed exists.

Notice that in the definition above, we have the occurrence of a modality ${}^{\mathcal{E}}\Rightarrow^{\emptyset}$ followed further on by ${}^{\emptyset}\Rightarrow^{\mathcal{E}}$ with an intervening later modality $\triangleright$. It is convenient to introduce notation for the following compound modality called a "step-taking update":

$$\Rightarrow\!\triangleright\!\Rightarrow_{\mathcal{E}} P \triangleq ({}^{\mathcal{E}}\Rightarrow^{\emptyset} \triangleright\, {}^{\emptyset}\Rightarrow^{\mathcal{E}} P)$$

Some specialized rules for this compound modality are shown in Figure 4.2. The first two are simple rules that can be derived from the rules already shown about $\triangleright$ and $\Rightarrow$. The last rule, STEP-FUPD-COMMUTE-PURE, is more complex. It lets us commute universal quantifiers over pure statements around repeated $\Rightarrow\!\triangleright\!\Rightarrow$ modalities, so long as quantification is not over an empty set.

Recall from the previous chapter that there were a number of generic structural rules about weakest precondition that did not make reference to the particular semantics of the language we were considering there. A selection of these rules are reproduced in Figure 4.3. All of these rules can be *derived* from the definition of weakest precondition. The only rule that requires

$$\triangleright P \vdash \Rrightarrow\!\triangleright\!\Rrightarrow_{\mathcal{E}} P \qquad\qquad {}^{\mathcal{E}}\Rrightarrow^{\emptyset\,\emptyset}\Rrightarrow^{\mathcal{E}} P \vdash \Rrightarrow\!\triangleright\!\Rrightarrow_{\mathcal{E}} P$$

STEP-FUPD-COMMUTE-PURE

$$\frac{\{x \mid R(x)\} \text{ inhabited}}{{}^{\emptyset}\Rrightarrow^{\top}(\Rrightarrow\!\triangleright\!\Rrightarrow^{n}_{\top}\forall x.\ulcorner R(x)\urcorner \Rightarrow \ulcorner\phi(x)\urcorner) \dashv\vdash \forall x.\ulcorner R(x)\urcorner \Rightarrow {}^{\emptyset}\Rrightarrow^{\top}(\Rrightarrow\!\triangleright\!\Rrightarrow^{n}_{\top}\ulcorner\phi(x)\urcorner)}$$

Figure 4.2: Rules for $\Rrightarrow\!\triangleright\!\Rrightarrow$ compound modality.

some special treatment is WP-Bind. When we discussed this rule in the context of the previous chapter, the language there had syntactic evaluation contexts that were used to structure evaluation. In the generic case, rather than assume that the language has a syntactic category of evaluation contexts, we say that an injective function $K : \mathsf{Expr} \to \mathsf{Expr}$ is an evaluation context if it has the following properties:

1. For all e, if e is not a value, then neither is $K(e)$.
2. If $e;\sigma \to e';\sigma'$, then $K(e);\sigma \to K(e');\sigma'$.
3. If e_1' is not a value and $K(e_1');\sigma_1 \to e_2;\sigma_2$, then there exists some e_2' such that $K(e_2') = e_2$ and $e_1';\sigma_1 \to e_2';\sigma_2$.

The filling operation for the syntactic evaluation contexts of the ML-like example language has all of these properties.

4.1.3 Lifting Lemmas

The weakest precondition rules shown so far are very "structural" or "generic". Of course, they have to be because we have not really assumed anything interesting about the generic language. After instantiating the logic with a particular language, the user will need to prove additional language specific rules. The following rule (one of several "lifting lemmas" in Iris) provides a way to make it easier to derive such results:

WP-LIFT-STEP

$$\frac{\mathrm{expr\_to\_val}(e_1) = \mathsf{None}}{\begin{array}{l}\forall\sigma_1.\ S(\sigma_1) \mathrel{-\!\!*} {}^{\mathcal{E}}\Rrightarrow^{\emptyset}\Big(\mathrm{red}(e_1,\sigma_1) * \\ \quad \triangleright\forall e_2,\sigma_2,T.\ (e_1,\sigma_1 \to e_2,\sigma_2,T) \mathrel{-\!\!*} {}^{\emptyset}\Rrightarrow^{\mathcal{E}}\Big(S(\sigma_2) * \mathsf{wp}^S_{\mathcal{E}}\, e_2\,\{x.\ P\} * \ast_{e_{\mathrm{f}}\in T}\mathsf{wp}^S\, e_{\mathrm{f}}\,\{\_.\ \mathsf{True}\}\Big)\Big) \\ \vdash \mathsf{wp}^S_{\mathcal{E}}\, e_1\,\{x.\ P\}\end{array}}$$

This rule is complex, but has a somewhat natural reading. It says that if we want to establish a weakest precondition about an expression e_1 that is not a value, then we have to show that for any state σ_1 we could be in, that e_1 can take a step, and for each possible thing it can step to, we have to prove the weakest precondition for the result; moreover, if reducing e_1 happens

WP-Frame
$$\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} * Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P * Q\}$$

WP-Value
$$[v/x]P \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{x.\, P\}$$

Upd-WP
$$^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$$

WP-Upd
$$\mathsf{wp}_{\mathcal{E}}\, e \left\{x.\, {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P\right\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\}$$

WP-Mono
$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{(\forall v.\, \Phi(v) \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}_2} \Psi(v)) * \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, \Phi(x)\} \vdash \mathsf{wp}_{\mathcal{E}_2}\, e\, \{x.\, \Psi(x)\}}$$

Atomic
$$\frac{\mathrm{atomic}(e)}{^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e \left\{x.\, {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{x.\, P\}}$$

Step-Later-Frame
$$\frac{\mathrm{expr\_to\_val}(e) = \mathsf{None}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P\} * \triangleright Q \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, P * Q\}}$$

WP-Bind
$$\frac{K \text{ is a context}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \mathsf{wp}_{\mathcal{E}}\, K(x)\, \{x'.\, P\}\} \dashv\vdash \mathsf{wp}_{\mathcal{E}}\, K(e)\, \{x'.\, P\}}$$

Figure 4.3: Selection of generic weakest precondition rules.

to fork off a list of threads T, we have to establish weakest preconditions for each of them. (If T is an empty list, then the iterated conjunction $\bigstar_{e_f \in T}$ is just True.) In order to establish all these obligations, we are given $S(\sigma_1)$, and are allowed to open up any enabled invariants in $\mathcal{E}$ via the update $^{\mathcal{E}_1}\!\Rrightarrow^{\emptyset}$, but then we are also obliged to close these at the end via $^{\emptyset}\!\Rrightarrow^{\mathcal{E}_1}$ and prove the new state interpretation $S(\sigma_2)$.

4.2 Adequacy

We return to the question of what exactly this definition of weakest precondition guarantees. The answer is summarized in the following adequacy theorem:

Theorem 4.1 (Adequacy)**.** If $[e_1]; \sigma_1 \to^n [e_2] +\!\!+ T; \sigma_2$, and

$$\mathsf{True} \vdash \Rrightarrow_{\top} \exists S : \mathsf{State} \to \mathsf{iProp}.\, S(\sigma) * \mathsf{wp}^{S}_{\top}\, e_1\, \{x.\, \ulcorner \phi(x) \urcorner\}$$

then

1. Every $e \in [e_2] +\!\!+ T$ is either reducible under σ_2 or is a value.
2. If e_2 is a value, then $\phi(e_2)$ holds.

Recall that $\ulcorner \cdot \urcorner$ is the embedding of meta-level propositions, so in the second statement here we mean that $\phi(e_2)$ holds at the meta-level. This means that executing the initial thread e_1 in

the state σ will not lead to any thread going wrong, and that the return value indeed satisfies the postcondition. The reason the postcondition is stated as a pure embedding is that we want to say that it holds independent of any understanding of the semantic model of Iris assertions.

This is only a *partial correctness* guarantee: if the program has a non-terminating execution, then for that execution, the theorem merely ensures it never reaches a stuck state.

This adequacy theorem has been proved several ways throughout the different versions of Iris. The proof I give below follows the framework explained in Jung et al. [66], except that more emphasis is placed on the role of the step-taking update compound modality $\Rightarrow\!\!\triangleright\!\!\Rightarrow_{\mathcal{E}}$, whose importance was observed by Timany et al. [118].

The proof of this theorem is divided into three steps. First, Jung et al. [66] show the "base" part of the logic aside from weakest precondition is sound, in the following sense:

Theorem 4.2 (Soundness of Base Logic)**.** If $\mathsf{True} \vdash \triangleright^n \ulcorner \phi \urcorner$ then ϕ holds.

This proof depends upon the details of how $\triangleright$ and $\ulcorner \phi \urcorner$ are defined in the semantic model, which we will not discuss. Because the semantic model will not change when we extend the program logic to support probabilistic reasoning, we may continue to treat it as a black box.

Using this soundness theorem, one can deduce the following extension, where the $\triangleright$ modality is replaced by iterated step-taking updates:

Theorem 4.3 (Soundness of Step-Taking Updates)**.** If $\mathsf{True} \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_{\top}^{n} \ulcorner \phi \urcorner$ then ϕ holds.

We then prove that from a weakest precondition we can deduce the *embedded* forms of the claims made in the adequacy statement, guarded by iterations of the step-taking update.

Theorem 4.4. If $[e_1] \mathbin{+\!\!+} T_1; \sigma_1 \to^n [e_2] \mathbin{+\!\!+} T_2; \sigma_2$, then

1. For all $e \in [e_2] \mathbin{+\!\!+} T_2$, the following is derivable

$$S(\sigma_1) * \mathsf{wp}_{\top}^{S}\, e_1\, \{x. \ulcorner \phi(x) \urcorner\} * \ast_{e' \in T_1} \mathsf{wp}_{\top}^{S}\, e'\, \{\_.\, \mathsf{True}\} \\ \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_{\top}^{n+1} \ulcorner \mathrm{is\_val}(e) \lor \mathrm{red}(e, \sigma_2) \urcorner$$

2. If e_2 is a value, then

$$S(\sigma_1) * \mathsf{wp}_{\top}^{S}\, e_1\, \{x. \ulcorner \phi(x) \urcorner\} * \ast_{e' \in T_2} \mathsf{wp}_{\top}^{S}\, e'\, \{\_.\, \mathsf{True}\} \\ \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_{\top}^{n+1} \ulcorner \phi(e_2) \urcorner$$

Theorem 4.1 then follows by combining Theorem 4.3 and Theorem 4.4, and using the fact that entailment is transitive.

Proof of Theorem 4.4. The first step is to prove that if $[e_1] \mathbin{+\!\!+} T_1; \sigma_1 \to^n [e_2] \mathbin{+\!\!+} T_2; \sigma_2$, then

$$S(\sigma_1) * \mathsf{wp}_{\top}^{S}\, e_1\, \{x. \ulcorner \phi(x) \urcorner\} * \ast_{e' \in T_2} \mathsf{wp}_{\top}^{S}\, e'\, \{\_.\, \mathsf{True}\} \\ \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_{\top}^{n} \big(S(\sigma_2) * \mathsf{wp}_{\top}^{S}\, e_2\, \{x. \ulcorner \phi(x) \urcorner\} * \ast_{e' \in T_2} \mathsf{wp}_{\top}^{S}\, e'\, \{\_.\, \mathsf{True}\}\big)$$

The proof is by induction on n. The base case is trivial. For the inductive case, we unfold the definition of weakest precondition for whichever thread took the step; because the corresponding thread just took a step, it is not a value, so the right disjunct of wp must hold. Eliminating the $-\!\!*$'s that occur in that disjunct gives us the desired results, however they are under an additional $\Rightarrow\!\!\triangleright\!\!\Rightarrow_\top$ because of the occurence of the updates and laters in wp.

Again by unfolding the definition of weakest precondition, we have that for all $e \in [e_2] +\!\!+ T_2$:

$$S(\sigma_2) * \mathsf{wp}^S_\top\, e_2\, \{x.\ulcorner \phi(x) \urcorner\} * \textstyle\bigstar_{e' \in T_2} \mathsf{wp}^S_\top\, e'\, \{\_.\, \mathsf{True}\}\big) \\ \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_\top \ulcorner \mathrm{is\_val}(e) \vee \mathrm{red}(e, \sigma_2) \urcorner$$

and similarly, if e_2 is a value,

$$S(\sigma_2) * \mathsf{wp}^S_\top\, e_2\, \{x.\ulcorner \phi(x) \urcorner\} * \textstyle\bigstar_{e' \in T_2} \mathsf{wp}^S_\top\, e'\, \{\_.\, \mathsf{True}\}\big) \\ \vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow_\top \ulcorner \phi(e_2) \urcorner$$

Putting these two facts together with the inductive result, we get the conclusion of the theorem. The $n+1$ iterations of $\Rightarrow\!\!\triangleright\!\!\Rightarrow_\top$ in the final result arise from n iterations in the inductive step, plus an additional iteration for the final conclusion. □

4.3 Instantiation

We now return back to the language considered in the previous chapter, describing how the generic framework we have discussed is instantiated to obtain the language specific rules described there. We first need to pick a state interpretation function. We can use a construction similar to the one we used to define fractional permissions in that chapter. Namely, we work with the resource algebra

$$\textsc{Auth}(\textsc{FinMap}(\mathsf{Loc}, \mathrm{Ex}(\mathsf{Val})))$$

In place of maps to pairs of fractions and values, we just need maps to values. We use the exclusive algebra for values since, unlike fractional permissions, one can only have a single points-to assertion for a given location at a time. The structure of this algebra leads to the following derived rules:

HEAP-VALIDITY

$$\boxed{\bullet\, \sigma_1 \cdot \circ\, \sigma_2}^\gamma \vdash \ulcorner \forall l \in \mathrm{dom}(\sigma_2).\ \sigma_1(l) = \sigma_2(l) \urcorner$$

HEAP-ALLOC

$$\frac{l \notin \mathrm{dom}(\sigma)}{\boxed{\bullet\, \sigma}^\gamma \vdash \Rightarrow_{\mathcal{E}} \boxed{\bullet\, \sigma[l := v] \cdot \circ\, \{l := v\}}^\gamma}$$

HEAP-UPDATE

$$\frac{l \in \mathrm{dom}(\sigma_2)}{\boxed{\bullet\, \sigma_1 \cdot \circ\, \sigma_2}^\gamma \vdash \Rightarrow_{\mathcal{E}} \boxed{\bullet\, \sigma_1[l := v] \cdot \circ\, \sigma_2[l := v]}^\gamma}$$

Using this, we define:

$$S^\gamma_{\mathsf{h}}(\sigma) \triangleq \boxed{\bullet\, \sigma}^\gamma \\ l \mapsto^\gamma v \triangleq \boxed{\circ\, \{l \mapsto v\}}^\gamma$$

where γ is some arbitrary ghost name. We will omit writing the annotation γ, with the convention that the same implicit γ is being used throughout proofs. Now the rule HEAP-VALIDITY can be used to ensure that from $l \mapsto v * S_{\mathsf{h}}(\sigma)$ we can deduce that $\sigma(l) = v$. Using the above rules for manipulating the heap resources and the lifting lemma WP-LIFT-STEP, one can derive the "high level" weakest precondition rules from Figure 3.8 in Chapter 3, in a manner similar to the arguments described for the fractional permission encoding. The main difference is that instead of opening up an invariant, the state interpretation S_{h} is what links the resource algebra to the physical state.

Lastly, the following lemma lets us apply the adequacy theorem (Theorem 4.1) directly to a weakest precondition proved with this instantiation of the framework:

Lemma 4.5. If for all γ, $\mathsf{True} \vdash \mathsf{wp}^{S_{\mathsf{h}}^{\gamma}}\, e\, \{x.\ P\}$, then for all σ,

$$\mathsf{True} \vdash \Rrightarrow_{\top} \exists S.\ S(\sigma) * \mathsf{wp}^{S}_{\top}\, e\, \{x.\ P\}$$

Proof. We use RES-ALLOC to obtain $\boxed{\bullet \sigma}^{\gamma}$ for some γ, which is equivalent to $S_{\mathsf{h}}^{\gamma}(\sigma)$. By assumption, $\mathsf{wp}^{S_{\mathsf{h}}^{\gamma}}\, e\, \{x.\ P\}$ holds for this choice of γ, and the result follows. □

Chapter 5

Polaris: Extending Iris with Probabilistic Relational Reasoning

This chapter presents Polaris, an extension of Iris with support for probabilistic relational reasoning. As mentioned already, the Iris semantic model is unchanged during the extension. Only the program logic changes, which is done by altering the definition of weakest precondition. Now that we have seen over the previous two chapters both the specific instantiation of Iris with the ML-like language as well as the general framework, we will explain Polaris by alternating between the general set-up and an instantiation.

5.1 Program Semantics

5.1.1 Probabilistic Transitions

As before, the logic is parameterized by a generic operational semantics, but this time for a probabilistic language. In addition to the assumptions about the language made in §4.1.1, we further assume that there is a function

$$P : \mathsf{Expr} \times \mathsf{State} \times \mathsf{Expr} \times \mathsf{State} \times \mathsf{ThreadPool} \to \mathbb{R}$$

where $P(e_1, \sigma_1, e_2, \sigma_2, T)$ is meant to describe the probability that the transition $e_1; \sigma_1 \to e_2; \sigma, T$ occurs. In other words, if we define $\mathsf{RedTo}(e_1, \sigma_1) = \{(e_2, \sigma_2, T) \mid e_1; \sigma_1 \to e_2; \sigma_2; T\}$, then $P(e_1, \sigma_1, -, -, -)$ will induce a distribution on $\mathsf{RedTo}(e_1, \sigma_1)$. We restrict to discrete distributions by requiring each $\mathsf{RedTo}(e_1, \sigma_1)$ to be countable. To ensure that P indeed describes proper probability distributions, we further require:

1. $P(e_1, \sigma_1, e_2, \sigma_2, T)$ is non-negative.
2. If $\mathsf{RedTo}(e_1, \sigma_1)$ is non-empty, then

$$\sum_{(e_2, \sigma_2, T) \in \mathsf{RedTo}(e_1, \sigma_1)} P(e_1, \sigma_1, e_2, \sigma_2, T) = 1$$

Syntax:

Val	v	:	...
Expr	e	:	...
State	σ	:	...
ThreadPool	T	:	List Expr
Config	ρ	:	$\{T : \text{ThreadPool} \mid T \neq \emptyset\} \times \text{State}$
Trace	χ	:	$\{l : \text{List Config} \mid l \neq \emptyset\}$
Sched	φ	:	$\text{Trace} \to \mathbb{N}$

Per-Thread Reduction: $e; \sigma \to e'; \sigma'; T$
Concurrent Reduction: $\rho \xrightarrow{i} \rho'$

$$\frac{|T_1| = i \qquad e; \sigma \to e'; \sigma'; T_f}{T_1 +\!\!+ [e] +\!\!+ T_2; \sigma \xrightarrow{i} T_1 +\!\!+ [e'] +\!\!+ T_2 +\!\!+ T_f; \sigma'}$$

Trace Semantics: $\chi \to_\varphi \chi'$

$$\frac{\varphi(\chi, \rho) = i \qquad \rho \xrightarrow{i} \rho'}{\chi, \rho \to_\varphi \chi, \rho, \rho'} \qquad \frac{\varphi(\chi, \rho) = i \qquad \neg(\exists \rho'.\, \rho \xrightarrow{i} \rho')}{\chi, \rho \to_\varphi \chi, \rho, \rho}$$

Figure 5.1: Syntax and semantics of generic probabilistic concurrent language.

3. $e_1; \sigma_1 \to e_2; \sigma_2, T$ if and only if $P(e_1, \sigma_1, e_2, \sigma_2, T) > 0$.

The first two requirements ensure that probabilities are non-negative and sum to 1 whenever we consider a non-stuck configuration. The last requirement ensures that a step is said to happen with non-zero probability exactly when the operational semantics permits the step.

As before, this per-thread reduction relation is then lifted to a concurrent transition system, whose rules and syntactic categories are shown in Figure 5.1. Whereas in the previous chapter, concurrent reduction was modeled just with a non-deterministic relation, in order to eventually discuss probabilities of concurrent transitions, we now need to describe how concurrent non-determinism will be resolved by a *scheduler*. Rather than explicitly modeling realistic schedulers, we will represent them as functions over the whole history of the program execution. Of course, a real implementation of a scheduler never inspects the whole execution history, but by conservatively considering this strong class of *adversarial* schedulers, results we prove will also hold for realistic schedulers, as we motivated in Chapter 1.

As before, we define configurations to be pairs of a non-empty thread pool and a state. Then a *trace* χ is a non-empty list of configurations representing a partial execution. We write χ, ρ for the trace which extends χ by appending the single configuration ρ to the end. A scheduler

is a function φ of type Trace $\to \mathbb{N}$. Given such a scheduler, we define a reduction relation $\chi \to_\varphi \chi'$ on traces. If $\varphi(\chi) = i$, then this indicates that the scheduler selects thread i to try to run next after the partial execution χ to obtain χ'. If the scheduler returns a thread number which cannot take a step or which does not exist, the system takes a "stutter" step and the current configuration is repeated again at the end of the trace. We write $\mathsf{curr}(\chi)$ for the last configuration in a trace. We write $\to_\varphi^n$ for the n-ary iteration of this relation, and if $\chi \to_\varphi^n \chi'$ holds then we say that χ reduces to χ' in n steps under φ. A configuration ρ is said to have terminated if the first thread in the pool is a value. We say that χ is terminating in at most n steps under φ if

$$\forall \chi', n'.\, (n' \geq n \wedge \chi \to_\varphi^{n'} \chi') \Rightarrow (\mathsf{curr}(\chi') \text{ has terminated})$$

5.1.2 Indexed Valuation Semantics

We now want to use P and this reduction relation to define a distribution on program executions given an initial state and a scheduler. However, in general, this would require measure theoretic probability to handle properly: even though our language only involves countable single step transition distributions, the set of all executions of a program may be uncountable if the program does not necessarily terminate. However, if we restrict consideration to programs that terminate in a bounded number of steps, we can avoid these issues[1]. Because Iris is a partial correctness logic, restricting consideration to such terminating executions does not lead to much further loss of generality.

With this restriction in place, we can interpret program executions as indexed valuations (as explained in Chapter 2, indexed valuations can be interpreted as probability distributions, and vice versa). Using P, we first define indexed valuations

$$\mathsf{primStep}(e_1, \sigma_1) : M_{\mathsf{I}}(\mathsf{Option}\ (\mathsf{Expr} \times \mathsf{State} \times \mathsf{ThreadPool}))$$

for per-thread steps from $e_1; \sigma_1$. If e_1 is not reducible in state σ_1, then this is the trivial indexed valuation that always returns None. If e_1, σ_1 is reducible, then because $\mathsf{RedTo}(e_1, \sigma_1)$ is countable, we can take the set of indices to be precisely $\mathsf{RedTo}(e_1, \sigma_1)$. Then the decode function d for this indexed valuation is just the function $\lambda x.\, \mathsf{Some}(x)$, and probabilities are assigned to indices with the function $\lambda\, (e_2, \sigma_2, T).\, P(e_1, \sigma_1, e_2, \sigma_2, T)$.

We then lift this to an indexed valuation $\mathsf{cfgStep}(\rho, i) : M_{\mathsf{I}}(\mathsf{Option\ Config})$ which runs primStep on the ith thread in ρ (if it exists), and otherwise just returns None. Using this, we finally define an indexed valuation for the trace step relation:

$$\begin{array}{rcl} \mathsf{tstep}_\varphi(\chi) & \triangleq & \mathsf{match}\ \mathsf{cfgStep}(\mathsf{curr}(\chi), \varphi(\chi))\ \mathsf{with} \\ & & \quad \mid\ \mathsf{Some}\, \rho \Rightarrow \mathsf{ret}\ (\chi, \rho) \\ & & \quad \mid\ \mathsf{None} \Rightarrow \mathsf{ret}\ (\chi, \mathsf{curr}(\chi)) \\ & & \mathsf{end} \end{array}$$

[1] A more denotational alternative, based on an approach due to Kozen [70], is to interpret programs as monotone maps on sub-distributions of states. Then recursive commands are interpreted as least fixed points. However, because the original soundness proof of Iris is given in terms of a language with an operational semantics, I found it more natural to use the semantics described in this section.

This uses the scheduler φ to select which thread to run with cfgStep. As with the relational trace step judgment, this "stutters" if we cannot step the selected thread. For each n, we define the indexed valuation $\mathsf{resStep}^n_\varphi(\chi)$ recursively by:

$$\begin{aligned}\mathsf{resStep}^0_\varphi(\chi) &\triangleq \mathsf{curr}(\chi)\\ \mathsf{resStep}^{n+1}_\varphi(\chi) &\triangleq \chi' \leftarrow \mathsf{tstep}_\varphi(\chi)\,;\ \mathsf{resStep}^n_\varphi(\chi')\end{aligned}$$

This corresponds to stepping the trace n times and returning the final configuration of the resulting trace. We regard the "return value" of a concurrent program to be the value that the first thread evaluates to, so in the event that the program terminates in n steps under the scheduler, the first thread in the configuration returned by $\mathsf{resStep}^n_\varphi(\chi)$ will be this return value. However, when considering an expected value such as $\mathbb{E}_g[\mathsf{resStep}^n_\varphi(\chi)]$ the function g must be of type $\mathsf{Config} \to \mathbb{R}$, so we have to say what real number to assign in the case where the first thread has not terminated. We will do so by assigning an arbitrary value in that case. (Later, the adequacy theorem for this logic will imply that this arbitrary value r does not affect the expected value of programs for which we can prove an appropriate weakest precondition.) Concretely, if $f : \mathsf{Val} \to \mathbb{R}$, we define

$$\mathsf{coerce}_{\mathsf{fun}}(f, r) = \lambda\rho.\ \begin{cases} f(v) & \text{if } \rho = ([v, \dots,], \sigma)\\ r & \text{otherwise}\end{cases}$$

And then we will consider expected values of the form $\mathbb{E}_{\mathsf{coerce}_{\mathsf{fun}}(f,r)}[\mathsf{resStep}^n_\varphi(\chi)]$. Similarly, given a relation $\phi : \mathsf{Val} \times T \to \mathsf{Prop}$ we define a lifting of this relation to configurations as follows:

$$\mathsf{coerce}_{\mathsf{pred}}(\phi) = \lambda(\rho, x).\ \begin{cases} \phi(v, x) & \text{if } \rho = ([v, \dots,], \sigma)\\ \mathsf{False} & \text{otherwise}\end{cases}$$

An alternative to using these coercions would have been to work with a dependently typed version, where the expected value is only defined if one can exhibit a proof that the program terminates in a well formed value within n steps. However, such a definition is less convenient to work with, because then one must constantly manipulate these termination proofs.

5.1.3 Randomization for the ML-Like Language

We now extend the language from Chapter 3 with a command for generating random bits with a specified probability. The syntax of expressions and evaluation contexts is extended to be

$$\begin{aligned} e &::= \cdots \mid \mathsf{flip}(e_1, e_2)\\ K &::= \cdots \mid \mathsf{flip}(K, e) \mid \mathsf{flip}(v, K)\end{aligned}$$

with additional head step reductions

$$\textsc{Flip-True}\quad \frac{0 < \dfrac{z_1}{z_2} \leq 1}{\mathsf{flip}(z_1, z_2); \sigma \to_{\mathsf{h}} \mathsf{true}; \sigma} \qquad \textsc{Flip-False}\quad \frac{0 \leq \dfrac{z_1}{z_2} < 1}{\mathsf{flip}(z_1, z_2); \sigma \to_{\mathsf{h}} \mathsf{false}; \sigma}$$

The idea is that with probability $\frac{z_1}{z_2}$, $\mathsf{flip}(z_1, z_2)$ should return True, and otherwise return False. For $0 \leq \frac{z_1}{z_2} \leq 1$ we therefore define

$$\begin{aligned} \Pr(K[\mathsf{flip}(z_1, z_2)], \sigma, K[\mathsf{true}], \sigma, []) &= \tfrac{z_1}{z_2} \\ \Pr(K[\mathsf{flip}(z_1, z_2)], \sigma, K[\mathsf{false}], \sigma, []) &= 1 - \tfrac{z_1}{z_2} \end{aligned}$$

All of the other possible per-thread reductions are deterministic, so we define the corresponding probabilities of their transitions to be 1.

5.2 Probabilistic Rules

We are now ready to discuss how to extend Iris with probabilistic relational reasoning. Our goal is to be able to use the logic to prove that there exists a suitable coupling between the indexed valuation of a program and some set $\mathcal{I}$ of indexed valuations. The existence of an appropriate coupling will let us use Theorem 2.8, so that we can bound the expected values of the program by bounding the extrema of $\mathcal{I}$.

To motivate the formulation of the extension, let us recall some of the background about probabilistic relational reasoning in the pRHL logic of Barthe et al. that was described in Chapter 1. The idea there, following Benton's Relational Hoare Logic [21], is to replace Hoare triples with Hoare *quadruples*, in which pre and post-conditions become pre and post-relations about *pairs* of programs. Translated to our setting, we would have assertions of the form:

$$\{P\}\, e \;\sim\; \mathcal{I}\, \{x, y.\, Q\}$$

where e is the program we are trying to relate to $\mathcal{I}$, and in the post-relation Q, we would substitute the return value of e in for x and the return value of $\mathcal{I}$ for y. Then, we would adapt the standard Hoare rules to consider the pairs of steps of e and $\mathcal{I}$, along with a special rule for coupling when e makes a randomized choice.

Although the work of Barthe et al. shows that this approach can be useful for reasoning about non-concurrent probabilistic programs, there is an issue with applying it in the concurrent setting: what do we do when e forks a new thread? We would then need to relate $\mathcal{I}$ to multiple program expressions at once.

Instead, we adapt the idea originally developed by Turon et al. [122] for non-probabilistic concurrent relational reasoning which was discussed in Chapter 1. Rather than including the $\mathcal{I}$ as part of a Hoare quadruple, we add a new assertion $\mathsf{Prob}(\mathcal{I})$ to the logic. That is, the specification computation becomes just another kind of "resource" that can be transferred between threads. Because $\mathsf{Prob}(\mathcal{I})$ is just an assertion like any other, we can control access to it between threads by storing it in an invariant. This idea of representing a specification computation as a resource assertion has been used in other separation logics based on Iris [75, 115].

We will then modify the definition of weakest precondition so that when an entailment of the form

$$\mathsf{Prob}(\mathcal{I}) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{x.\, \exists x'.\, \mathsf{Prob}(\mathsf{ret}\ x') * \ulcorner R(x, x') \urcorner\}$$

holds, it will imply the existence of an R-coupling between the concrete execution of e and $\mathcal{I}$. The exact statement and the new definition of weakest precondition will be given in the

next section. We first describe some of the reasoning rules we will obtain with the new form of weakest precondition. As we will see, the idea is that when a thread e owns $\mathsf{Prob}(x \leftarrow \mathcal{I}\,;\ g(x))$ and takes a randomized step, it will then have "permission" to modify $\mathsf{Prob}(\mathcal{I})$ by establishing a coupling between its step and $\mathcal{I}$.

5.2.1 Rules for the ML-Like Language

We start by examining what the resulting probabilistic rules look like for our example ML-like language. Using the same state interpretation as in §4.3, all of the previous weakest precondition rules from Figure 3.8 still hold. In addition, we have two new rules for $\mathsf{flip}(z_1, z_2)$:

FLIP-COUPLE

$$\frac{0 \le \frac{z_1}{z_2} \le 1 \qquad \mathsf{ret\ true} \oplus_{\frac{z_1}{z_2}} \mathsf{ret\ false} \sim \mathcal{I} : R}{\triangleright\, \mathsf{Prob}(x \leftarrow \mathcal{I}\,;\ F(x)) * \triangleright\, (\forall b.\ (\exists x.\ \mathsf{Prob}(F(x)) * \ulcorner R(b,x) \urcorner) \mathrel{-\!\!*} \Phi(b)) \vdash \mathsf{wp}_{\mathcal{E}}^{S_{\mathsf{h}}}\ \mathsf{flip}(z_1, z_2)\ \{\Phi\}}$$

FLIP-IRREL

$$\frac{0 \le \frac{z_1}{z_2} \le 1}{\triangleright\, (\forall b.\ \ulcorner (\frac{z_1}{z_2} = 0 \Rightarrow b = \mathsf{false} \wedge \frac{z_1}{z_2} = 1 \Rightarrow b = \mathsf{true}) \urcorner \mathrel{-\!\!*} \Phi(b)) \vdash \mathsf{wp}_{\mathcal{E}}^{S_{\mathsf{h}}}\ \mathsf{flip}(z_1, z_2)\ \{\Phi\}}$$

These rules are again written in the same continuation-passing style as those from Figure 3.8. The first rule requires owning a monadic computation of the form $x \leftarrow \mathcal{I}\,;\ F(x)$, and we must exhibit a coupling between a random choice between true and false, (weighted by $\frac{z_1}{z_2}$) and the monadic specification $\mathcal{I}$. Then, afterward, we get back the monadic resource, but updated so that it is now of the form $\mathsf{Prob}(F(x))$ for some x. In addition, b (the outcome of the $\mathsf{flip}(z_1, z_2)$ command) and this x are related by R, the postcondition of the coupling we exhibited.

The second rule, FLIP-IRREL, lets us establish a weakest precondition about $\mathsf{flip}(z_1, z_2)$ in the case where we do not want to couple its reduction to the monadic computation. In this case, all we know is that if $\frac{z_1}{z_2}$ is 0, then the return value must be false, and analogously if $\frac{z_1}{z_2} = 1$.

In both rules there is a side-condition requiring that $\frac{z_1}{z_2}$ corresponds to a proper probability, because otherwise the expression gets stuck.

5.2.2 Lifting Lemmas

Now that we have seen rules for $\mathsf{Prob}(\mathcal{I})$ in the specific case of the ML-like language, let us turn to the general setting. Just as Iris had the lifting lemma WP-LIFT-STEP to derive language-specific weakest precondition rules, Polaris has the following "coupling" lifting lemma (differences from

WP-LIFT-STEP are highlighted in blue):

WP-LIFT-COUPLE-STEP

$$\frac{\mathrm{expr\_to\_val}(e_1) = \mathsf{None}}{\begin{array}{l}\triangleright\,\mathsf{Prob}(x \leftarrow \mathcal{I}\,;\ g(x)) * \forall \sigma_1.\ S(\sigma_1) \twoheadrightarrow {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \Big(\mathrm{red}(e_1, \sigma_1) * (\triangleright \ulcorner \mathsf{primStep}(e_1, \sigma_1) \sim \mathcal{I} : R \urcorner) * \\ \quad \triangleright \forall e_2, \sigma_2, T, x.\ \ulcorner (e_1, \sigma_1 \to e_2, \sigma_2, T) \wedge R(\mathsf{Some}(e_2, \sigma_2, T), x) \urcorner * \mathsf{Prob}(g(x)) \\ \quad\quad \twoheadrightarrow {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \Big(S(\sigma_2) * \mathsf{wp}^S_{\mathcal{E}}\, e_2\, \{x.\ P\} * \ast_{e_{\mathrm{f}} \in T} \mathsf{wp}^S\, e_{\mathrm{f}}\, \{\_.\ \mathsf{True}\}\Big)\Big) \\ \vdash \mathsf{wp}^S_{\mathcal{E}}\, e_1\, \{x.\ P\}\end{array}}$$

As with WP-LIFT-STEP, the rule says that to establish a weakest precondition about an expression e_1 which is not a value, we have to show that for any state σ_1, assuming we have the state interpretation $S(\sigma_1)$, that e_1 is reducible in σ_1 and for all e_2, σ_2, T which it can step to, we have to prove the weakest precondition for e_2 and all threads in T. However, we now also start with $\mathsf{Prob}(x \leftarrow \mathcal{I}\,;\ g(x))$ and must exhibit an R-coupling between $\mathsf{primStep}(e_1, \sigma_1)$ and $\mathcal{I}$. Then, in addition to quantifying over the things e_1 can step to, we also quantify over a return value x for $\mathcal{I}$, and get $\mathsf{Prob}(g(x))$, the "updated" form of the probabilistic computation. In addition, we can assume that R relates (e_2, σ_2, T) and x – that is, we're reasoning as if the returned values are linked by the coupling we exhibited.

There are cases where one wants to prove a weakest precondition about an expression whose next step is either not randomized, or if it is randomized, has nothing to do with the coupling we want to establish with the monadic computation (like FLIP-IRREL). In that case, the above rule turns out to be problematic to use: we don't want to always have to own this $\mathsf{Prob}(\mathcal{I})$ assertion just to take a step. Fortunately, the original WP-LIFT-STEP is still sound under the modified definition of weakest precondition, so we can use that instead. This means that if we take a pre-existing instantiation of Iris with some language, and we add some additional probabilistic operations, any non-probabilistic language-specific rules derived in Iris using WP-LIFT-STEP automatically also hold in Polaris.

Moreover, all other generic weakest precondition rules from Iris (some of which were listed in Figure 4.3) continue to hold. The only change is that in the rule WP-BIND, we must assume the following fourth property about the context K:

4. For all $e_1, \sigma_1, e_2, \sigma_2, T$, if e_1 is not a value, then

$$P(e_1, \sigma_1, e_2, \sigma_2, T) = P(K(e_1), \sigma_1, K(e_2), \sigma_2, T)$$

which ensures that the context does not affect transition probabilities.

Finally, we permit weakening ownership of a resource using the following rule:

$$\textsc{ProbLe}\ \frac{\mathcal{I}' \subseteq_{\mathrm{p}} \mathcal{I}}{\mathsf{Prob}(\mathcal{I}) \vdash \mathsf{Prob}(\mathcal{I}')}$$

We can use this to manipulate the $\mathcal{I}$ into a form required by something like WP-LIFT-COUPLE-STEP. Because $\mathcal{I}' \equiv \mathcal{I}$ implies $\mathcal{I}' \subseteq_{\mathrm{p}} \mathcal{I}$, it follows that if $\mathcal{I}' \equiv \mathcal{I}$ then $\mathsf{Prob}(\mathcal{I}) \Leftrightarrow \mathsf{Prob}(\mathcal{I}')$.

5.3 Adequacy

In order to discuss the adequacy result for this extension, we first define $\mathsf{Prob}(\mathcal{I})$ in terms of the other connectives of Iris. The approach is similar to how the state interpretation and $l \mapsto v$ assertions were defined for the language in §3. Namely, if $\mathcal{I}$ has type $M_{\mathsf{NI}}(T)$, we use the RA[2] $\textsc{Auth}(\textsc{Opt}(\mathrm{Ex}(M_{\mathsf{NI}}(T))))$ so as to be able to represent an "authoritative" version of the monadic computation along with a "fragment" that is used in the definition of Prob:

$$\mathsf{Prob}^{\gamma}(\mathcal{I}) \triangleq \exists \mathcal{I}'. \ulcorner \mathcal{I} \subseteq_{\mathsf{p}} \mathcal{I}' \urcorner * \boxed{\circ\, \mathsf{ex}(\mathcal{I}')}^{\gamma}$$

The fully formal definition here is parameterized by a ghost name γ. But, as before when the same technicality arose with the definition of $l \mapsto v$ in §4.3, we can leave this name implicit as long as we work with the convention that throughout a proof we quantify over some common name.

The authoritative RA instantiated with $\textsc{Opt}(\mathrm{Ex}(M_{\mathsf{NI}}(T)))$ supports the following derived rules:

PROB-RES-VALIDITY
$$\boxed{\bullet\, \mathsf{ex}(\mathcal{I}_1) \cdot \circ\, \mathsf{ex}(\mathcal{I}_2)}^{\gamma} \vdash \ulcorner \mathcal{I}_1 \equiv \mathcal{I}_2 \urcorner$$

PROB-RES-UPDATE
$$\boxed{\bullet\, \mathsf{ex}(\mathcal{I}) \cdot \circ\, \mathsf{ex}(\mathcal{I})}^{\gamma} \vdash \Rightarrow_{\mathcal{E}} \boxed{\bullet\, \mathsf{ex}(\mathcal{I}') \cdot \circ\, \mathsf{ex}(\mathcal{I}')}^{\gamma}$$

PROB-RES-INIT
$$\mathsf{True} \vdash \exists \gamma. \Rightarrow_{\mathcal{E}} \boxed{\bullet\, \mathsf{ex}(\mathcal{I}) \cdot \circ\, \mathsf{ex}(\mathcal{I})}^{\gamma}$$

Essentially, because of the way that the exclusive RA works, we can freely change the represented monadic computation so long as we have both the authoritative and fragment resources.

The following adequacy theorem for the logic will guarantee that if we prove an appropriate weakest precondition involving $\mathsf{Prob}(\mathcal{I})$, then a certain coupling exists between the concrete program and $\mathcal{I}$:

Theorem 5.1. For all $\mathcal{I} : M_{\mathsf{NI}}(T)$ and $\phi : \mathsf{Val} \times T \to \mathsf{Prop}$, if

$$\mathsf{True} \vdash \Rightarrow_{\top} \exists S.\, \forall \gamma.\, S(\sigma) * (\mathsf{Prob}^{\gamma}(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^{S}_{\top}\, e_1\, \{x.\, \exists x'.\, \mathsf{Prob}^{\gamma}(\mathsf{ret}\ x') * \ulcorner \phi(x, x') \urcorner\})$$

holds, and $([e_1], \sigma_1)$ terminates in at most n steps under the scheduler φ, then there exists a $\mathsf{coerce}_{\mathsf{pred}}(\phi)$-coupling between $\mathsf{resStep}^{n}_{\varphi}([e_1], \sigma_1)$ and $\mathcal{I}$.

The following corollary lets us transfer bounds on the extrema of a monadic model to a concrete program:

Corollary 5.2. Under the same assumptions and notation as Theorem 5.1, suppose $f : \mathsf{Val} \to \mathbb{R}$ and $g : T \to \mathbb{R}$ are functions such that g is bounded on the support of $\mathcal{I}$. If for all x and x', $\phi(x, x')$ implies $f(x) = g(x')$, then for all r, $\mathbb{E}_{\mathsf{coerce}_{\mathsf{fun}}(f,r)}[\mathsf{resStep}^{n}_{\varphi}([e_1], \sigma_1)]$ exists and

$$\mathbb{E}^{\mathsf{min}}_{g}[\mathcal{I}] \leq \mathbb{E}_{\mathsf{coerce}_{\mathsf{fun}}(f,r)}[\mathsf{resStep}^{n}_{\varphi}([e_1], \sigma_1)] \leq \mathbb{E}^{\mathsf{max}}_{g}[\mathcal{I}]$$

[2] In the machine checked proofs, we instead use a version that does not fix the type T of the represented monadic computation ahead of time. Specifically, we use the RA $\textsc{Auth}(\textsc{Opt}(\mathrm{Ex}(\Sigma_{T:\mathsf{Type}} M_{\mathsf{NI}}(T))))$, so that we work with dependent pairs of a type T and a monadic computation of that type. However, within a given proof one will only use a particular type T, so we do not describe the dependently typed version here.

Proof. By Theorem 5.1, there is a $\mathsf{coerce}_{\mathsf{pred}}(\phi)$-coupling between $\mathsf{resStep}^n_\varphi([e_1], \sigma_1)$ and $\mathcal{I}$. We next show that for all ρ and x, if $\mathsf{coerce}_{\mathsf{pred}}(\phi)(\rho, x)$ holds, then $\mathsf{coerce}_{\mathsf{fun}}(f, r)(\rho) = g(x)$. From the definition of $\mathsf{coerce}_{\mathsf{pred}}(\phi)$ there are two cases. In the first, ρ is of the form $([v, \ldots,], \sigma)$ for some v and σ and $\phi(v, x)$ holds, so by assumption $f(v) = g(x)$ and $\mathsf{coerce}_{\mathsf{fun}}(f, r)(\rho) = g(x)$. In the second case, ρ is not of this form, but then $\mathsf{coerce}_{\mathsf{pred}}(\phi)(\rho, x)$ is False by definition, so the conclusion follows immediately.

Applying Conseq, we obtain a $(\lambda \rho, x.\, \mathsf{coerce}_{\mathsf{fun}}(f, r)(\rho) = g(x))$-coupling. Theorem 2.8 then gives the desired conclusion.

□

Finally, we have an analogue of Theorem 4.1:

Theorem 5.3. If $[e_1]; \sigma_1 \to^n [e_2] +\!\!+ T; \sigma_2$ and $\mathsf{True} \vdash \Rrightarrow_\top \exists S.\, \forall \gamma.\, S(\sigma) * (\mathsf{Prob}^\gamma(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^S_\top\, e_1\, \{x.\, \ulcorner \phi(x) \urcorner\})$, then:

1. Every $e \in [e_2] +\!\!+ T$ is either reducible under σ_2 or is a value.
2. If e_2 is a value, then $\phi(e_2)$ holds.

Like Iris, Polaris is a partial correctness logic. Note that Theorem 5.1 and Corollary 5.2 only hold for schedulers under which the program is guaranteed to terminate in some number of steps.

The modified definition of weakest precondition for which these results hold is (differences highlighted in blue):

$$
\begin{aligned}
&\mathsf{wp}^S \triangleq \mu\, \mathsf{wp}.\, \lambda\, \mathcal{E}, e, \Phi. \\
&\quad (\exists v.\, \mathrm{expr\_to\_val}(e) = \mathsf{Some}\ v \wedge \Rrightarrow_{\mathcal{E}} \Phi(v)) \vee \\
&\quad \Big(\mathrm{expr\_to\_val}(e) = \mathsf{None} \wedge \forall \sigma, \mathcal{I}.\, (S(\sigma) * \boxed{\bullet\, \mathsf{ex}(\mathcal{I})}^{\gamma}) \mathrel{-\!\!*} \\
&\qquad {}^{\mathcal{E}}\Rrightarrow^{\emptyset} \Big(\ulcorner \mathrm{red}(e, \sigma) \urcorner * \triangleright\, \exists R, \mathcal{I}', F. \\
&\qquad\quad \ulcorner (x \leftarrow \mathcal{I}';\ F(x) \subseteq_{\mathsf{p}} \mathcal{I}) \wedge \mathsf{primStep}(e, \sigma) \sim \mathcal{I}' : R \urcorner \wedge \forall e', \sigma', T, x. \\
&\qquad\quad \ulcorner (e, \sigma \to e', \sigma', T) \wedge R(\mathsf{Some}(e_2, \sigma_2, T), x) \urcorner \mathrel{-\!\!*} \\
&\qquad\quad {}^{\emptyset}\Rrightarrow^{\mathcal{E}} \Big(S(\sigma') * \boxed{\bullet\, \mathsf{ex}(F(x))}^{\gamma} * \mathsf{wp}(\mathcal{E}, e', \Phi) * \\
&\qquad\qquad \textstyle\bigstar_{e'' \in T} \mathsf{wp}(\top, e'', \lambda\, \_.\, \mathsf{True})\Big)\Big)\Big)
\end{aligned}
$$

Compared to the original definition of weakest precondition, this says that in the case where e is not a value, we get not only the state interpretation $S(\sigma)$ for some state, but also the authoritative resource version $\boxed{\bullet\, \mathsf{ex}(\mathcal{I})}^{\gamma}$ of some monadic computation. From there, we have to show that $\mathcal{I}$ is greater than or equal to $x \leftarrow \mathcal{I}';\ F(x)$ under the $\subseteq_{\mathsf{p}}$ ordering for some $\mathcal{I}'$ and F, and must exhibit an R-coupling between the reduction of e in state σ and $\mathcal{I}'$. Then, as in the lifting lemma, in addition to quantifying over what e can reduce to, we also quantify over values x that can be returned by $\mathcal{I}'$, and can assume that the reducts of e and x are related by R. Besides establishing the new state interpretation and weakest preconditions for the results

of stepping e, we also have to update the authoritative copy of the monadic computation to $\boxed{\bullet\, \mathsf{ex}(F(x))}^{\gamma}$.

The proof of Theorem 5.3 is very similar to that of Theorem 4.1, so we will not discuss it. For the proof of Theorem 5.1, we we start with the following result, which plays a role similar to that of Theorem 4.4. Namely, we will show that if an appropriate weakest precondition holds, then this entails the existence of the coupling guarded by iterated step-taking update modalities:

Theorem 5.4. Let φ be a scheduler and χ be some partial trace. If $\chi, ([e_1] +\!\!+ T, \sigma)$ terminates in at most n steps under φ, then

$$S(\sigma) * \boxed{\bullet\, \mathsf{ex}(\mathcal{I})}^{\gamma} * \mathsf{wp}^S_{\top}\, e_1\, \{x.\, \exists x'.\, \mathsf{Prob}^{\gamma}(\mathsf{ret}\ x') * \ulcorner \phi(x, x') \urcorner\} * \ast_{e' \in T} \mathsf{wp}^S_{\top}\, e'\, \{\_.\, \mathsf{True}\}$$
$$\vdash \Rightarrow\!\!\triangleright\!\!\Rightarrow^{n+1}_{\top} \ulcorner \mathsf{resStep}^n_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma)) \sim \mathcal{I} : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

Proof. Before delving into the details, let us sketch the idea at a high level. We have re-defined weakest precondition so that if it holds, then for each step of the concrete program there is a coupling between that step and the current "head" of the monadic computation $\mathcal{I}$. All we need to do then is to combine these step-by-step couplings together to get a coupling between the whole concrete program and $\mathcal{I}$. The way we combine them together is via the rule Bind. Of course, as we unfold the definition of weakest precondition, the existence proofs for the step-by-step couplings are under successively more iterations of the update and later modalities, which is why the iterated step-taking update modality appears in the statement of the theorem.

Now, the formal proof is by induction on n. In the base case, e_1 must be a value, (by the assumption about termination) and $\mathsf{resStep}^0_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma)) = \mathsf{ret}\ ([e_1] +\!\!+ T, \sigma)$. Because e_1 is a value, we must be in the setting of the left disjunct of wp, so the post-condition holds under an update. Hence there exists some x' such that $\mathsf{ret}\ x' \subseteq_{\mathsf{p}} \mathcal{I}$ and $\phi(e_1, x')$. This means we have $\ulcorner \mathsf{coerce}_{\mathsf{pred}}(\phi)(([e_1] +\!\!+ T, \sigma), x') \urcorner$ under the update modality. We eliminate the modality, and then the existence of the coupling follows from Ret from §2.5.

For the inductive case where $n > 0$, set $i = \varphi(\chi, ([e_1] +\!\!+ T, \sigma))$. Then there are three subcases, either: (1) i is greater than the length of $[e_1] +\!\!+ T$, (2) the ith thread in $[e_1] +\!\!+ T$ is a value, or (3) the ith thread is a non-value. For the first two of these three, the trace semantics takes a stutter step. We have $\mathsf{cfgStep}(([e_1] +\!\!+ T, \sigma), i) \equiv \mathsf{ret}\ \mathsf{None}$ and hence

$$\mathsf{resStep}^n_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma)) \equiv \mathsf{resStep}^{n-1}_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma), ([e_1] +\!\!+ T, \sigma)) \tag{5.1}$$

Then $\chi, ([e_1] +\!\!+ T, \sigma), ([e_1] +\!\!+ T, \sigma)$ terminates in at most $n-1$ steps under φ, so the induction hypothesis applied to this extended trace gives us

$$\Rightarrow\!\!\triangleright\!\!\Rightarrow^{n}_{\top} \ulcorner \mathsf{resStep}^{n-1}_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma), ([e_1] +\!\!+ T, \sigma)) \sim \mathcal{I} : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

We may use the equivalence (5.1) to obtain:

$$\Rightarrow\!\!\triangleright\!\!\Rightarrow^{n}_{\top} \ulcorner \mathsf{resStep}^{n}_{\varphi}(\chi, ([e_1] +\!\!+ T, \sigma)) \sim \mathcal{I} : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

Finally, we can weaken by an additional iteration of $\Rightarrow\!\!\triangleright\!\!\Rightarrow_{\top}$.

For the third case, let e_i be the ith thread in $[e_1] +\!\!+ T$, so that there exists lists T_l and T_r such that $[e_1] +\!\!+ T = T_l +\!\!+ [e_i] +\!\!+ T_r$. Set $\rho = (T_l +\!\!+ [e_i] +\!\!+ T_r, \sigma)$. We start by unfolding the

weakest precondition for e_i. Because this thread is not a value, the right disjunct must hold. Eliminating the first $-\!\!*$ that occurs in that disjunct, in addition to the corresponding update and later modalities, gives us that there exists $\mathcal{I}'$, F, and R such that

$$(x \leftarrow \mathcal{I}' ;\ F(x)) \subseteq_{\mathsf{p}} \mathcal{I} \tag{5.2}$$

and

$$\mathsf{primStep}(e_i, \sigma) \sim \mathcal{I}' : R \tag{5.3}$$

Rewriting by (5.2) and unfolding the definition of resStep, it suffices to prove

$$^{\mathcal{E}}\!\Vdash^{\emptyset} \Vdash\!\!\Rightarrow\!\!\Rightarrow^{n}_{\top} \ulcorner \big(x \leftarrow \mathsf{primStep}(e_i, \sigma) ;\ G(x)\big) \sim \big(x \leftarrow \mathcal{I}' ;\ F(x)\big) : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner \tag{5.4}$$

where

$$\begin{array}{rcl} G & \triangleq & \lambda x.\, \mathsf{match}\ x\ \mathsf{with} \\ & & \quad \mid\ \mathsf{Some}\,(e_i', \sigma', T') \Rightarrow \mathsf{resStep}^{n-1}_{\varphi}(\chi, \rho, (T_l +\!\!+ [e_i'] +\!\!+ T_r +\!\!+ T', \sigma')) \\ & & \quad \mid\ \mathsf{None} \Rightarrow \mathsf{resStep}^{n-1}_{\varphi}(\chi, \rho, \rho) \\ & & \mathsf{end} \end{array}$$

Note that because we eliminated the modalities in the weakest precondition, the outermost $\Vdash\!\!\Rightarrow\!\!\Rightarrow_{\top}$ in what we are trying to prove has become just a $^{\mathcal{E}}\!\Vdash^{\emptyset}$ in (5.4).

Define

$$R' \triangleq \lambda x, y.\ R(x, y) \wedge \exists e_i', \sigma_1', T'.\ x = \mathsf{Some}((e_i', \sigma_1', T')) \wedge e_i; \sigma_1 \to e_i'; \sigma_1'; T'$$

We can strengthen the coupling in (5.3) to obtain $\mathsf{primStep}(e_i, \sigma_1) \sim \mathcal{I}' : R'$ because e_i is reducible in σ_i, so the support of $\mathsf{primStep}(e_i, \sigma_1)$ only contains things which e_i can step to in state σ_1. Applying Bind with this strengthened coupling, it suffices to show:

$$^{\mathcal{E}}\!\Vdash^{\emptyset} \Vdash\!\!\Rightarrow\!\!\Rightarrow^{n}_{\top} \ulcorner \forall x, y.\ R'(x, y) \Rightarrow G(x) \sim F(y) : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

The set $\{(x, y) \mid R'(x, y)\}$ is inhabited. To see this, recall that for any non-deterministic R'-coupling to hold, there must be an underlying R'-coupling between two indexed valuations. And, an R'-coupling between two indexed valuations is itself an indexed valuation for which R' must hold on any elements of its support. Finally, the support of an indexed valuation must be non-empty because the probabilities of all indices sum to 1, so at least one index must occur with non-zero probability.

Thus, we can use step-fupd-commute-pure and so it suffices to prove

$$\forall x, y. \ulcorner R'(x, y) \urcorner \Rightarrow {}^{\mathcal{E}}\!\Vdash^{\emptyset} \Vdash\!\!\Rightarrow\!\!\Rightarrow^{n}_{\top} \ulcorner G(x) \sim F(y) : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

Introducing these quantifier and the pure assertion R', we have that $x = \mathsf{Some}((e_i', \sigma', T'))$ and $e_i; \sigma \to e_i'; \sigma'; T'$, and $R(x, y)$ holds. Simplifying the definition of $G(x)$, it suffices to show:

$$^{\mathcal{E}}\!\Vdash^{\emptyset} \Vdash\!\!\Rightarrow\!\!\Rightarrow^{n}_{\top} \ulcorner \mathsf{resStep}^{n-1}_{\varphi}(\chi, \rho, (T_l +\!\!+ [e_i'] +\!\!+ T_r +\!\!+ T', \sigma')) \sim F(y) : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

We can now eliminate the second wand and update modality in the definition of weakest precondition for e_i. We obtain $S(\sigma')$, $\boxed{\bullet\, \mathsf{ex}(F(y))}^{\gamma}$, and the weakest precondition for e_i'. Moreover, the extended trace $\chi, \rho, (T_l +\!\!+ [e_i'] +\!\!+ T_r +\!\!+ T', \sigma')$ terminates in at most $n-1$ steps under φ. We can thus apply the induction hypothesis to this extended trace, and the result follows. □

Using this, the main coupling result follows straightforwardly:

Proof of Theorem 5.1. Using Theorem 4.3, it suffices to show

$$\mathsf{True} \vdash \Rightarrow\!\!\!\triangleright\!\!\!\Rightarrow^{n+2}_{\top} \ulcorner \mathsf{resStep}^n_{\varphi}(([e_1], \sigma_1)) \sim \mathcal{I} : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner$$

By assumption, we have:

$$\mathsf{True} \vdash \Rightarrow_{\top} \exists S.\, \forall \gamma.\, S(\sigma) * (\mathsf{Prob}^{\gamma}(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^{S}_{\top}\, e_1\, \{x.\, \exists x'.\, \mathsf{Prob}^{\gamma}(\mathsf{ret}\ x') * \ulcorner \phi(x, x') \urcorner\}) \tag{5.5}$$

So from transitivity of entailment, we just have to show that

$$\begin{aligned} \Rightarrow_{\top} \exists S.\, \forall \gamma.\, S(\sigma) * (\mathsf{Prob}^{\gamma}(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^{S}_{\top}\, e_1\, \{x.\, \exists x'.\, \mathsf{Prob}^{\gamma}(\mathsf{ret}\ x') * \ulcorner \phi(x, x') \urcorner\} \\ \vdash \Rightarrow\!\!\!\triangleright\!\!\!\Rightarrow^{n+2}_{\top} \ulcorner \mathsf{resStep}^n_{\varphi}(([e_1], \sigma_1)) \sim \mathcal{I} : \mathsf{coerce}_{\mathsf{pred}}(\phi) \urcorner \end{aligned}$$

We use PROB-RES-INIT to obtain $\boxed{\bullet\, \mathsf{ex}(\mathcal{I})}^{\gamma} * \boxed{\circ\, \mathsf{ex}(\mathcal{I})}^{\gamma}$ for some γ. From $\boxed{\circ\, \mathsf{ex}(\mathcal{I})}^{\gamma}$ we get $\mathsf{Prob}^{\gamma}(\mathcal{I})$. We eliminate the modality and existential on the left side of the entailment. Instantiating the universal quantifier on the left side with γ, we then have $S(\sigma)$ for some S and

$$\mathsf{Prob}^{\gamma}(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^{S}_{\top}\, e_1\, \{x.\, \exists x'.\, \mathsf{Prob}^{\gamma}(\mathsf{ret}\ x') * \ulcorner \phi(x, x') \urcorner\}$$

After giving up $\mathsf{Prob}^{\gamma}(\mathcal{I})$ to eliminate this implication, we have what is needed to apply Theorem 5.4, from which the conclusion follows. □

In light of the definition of wp, the justification of WP-LIFT-COUPLE-STEP is straightforward, because it is essentially the right disjunct of the weakest precondition. Unfolding the definition of Prob and using PROB-RES-VALIDITY, ownership of Prob lets us conclude that the authoritative copy of the monadic resource must be greater than or equal to the $x \leftarrow \mathcal{I}\,;\ g(x)$ in the statement of the rule.

How do we show that WP-LIFT-STEP still holds, as claimed above? Recall that this rule is:

$$\frac{\mathrm{expr\_to\_val}(e_1) = \mathsf{None}}{\begin{array}{l} \forall \sigma_1.\, S(\sigma_1) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\emptyset} \Big(\mathrm{red}(e_1, \sigma_1) * \\ \quad \triangleright \forall e_2, \sigma_2, T.\, (e_1, \sigma_1 \to e_2, \sigma_2, T) \mathrel{-\!\!*} {}^{\emptyset}\!\Rightarrow^{\mathcal{E}} \Big(S(\sigma_2) * \mathsf{wp}^{S}_{\mathcal{E}}\, e_2\, \{x.\, P\} * \mathop{\Large *}_{e_{\mathrm{f}} \in T} \mathsf{wp}^{S}\, e_{\mathrm{f}}\, \{\_.\, \mathsf{True}\}\Big)\Big) \\ \vdash \mathsf{wp}^{S}_{\mathcal{E}}\, e_1\, \{x.\, P\} \end{array}}$$

So, the assumptions here let us discharge the non-probabilistic parts of the right disjunct of wp, but not the new parts that require us to exhibit a coupling. Fortunately, when we have

$\boxed{\bullet\, \mathsf{ex}(\mathcal{I})}^{\gamma}$ in the course of proving wp, we can always rewrite this as $\boxed{\bullet\, \mathsf{ex}(x \leftarrow \mathsf{ret}\ ();\ \mathcal{I})}^{\gamma}$, and then use Trivial to exhibit a coupling between the concrete reduction and ret $()$.

Finally, we have the following analogue of Theorem 4.5, which lets us more easily use the probabilistic adequacy theorems above for the instantiation of the framework with the ML-like language:

Lemma 5.5. If for all γ_h and γ_p, $\mathsf{Prob}^{\gamma_p}(\mathcal{I}) \vdash \mathsf{wp}^{S_{\mathsf{h}}^{\gamma_h}}\ e\ \{x.\ P\}$ then for all σ,

$$\mathsf{True} \vdash \Rightarrow\!\!\!\!\Rightarrow_{\top} \exists S.\ \forall \gamma.\ S(\sigma) * (\mathsf{Prob}^{\gamma}(\mathcal{I}) \mathrel{-\!\!*} \mathsf{wp}^{S}_{\top}\ e\ \{x.\ P\})$$

As before, the statement requires quantifying over the ghost names γ_h and γ_p. In the next chapter we will suppress these annotations from our assertions, with the convention that they are implicitly quantified over.

Note that Corollary 5.2 gives us bounds on expected values of results returned by programs. However, recall from §1.2.1 that we can always instrument a program in order to record the number of steps taken by the program as a return value. In this way, we can use Corollary 5.2 to reason about complexity properties as well.

Chapter 6

Examples

In this chapter, we apply the program logic from the previous chapter to two of the introductory examples described in §1.1: approximate counters and concurrent skip lists. Besides demonstrating that Polaris can be used to verify interesting concurrent randomized algorithms, these examples show how the program logic is used with the equational and quantitative rules for the monad described in Chapter 2. In each example, we follow the same pattern. First, we formulate a monadic model of the data structure's behavior. Next, we use the program logic to derive rules for establishing weakest preconditions for concrete programs that use these data structures. We call these the "specifications". Although the two algorithms are quite different, the proofs of these specifications follow a similar overall structure, which will become apparent when the skip list example is described. Finally, by applying Corollary 5.2, the resulting derivations reduce the analysis of the probabilistic behavior of a concrete program to that of the monadic model.

6.1 Approximate Counter

We start by establishing results about weakest preconditions that relate the unbiased approximate counter algorithm from Figure 1.1c to the monadic computation approxN from Figure 2.2. The concrete code and monadic version are reproduced in Figure 6.1.

6.1.1 Specification and Example Client

The specification rules we have proved about this data structure are given in Figure 6.2. The rules use a predicate $\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q, n)$, which can be treated by a user as an abstract predicate representing the permission to perform n increments to the counter at l. The parameter q is a fractional permission [25] that we use to track how many threads can access the counter. (Ignore the names γ_l, γ_p, and γ_c – we will describe how they are used when we give the definition of ACounter later). The rule ACounterNew says that we can create a new counter by allocating a reference cell containing 0. It takes the monadic specification $\mathsf{Prob}(\mathsf{approxN}\ n\ 0)$ as a precondition, and returns the full ACounter permission for n increments. The rule ACounterSep lets us split or join this ACounter permission into pieces. If we have permission to perform at least

```
incr l ≜
  let k = min(!l, MAX) in
  let b = flip(1/(k+1)) in
  if b then (FAA(l, k+1); ())
  else ()

read l ≜ !l
```

```
approxIncr ≜
  k ← ret 0 ∪ ··· ∪ ret MAX;
  ret (k+1) ⊕_{1/(k+1)} ret 0

approxN 0 z ≜ ret z
approxN (n+1) z ≜
  k ← approxIncr;
  approxN n (z+k)
```

Figure 6.1: Approximate counter code and monadic model.

ACounterNew
$\mathsf{Prob}(\mathsf{approxN}\ n\ 0) \vdash \mathsf{wp}\ \mathsf{ref}\ 0\ \{l.\ \exists \gamma_\mathsf{l}, \gamma_\mathsf{p}, \gamma_\mathsf{c}, \mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, 1, n)\}$

ACounterSep
$\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q+q', n+n') \dashv\vdash \mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q, n) * \mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q', n')$

ACounterIncr
$\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q, n+1) \vdash \mathsf{wp}\ \mathsf{incr}\ l\ \{\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q, n)\}$

ACounterRead
$\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, 1, 0) \vdash \mathsf{wp}\ \mathsf{read}\ l\ \{v.\ \exists n.\ \mathsf{Prob}(\mathsf{ret}\ n) \land v = n\}$

Figure 6.2: Specification for approximate counters.

$$\mathsf{countTrue}\ c\ lb \triangleq \mathsf{foldLeft}\ (\lambda\ \_\ b.\ \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{incr}\ c)\ \mathsf{else}\ ())\ lb\ ()$$

$$\begin{array}{c}
\{\mathsf{Prob}(\mathsf{approxN}\ (|lb_1|_t + |lb_2|_t)\ 0)\} \\
\mathsf{let}\ c = \mathsf{ref}\ 0\ \mathsf{in} \\
\{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1, |lb_1|_t + |lb_2|_t)\} \\
\begin{array}{c|c}
\{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1/2, |lb_1|_t)\} & \{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1/2, |lb_2|_t)\} \\
\mathsf{countTrue}\ c\ lb_1 & \mathsf{countTrue}\ c\ lb_2 \\
\{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1/2, 0)\} & \{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1/2, 0)\}
\end{array} \\
\{\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(c, 1, 0)\} \\
\mathsf{read}\ c \\
\{v.\ \exists n.\ \mathsf{Prob}(\mathsf{ret}\ n) \land v = n\}
\end{array}$$

Figure 6.3: Example client using approximate counters.

one increment, we can use ACounterIncr, which gives us back ACounter with permission to do one fewer increment. Finally, if we have ACounter with the full fractional permission 1, and there are 0 pending increments, we can use ACounterRead. In the post condition we get back $\mathsf{Prob}(\mathsf{ret}\ n)$ for some n which is equal to the value v that the call to read returns. Note that when we write something like $v = n$, we are omitting the implicit conversion from numbers to the type Val.

At first this specification seems weak, but this is exactly what we will need in order to use Corollary 5.2. To see how we can use these rules to reason about a client program that uses the approximate counter, consider the example client in Figure 6.3, which is annotated with a Hoare-triple proof outline (recall that Hoare triples can be encoded as weakest preconditions). We start with a helper function countTrue, which takes an approximate counter c and a list of booleans lb, and counts the number of times true occurs in lb using the counter. The client begins by creating a new counter c. It then runs two threads in parallel that run countTrue on two lists lb_1 and lb_2, using the shared counter c – we denote this parallel composition using $\|$. The parent blocks until both threads finish[1] and then reads from the counter[2].

Refer to this client code as e. If we write $|lb|_t$ for the function giving the number of times True occurs in lb, then we would like to show that in expectation, e returns $|lb_1|_t + |lb_2|_t$. The

[1]This parallel composition operator is implemented in terms of the primitive fork$\{e\}$ operation. Pre-existing derived rules for parallel composition were already formalized in the Iris Coq development, and the same proofs work unchanged with Polaris.

[2]Of course, here the threads may as well maintain their own exact counters and combine them at the end. But in a real application such as [121], there are tens of millions of counters and hundreds of threads, so having each thread maintain its own set of counters would be expensive.

CountGeq
$$\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\,(q, n')}^{\gamma} \vdash \ulcorner n \geq n' \urcorner$$

CountEq
$$\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\,(1, n')}^{\gamma} \vdash \ulcorner n = n' \urcorner$$

CountPerm
$$\boxed{\circ\,(q, n)}^{\gamma} * \boxed{\circ\,(q', n')}^{\gamma} \vdash \ulcorner q + q' \leq 1 \urcorner$$

CountSep
$$\boxed{\circ\,(q, n)}^{\gamma} * \boxed{\circ\,(q', n')}^{\gamma} \dashv\vdash \boxed{\circ\,(q + q', n + n')}^{\gamma}$$

CountAlloc
$$\mathsf{True} \vdash \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \boxed{\bullet\, n}^{\gamma} * \boxed{\circ\,(1, n)}^{\gamma}$$

CountUpd
$$\boxed{\bullet\,(n + k)}^{\gamma} * \boxed{\circ\,(q, n)}^{\gamma} \vdash \Rrightarrow_{\mathcal{E}} \boxed{\bullet\,(n' + k)}^{\gamma} * \boxed{\circ\,(q, n')}^{\gamma}$$

Figure 6.4: Counter resource rules.

outlined derivation in Figure 6.2 shows that

$$\mathsf{Prob}(\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0) \vdash \mathsf{wp}\ e\ \{v.\, \exists n.\, \mathsf{Prob}(\mathsf{ret}\ n) \wedge v = n\}$$

holds. Let $f : \mathsf{Val} \to \mathbb{R}$ be defined by:

$$f(v) = \begin{cases} z & v \text{ is equal to the integer } z \\ 0 & v \text{ is a non-integer value} \end{cases}$$

That is, it coerces integer values to the corresponding real number, and sends all other values to 0. Then the above implies:

$$\mathsf{Prob}(\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0) \vdash \mathsf{wp}\ e\ \{v.\, \exists n.\, \mathsf{Prob}(\mathsf{ret}\ n) \wedge f(v) = n\}$$

In addition, it is not hard to show that for each k, there is an upper bound on the value returned by approxN k 0, so by Corollary 5.2 and Lemma 5.5 we have:

$$\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0] \leq \mathbb{E}_{\mathsf{coerce}_{\mathsf{fun}}(f,0)}[\mathsf{resStep}^n_{\varphi}([e], \sigma)] \leq \mathbb{E}^{\mathsf{max}}_{\mathsf{id}}[\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0]$$

And, we have shown that $\mathbb{E}^{\mathsf{min}}_{\mathsf{id}}[\mathsf{approxN}\ (|lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}})\ 0] = |lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}}$ in Example 2.3 from §2.3, so it follows that

$$\mathbb{E}_{\mathsf{coerce}_{\mathsf{fun}}(f,0)}[\mathsf{resStep}^n_{\varphi}([e], \sigma)] = |lb_1|_{\mathrm{t}} + |lb_2|_{\mathrm{t}}$$

Of course, it is possible to derive rules about reading from the counter when we have less than the full fractional permission, but in that case, because there can be pending concurrent increments, it is harder to make guarantees about the probabilistic behavior. We will return to this issue in Chapter 7.

6.1.2 Counter Resources

In order to carry out the proofs of the rules from Figure 6.2, we will need a suitable notion of resource that will represent the state of the counter. Fortunately, the Iris Coq library already

$$\mathsf{LocInv}_{\gamma_l}(l) \triangleq \exists n.\, l \mapsto n * \boxed{\bullet n}^{\gamma_l}$$

$$\mathsf{ProbInv}_{\gamma_p,\gamma_c} \triangleq \exists n_1, n_2.\, \boxed{\bullet n_1}^{\gamma_p} * \boxed{\bullet n_2}^{\gamma_c} * (\mathsf{Prob}(\mathsf{approxN}\ n_1\ n_2) \vee \boxed{\circ (1, n_1)}^{\gamma_p})$$

$$\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(l, q, n) \triangleq \exists \iota_1, \iota_2, n'.\, \boxed{\mathsf{LocInv}_{\gamma_l}(l)}^{\iota_1} * \boxed{\mathsf{ProbInv}_{\gamma_p,\gamma_c}}^{\iota_2} * \boxed{\circ (q, n')}^{\gamma_l} * \boxed{\circ (q, n)}^{\gamma_p} * \boxed{\circ (q, n')}^{\gamma_c}$$

Figure 6.5: Invariants and definitions for proof.

contains a "counter resource" that does what we need. There are two kinds of these counter resources, represented by the following assertions:

$$\boxed{\bullet n}^{\gamma} \qquad \text{and} \qquad \boxed{\circ (q, n')}^{\gamma}$$

where n and n' are natural numbers, and $0 < q \leq 1$ is a rational. The construction here is much like the authoritative RA, except that the fragments $\circ (q, n)$ are annotated with a fractional permission q that tracks how many fragments there are. If we think of the counter as being composed of n "units", then the resource $\circ (q, n')$ represents a "stake" or ownership of n' of the units in the global counter. When $q = 1$, this represents full ownership, so no other threads have a stake[3].

Rules for using these assertions are given in Figure 6.4. The rules CountGeq and CountEq let us conclude that the global counter value must be at least as big as any stake's value; and when a stake's q value is 1, we furthermore know that the counter and the stake value are the same. The rule CountSep lets us join (or conversely, split) two stakes by summing their permissions and their count values, subject to the (implicit) constraint that q, q', and $q + q'$ all lie in the interval $(0, 1]$. The CountAlloc rule lets us create a new counter with some existentially quantified name. Finally, CountUpd lets us modify a counter: if we own the global value and a stake, we can update the value and the stake, so long as we preserve the part of the counter value owned by other stakes (represented by k in the rule).

6.1.3 Proofs of Specification

The definition of ACounter and the invariants used in the proof are given in Figure 6.5. The proof uses three counter resources to track (1) the number of increments left to perform in the monadic specification, (2) the accumulated count in the monadic specification, and (3) the actual count currently stored in the concrete program. We use two invariants to connect the counter resources to these intended interpretations. First, we have $\mathsf{LocInv}_{\gamma_l}(l)$ which says that the counter resource named γ_l stores some value n and the physical location l points to that same value n. Then, assertion $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$ says that there are two counter resources containing some n_1 and n_2, and the invariant either contains (a) the monadic specification resource

[3]Note that the q is *not* the fraction $\frac{n'}{n}$ of the global counter value represented by the stake's value.

$\mathsf{Prob}(\mathsf{approxN}\ n_1\ n_2)$ (*i.e.,* there are n_1 further increments to perform, and the monadic counter has accumulated a value of n_2), or (b) it contains the complete stake for one of the counter resources. Then ACounter says that these two invariants have been set up with some names, and we own a stake in the γ_p permission corresponding to the number of increments this permission allows. Further, for some n' there is a stake in the γ_l and γ_c counters both equal to n', which represents the total amount that this permission has been used to add to the counter.

We will only describe the proofs of ACounterIncr and ACounterRead, because ACounterNew is straight-forward.

Proof of ACounterIncr**.** Eliminating the existentials in the definition of ACounter, we get that the appropriate invariants have been set up and there is some n'-stake in γ_l and γ_c, along with the $n+1$ stake in γ_p. The first step of incr l reads the value of l; to perform this read the thread needs to own $l \mapsto v$ for some v. To get this resource, it opens the $\mathsf{LocInv}_{\gamma_\mathsf{l}}(l)$ invariant; after completing the read, the $l \mapsto v$ resource is returned to close the invariant. The code then takes the minimum of the value read and MAX, and binds this value to k.

It then performs $\mathsf{flip}(1, k+1)$. We want to use flip-couple to couple this flip with the monadic code. To do so, we first open the invariant $\mathsf{ProbInv}_{\gamma_\mathsf{p},\gamma_\mathsf{c}}$. We know this will contain $\boxed{\bullet\, n_1'}^{\gamma_\mathsf{p}}$ and $\boxed{\bullet\, n_2'}^{\gamma_\mathsf{c}}$ for some n_1' and n_2', and either $\mathsf{Prob}(\mathsf{approxN}\ n_1'\ n_2')$ or a full stake $\boxed{\circ\,(1, n_1')}^{\gamma_\mathsf{p}}$. However, the latter is impossible because the $\mathsf{ACounter}_{\gamma_\mathsf{l},\gamma_\mathsf{p},\gamma_\mathsf{c}}(l, q, n+1)$ resource entails ownership of $\boxed{\circ\,(q, n+1)}^{\gamma_\mathsf{p}}$, but $q+1 > 1$, contradicting CountPerm. So, we obtain $\mathsf{Prob}(\mathsf{approxN}\ n_1'\ n_2')$. Now, by CountGeq we know that $n_1' \geq n+1$, hence we can unfold $\mathsf{approxN}\ n_1'\ n_2'$ to obtain $\mathsf{Prob}(k \leftarrow \mathsf{approxIncr}\,;\ \mathsf{approxN}\ (n_1'-1)\ n_2')$.

We can now use flip-couple so long as we can exhibit a coupling between the concrete program's coin flip and approxIncr. First, because $0 \leq k \leq \mathsf{MAX}$, we can show that:

$$\begin{aligned}
&(\mathsf{ret}\ k+1) \oplus_{\frac{1}{k+1}} (\mathsf{ret}\ 0) \\
&\quad \subseteq (x \leftarrow \mathsf{ret}\ 0 \cup \cdots \cup \mathsf{ret}\ \mathsf{MAX}\,;\ (\mathsf{ret}\ x+1 \oplus_{\frac{1}{x+1}} \mathsf{ret}\ 0)) \\
&\quad \equiv \mathsf{approxIncr}
\end{aligned}$$

hence by Equiv, it suffices to exhibit a coupling between $(\mathsf{ret}\ \mathsf{True} \oplus_{\frac{1}{k+1}} \mathsf{ret}\ \mathsf{False})$ and $(\mathsf{ret}\ k+1 \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0)$. Taking $R(x,y)$ to be $(x = \mathsf{True} \wedge y = k+1) \vee (x = \mathsf{False} \wedge y = 0)$, then we can use P-Choice and Ret to prove the existence of an R-coupling.

Applying flip-couple with this coupling, we then have $\mathsf{Prob}(\mathsf{approxN}\ (n_1'-1)\ (n_2'+v'))$ where v' and the return value v of the $\mathsf{flip}(1,k)$ are related by R. We use CountUpd to update the thread's stake in the γ_p resource to n, and the global value to $n_1'-1$ (to record that a simulated increment has performed), similarly, we update the thread's stake in the γ_c counter to $n'+v'$ and the global value to $n_2'+v'$ (to record the new total) and then close the $\mathsf{ProbInv}_{\gamma_\mathsf{p},\gamma_\mathsf{c}}$ invariant.

The code then cases on the value v returned by the flip. If it is false, then v' is 0, the code returns, and the post condition holds. If v is true, then $v' = k+1$, the amount that the code adds using a fetch-and-add. We therefore open the $\mathsf{LocInv}_{\gamma_\mathsf{l}}(l)$ invariant again to get access to l, perform the increment, and update the γ_l counter and stake using CountUpd to record the fact that we are adding $k+1$.

Proof of ACounterRead The precondition $\mathsf{ACounter}_{\gamma_l,\gamma_p,\gamma_c}(l,1,0)$ represents the full stake in each counter, and the 0 argument means there are no pending increments to perform. Thus, when we open the $\mathsf{LocInv}_{\gamma_l}(l)$ and $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$ invariants we know that for some n', $l \mapsto n'$ and we have $\mathsf{Prob}(\mathsf{ret}\ n')$. So, we can read from l, knowing the returned value will be n'. After reading, we must close the invariant. This time we will keep the $\mathsf{Prob}(\mathsf{ret}\ n')$ resource so that we can put it in the post condition, instead we give up $\boxed{\circ\,(1,0)}^{\gamma_p}$ to satisfy the disjunction in $\mathsf{ProbInv}_{\gamma_p,\gamma_c}$.

6.1.4 Variations

In the mechanized proofs, I have verified two additional variations on this approximate counter example.

Non-deterministic Number of Increments

The first variation addresses a limitation of the specification we have described so far. Notice that to use the rules in Figure 6.2 and obtain a suitable derivation to use with Theorem 5.5, the total number of calls to incr must be a deterministic function of the program: we have to pick some n when we initialize the counter using ACounterNew. In the case of our example client, we chose n to be the number of times that true occurred in the two lists. But what if the number of calls to increment is itself probabilistic or non-deterministic? In this case we still would like to know that the expected value returned by the approximate counter is equal to the expected number of times the counter was incremented. However, if the number of times the counter is incremented is completely arbitrary, this expected value may not exist! To guarantee that the expected value will exist, our specification imposes an upper bound on the total number of increments that can be performed, and then allows us to establish a coupling with the following monadic computation:

$$\mathsf{approxN}'\ 0\ t\ z \triangleq \mathsf{ret}\ (t,z)$$
$$\mathsf{approxN}'\ (n+1)\ t\ z \triangleq (\mathsf{ret}\ (t,z)) \cup (k \leftarrow \mathsf{approxIncr}\,;\ \mathsf{approxN}'\ n\ (t+1)\ (z+k))$$

The first argument of approxN′ gives an upper bound on the remaining number of increments that can be performed, the second argument t tracks the total number of increments that have been done so far, and z again tracks the current value in the counter. When there are no remaining increments, it returns the pair (t,z). Otherwise, in contrast to the original approxN, when there is a possibility to perform an increment, there is a non-deterministic choice between simply returning (t,z) and actually doing the increment. Let f be the function $\lambda(x,y).\,x-y$. We prove that

$$\mathbb{E}_f^{\mathsf{min}}[\mathsf{approxN}'\ n\ 0\ 0] = \mathbb{E}_f^{\mathsf{max}}[\mathsf{approxN}'\ n\ 0\ 0] = 0$$

i.e., the expected value of the difference between the total number of increments done and the value in the counter is 0. The mechanization includes more flexible versions of the rules in Figure 6.2 that use this approxN′ instead.

Variance and Deviation Bounds

For the second variation, we consider a version of incr which directly uses the current value it reads from the counter, rather than taking the minimum of this value and MAX. This variation, and the updated monadic model are shown in Figure 6.6. The expected value for approxN n 0 is again n. In addition, the same rules from Figure 6.2 hold with these modifications. The proofs of the rules from Figure 6.2 are similar, so they are omitted[4].

The interesting aspect of this example is in bounding the *variance* of the count. This lets us use Chebyshev's inequality (2.6) to bound the probability that the count differs from n by more than a given amount.

To begin, we set $B(n, z) = \frac{3}{2}n^2 - \frac{1}{2}n + 3nz + z^2$. We will see that

$$\mathbb{E}^{\mathsf{max}}_{\lambda x.\, x^2}[\mathsf{approxN}\ n\ z] \leq B(n, z) \tag{6.1}$$

First, let us see how one might guess that such a bound would work. In the case of just the expected value of the count, it did not matter how the non-determinism in approxIncr was resolved – each call to approxIncr would add 1 in expectation. When we are considering the value of the count *squared*, the non-determinism does matter. It is somewhat intuitive that we can maximize the expected value squared by always resolving the non-determinism in approxIncr to choose the biggest possible value. If we write down the expected value for the version in which the non-determinism is always resolved this way, we get a recurrence relation that can be solved exactly[5] to yield the right hand side of (6.1).

So, the only challenge is to formally establish that this is indeed a maximum no matter how the non-determinism is resolved. The proof is by induction on n. The base case is trivial. For the inductive case, it suffices to show for all k satisfying $0 \leq k \leq z$ that

$$\begin{aligned}&\mathbb{E}^{\mathsf{max}}_{\lambda x.\, x^2}[x \leftarrow \mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0\,;\ \mathsf{approxN}\ (n-1)\ (z+x)] \\ &\qquad \leq B(n, z)\end{aligned}$$

Simplifying the left hand side and applying the induction hypothesis we get:

$$\begin{aligned}&\mathbb{E}^{\mathsf{max}}_{\lambda x.\, x^2}[x \leftarrow \mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0\,;\ \mathsf{approxN}\ (n-1)\ (z+x)] \\ &\quad = \frac{1}{k+1}\left(\mathbb{E}^{\mathsf{max}}_{\lambda x.\, x^2}[\mathsf{approxN}\ (n-1)\ (z+k+1)]\right) + \frac{k}{k+1}\left(\mathbb{E}^{\mathsf{max}}_{\lambda x.\, x^2}[\mathsf{approxN}\ (n-1)\ z]\right) \\ &\quad \leq \frac{1}{k+1}B(n-1, z+k+1) + \frac{k}{k+1}B(n-1, z)\end{aligned}$$

The last expression in this inequality is differentiable as a function of k on the interval $[0, z]$.

[4]The only interesting difference is we have to strengthen the invariants and resources used slightly to be able to ensure that the value of the counter never decreases.

[5]The resulting recurrence is essentially the same as one that arises in the analysis of variance for Morris's original counter algorithm.

The derivative is positive in this interval, so the function is increasing. Hence, we have:

$$\begin{aligned}
&\frac{1}{k+1}B(n-1, z+k+1) + \frac{k}{k+1}B(n-1, z) \\
&\quad \leq \frac{1}{l+1}B(n-1, 2z+1) + \frac{l}{l+1}B(n-1, z) \\
&\quad = B(n, z)
\end{aligned}$$

completing the inductive case.

Next, using Lemma 2.7, we have that:

$$\begin{aligned}
\mathbb{E}^{\mathsf{max}}_{\lambda x.\,(x-n)^2}[\mathsf{approxN}\ n\ 0] &\leq \mathbb{E}^{\mathsf{max}}_{\lambda x.\,x^2}[\mathsf{approxN}\ n\ 0] - 2n\mathbb{E}^{\mathsf{min}}_{\lambda x.\,x}[\mathsf{approxN}\ n\ 0] + n^2 \\
&\leq \frac{3}{2}n^2 - \frac{1}{2}n - 2n^2 + n^2 \\
&= \frac{1}{2}(n^2 - n)
\end{aligned}$$

And then applying Theorem 2.6 gives:

$$\Pr^{\mathsf{max}}_{\lambda x.\,|x-n|>\delta}[\mathsf{approxN}\ n\ 0] \leq \frac{1}{2\delta^2}(n^2 - n)$$

This bound is not particularly strong simply because the variance is very large. For example, if we take $n = 100$, $k = 90$, the right hand side is ≈ 0.61, which is not much of a guarantee in practical terms. This high variance also arises in Morris's original counting algorithm. For that reason, he also described a variant which has a parameter that one can tune to decrease variance. Translated to the setting of the concurrent version above, rather than just directly using the current value k read from the counter during an increment, one would first divide k by some factor b. For b large enough, one can get much tighter bounds from Chebyshev's inequality. Because the language we are considering here does not have primitives for rational arithmetic, and scaling the probabilities in this way does not really change the challenging part of the verification, we will not analyze this variant.

6.2 Concurrent Skip List

For our next example, we verify properties of a probabilistic 2-level concurrent skip list. The code and proofs for this example are more complex, so we will only briefly recall the high-level description of the algorithm from §1.1.3 and give an overview of the proof.

Recall that a 2-level skip list is composed of 2 sorted linked lists, where the set of elements occurring in the "top" list is a subset of the bottom list, and every node in the top list contains a pointer to the node with the same value in the bottom list. We require all keys to be integers between INTMIN and INTMAX, which are some arbitrarily chosen parameters. Dummy sentinel nodes occur at the beginning and end of each list containing INTMIN and INTMAX respectively. To find a key k, we start searching in the top list. If we encounter a key with a value larger than the one sought, we descend to the bottom list, following the pointer in the top list node with the largest key less than k, and resume searching from there.

$$
\begin{array}{l}
\mathsf{incr}\ l \triangleq \\
\quad \mathsf{let}\ k = !l\ \mathsf{in} \\
\quad \mathsf{let}\ b = \mathsf{flip}(1/(k+1))\ \mathsf{in} \\
\quad \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{FAA}(l, k+1); ()) \\
\quad \mathsf{else}\ ();
\end{array}
$$

$$
\begin{array}{l}
\mathsf{approxIncr}\ c \triangleq \\
\quad k \leftarrow \mathsf{ret}\ 0 \cup \cdots \cup \mathsf{ret}\ c\,; \\
\quad \mathsf{ret}\ (k+1) \oplus_{\frac{1}{k+1}} \mathsf{ret}\ 0 \\
\\
\mathsf{approxN}\ 0\ z \triangleq \mathsf{ret}\ z \\
\mathsf{approxN}\ (n+1)\ z \triangleq \\
\quad k \leftarrow \mathsf{approxIncr}\ z\,; \\
\quad \mathsf{approxN}\ n\ (z+k)
\end{array}
$$

Figure 6.6: Counter variant without a maximum increment size.

Each node also contains a lock. When searching, no locks need to be acquired. However, when inserting a new key k, we first find the nodes N_t and N_b that ought to be the new key's "predecessor" in the top and bottom list respectively, acquiring their locks once we identify them. (Once we get each predecessor's lock, we check again that it is still a proper predecessor.) We then insert a new node for k in both lists with probability p, and otherwise just insert k into the bottom list.

We will just consider the case where $p = 1/2$ here. This means that on average, half of the keys will be in the top list, so when searching in the top list we will only have to examine roughly half as many keys as we would in a regular linked list. Then, if we do not find the key in the top list, we will have a few additional keys to examine in the bottom. We will formally establish a bound on the expected number of comparisons.

6.2.1 Monadic Model

We follow the same pattern as in our verification of the approximate counter example: we first define a monadic model of the data structure, bound appropriate expected values of the monadic computation, and then develop rules that can be used to prove the existence of a coupling between programs using the skip list and the monadic model.

Our monadic model is the following:

$$
\begin{array}{rcl}
\mathsf{skiplist}\ \epsilon\ tl\ bl & \triangleq & \mathsf{ret}\ (\mathsf{sort}(tl), \mathsf{sort}(bl)) \\
\mathsf{skiplist}\ (k :: l)\ tl\ bl & \triangleq & k' \leftarrow \bigcup_{i \in k::l} \mathsf{ret}\ i\,; \\
 & & tl' \leftarrow (\mathsf{ret}\ tl) \oplus_{1/2} (\mathsf{ret}\ k' :: tl)\,; \\
 & & \mathsf{skiplist}\ (\mathsf{remove}\ k'\ (k :: l))\ tl'\ (k' :: bl)
\end{array}
$$

The computation skiplist l tl bl simulates adding keys from the list l to a skip list, where the arguments tl and bl are lists represent the keys in the top and bottom lists of the skip list, respectively. If the first argument l is empty, it sorts tl and bl and returns the result. If l is non-empty, it first non-deterministically selects a key k' from l. Then, with probability 1/2 it adds

this key to tl. It then removes any copies of k' from l, and recurses to process the remaining elements with the updated top and bottom lists. (There is no point in keeping the arguments tl and bl sorted throughout the recursive calls in this monadic formulation.)

We define a function $\mathsf{skipcost}(tl, bl, k)$ which gives the number of comparisons needed to check if k is in the skip list when the elements in the top and bottom lists are tl and bl, respectively:

$$\begin{aligned}
\mathsf{topcost}(tl, k) &= 1 + |\{i \in tl \mid \mathsf{INTMIN} < i < k\}| \\
\mathsf{maxbelow}(tl, k) &= \max(\{i \in tl \mid i < k\} \cup \{\mathsf{INTMIN}\}) \\
\mathsf{botcost}(tl, bl, k) &= 1 + |\{i \in bl \mid \mathsf{maxbelow}(tl, k) < i < k\}| \\
\mathsf{skipcost}(tl, bl, k) &= \begin{cases} \mathsf{topcost}(tl, k) & \text{if } k \in tl \\ \mathsf{topcost}(tl, k) + \mathsf{botcost}(tl, bl, k) & \text{if } k \notin tl \end{cases}
\end{aligned}$$

If the key k is in the top list, then the number of comparisons is 1 plus the number of elements in the list less than k ($\mathsf{topcost}(tl, k)$). If k is not in the top list, then we must first still perform the same number of comparisons while searching through the top list. Then we search in the bottom list starting from the largest key less than k that was in tl ($\mathsf{maxbelow}(tl, k)$). The total number of comparisons in the second list is the number of keys between $\mathsf{maxbelow}(tl, k)$ and k ($\mathsf{botcost}(tl, bl, k)$).

We then bound $\mathbb{E}^{\mathsf{max}}_{\mathsf{skipcost}(-,-,k)}[\mathsf{skiplist}\ l\ \epsilon\ \epsilon]$ to obtain an upper bound on the expected value of searching for a key k. The first step in the proof is to consider an alternative version of the monadic code in which the elements are inserted into the skip list in a deterministic order. That is, we define:

$$\begin{aligned}
\mathsf{skiplist}'\ \epsilon\ tl\ bl &\triangleq \mathsf{ret}\ (\mathsf{sort}(tl), \mathsf{sort}(bl)) \\
\mathsf{skiplist}'\ (k :: l)\ tl\ bl &\triangleq hd \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ k :: \epsilon)\,; \\
&\quad\ \mathsf{skiplist}'\ l\ (hd +\!\!+ tl)\ (k :: bl)
\end{aligned}$$

We will prove that if l has no duplicate elements, then $\mathsf{skiplist}\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l\ tl\ bl$. We first need several auxiliary lemmas about $\mathsf{skiplist}'$.

Lemma 6.1. If tl is a permutation of tl' and bl is a permutation of bl', then $\mathsf{skiplist}'\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l\ tl'\ bl'$.

Proof. The proof is by induction on l, and follows from the fact that we sort both tl and bl at the end of the recursive calls, so the intermediary order does not matter. □

Lemma 6.2. If l is a permutation of l', then for all tl and bl, $\mathsf{skiplist}'\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l'\ tl\ bl$.

Proof. In our formalization, the permutation relation $\mathsf{perm}(l, l')$ is inductively generated by the following rules:

$$\mathsf{perm}(\epsilon, \epsilon) \qquad \frac{\mathsf{perm}(l, l')}{\mathsf{perm}(x :: l, x :: l')} \qquad \mathsf{perm}(x :: y :: l, y :: x :: l) \qquad \frac{\mathsf{perm}(l, l') \qquad \mathsf{perm}(l', l'')}{\mathsf{perm}(l, l'')}$$

Hence, by induction on $\mathsf{perm}(l, l')$, it suffices to show the following cases

1. $\mathsf{skiplist}'\ \epsilon\ tl\ bl \equiv \mathsf{skiplist}'\ \epsilon\ tl\ bl$.

   This follows from reflexivity of $\equiv$.

2. If $\mathsf{skiplist}'\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l'\ tl\ bl$, then $\mathsf{skiplist}'(x :: l)\ tl\ bl \equiv \mathsf{skiplist}'(x :: l')\ tl\ bl$.

   This case follows from the fact that $\equiv$ is a congruence relation on the monad operations.

3. $\mathsf{skiplist}'\ (x :: y :: l)\ tl\ bl \equiv \mathsf{skiplist}'\ (y :: x :: l)\ tl\ bl$.

   For this case, we have

$$
\begin{aligned}
&\mathsf{skiplist}'\ (x :: y :: l)\ tl\ bl \\
&\quad\equiv h_1 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ x :: \epsilon)\,; && \text{(definition of skiplist}'\text{)} \\
&\qquad h_2 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ y :: \epsilon)\,; \\
&\qquad \mathsf{skiplist}'\ l\ (h_2 +\!\!+ h_1 +\!\!+ tl)\ (y :: x :: bl) \\
&\quad\equiv h_2 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ y :: \epsilon)\,; && \text{(last rule of Figure 2.1)} \\
&\qquad h_1 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ x :: \epsilon)\,; \\
&\qquad \mathsf{skiplist}'\ l\ (h_2 +\!\!+ h_1 +\!\!+ tl)\ (y :: x :: bl) \\
&\quad\equiv h_2 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ y :: \epsilon)\,; && \text{(Lemma 6.1)} \\
&\qquad h_1 \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ x :: \epsilon)\,; \\
&\qquad \mathsf{skiplist}'\ l\ (h_1 +\!\!+ h_2 +\!\!+ tl)\ (x :: y :: bl) \\
&\quad\equiv \mathsf{skiplist}'\ (y :: x :: l)\ tl\ bl
\end{aligned}
$$

4. If $\mathsf{skiplist}'\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l'\ tl\ bl$ and $\mathsf{skiplist}'\ l'\ tl\ bl \equiv \mathsf{skiplist}'\ l''\ tl\ bl$, then $\mathsf{skiplist}'\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l''\ tl\ bl$.

   This follows from transitivity of $\equiv$.

□

Lemma 6.3. If l has no duplicate elements, then $\mathsf{skiplist}\ l\ tl\ bl \equiv \mathsf{skiplist}'\ l\ tl\ bl$.

Proof. The proof is by induction on the length of l. The base case is trivial. For the inductive case, we have

$$
\begin{aligned}
&\mathsf{skiplist}\ (k :: l)\ tl\ bl \\
&\quad \equiv\ k' \leftarrow \bigcup_{i \in k::l} \mathsf{ret}\ i\,; \\
&\qquad\ tl' \leftarrow (\mathsf{ret}\ tl) \oplus_{1/2} (\mathsf{ret}\ k' :: tl)\,; \\
&\qquad\ \mathsf{skiplist}\ (\mathsf{remove}\ k'\ (k :: l))\ tl'\ (k' :: bl) \\
&\quad \equiv \bigcup_{i \in k::l} \begin{pmatrix} tl' \leftarrow (\mathsf{ret}\ tl) \oplus_{1/2} (\mathsf{ret}\ i :: tl)\,; \\ \mathsf{skiplist}\ (\mathsf{remove}\ i\ (k :: l))\ tl'\ (i :: bl) \end{pmatrix} \\
&\quad \equiv \bigcup_{i \in k::l} \begin{pmatrix} h \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ i :: \epsilon)\,; \\ \mathsf{skiplist}\ (\mathsf{remove}\ i\ (k :: l))\ (h +\!\!+ tl)\ (i :: bl) \end{pmatrix} \\
&\quad \equiv \bigcup_{i \in k::l} \begin{pmatrix} h \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ i :: \epsilon)\,; \\ \mathsf{skiplist}'\ (\mathsf{remove}\ i\ (k :: l))\ (h +\!\!+ tl)\ (i :: bl) \end{pmatrix}
\end{aligned}
$$

We want to show that this union is equivalent to $\mathsf{skiplist}'\ (k :: l)\ tl\ bl$. Recall from Figure 2.1 that for all $\mathcal{I}, \mathcal{I} \cup \mathcal{I} \equiv \mathcal{I}$. Generalizing this principle, it suffices to show that for each $i \in k :: l$,

$$
\begin{aligned}
&h \leftarrow (\mathsf{ret}\ \epsilon) \oplus_{1/2} (\mathsf{ret}\ i :: \epsilon)\,; \\
&\mathsf{skiplist}'\ (\mathsf{remove}\ i\ (k :: l))\ (h +\!\!+ tl)\ (i :: bl)
\end{aligned}
\qquad \equiv \mathsf{skiplist}'\ (k :: l)\ tl\ bl \tag{6.2}
$$

Since $k :: l$ has no duplicates by assumption, there exists l_1, l_2 such that $i :: l_1 +\!\!+ l_2$ is a permutation of $k :: l$, and $\mathsf{remove}\ i\ (k :: l)$ is a permutation of $l_1 +\!\!+ l_2$. Then the left hand side of (6.2) is equivalent to $\mathsf{skiplist}'\ (i :: l_1 +\!\!+ l_2)\ tl\ bl$ by definition, so that the result follows from Lemma 6.2. □

This lemma means that it suffices to bound the expected values of $\mathsf{skiplist}'$ in order to obtain bounds on skiplist. The advantage of working with $\mathsf{skiplist}'$ is that the definition is simpler, structurally recursive, and has no adversarial non-determinism. Thus, we can follow an argument similar to the usual pencil-and-paper analysis of skiplists (though somewhat simplified, since we are only considering a version with 2 lists). Assuming l has no duplicates, the key k and all keys in l lie between INTMIN and INTMAX, and there are n keys less than k in l, one has:

$$
\mathbb{E}^{\max}_{\mathsf{skipcost}(-,-,k)}[\mathsf{skiplist}\ l\ \epsilon\ \epsilon] = \mathbb{E}^{\max}_{\mathsf{skipcost}(-,-,k)}[\mathsf{skiplist}'\ l\ \epsilon\ \epsilon] \leq 1 + \frac{n}{2} + 2\left(1 - \frac{1}{2^{n+1}}\right) \tag{6.3}
$$

The proof comes from bounding the expected values of topcost and botcost and then adding them. The former is essentially a binomial distribution, and the latter is a (truncated) geometric distribution. The bound means that on average we have to do about half the number of comparisons that would be required to search for the key in a regular sorted linked list.

6.2.2 Weakest Precondition Specifications and Proof Overview

Figure 6.7 shows rules derived rules for the skip list. The specification defines an assertion $\mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b)$, which represents permission to access a skip list whose top left sentinel is v. The argument Γ is just a set of resource names (like the γ's in the counter example), q is a fractional permission, S is the finite set of keys which may be added to the list, and S_t and S_b are a subset of the keys currently in the top and bottom lists. Additional keys from S may be in either S_t or S_b, but the owner of this permission assertion knows that they contain *at least* these sets.

The expression newSkipList creates a new skip list. The precondition for the rule in SkipNew requires us to own the monadic computation[6] $\mathsf{Prob}(\text{skiplist } S\ \epsilon\ \epsilon)$. The post condition gives the full permission ($q = 1$) to access the skip list, with empty top and bottom lists. To use this rule, all the keys in S must be between INTMIN and INTMAX. Notice here that the set of keys S which will be added to the skiplist must be deterministic, so that it can be decided in this precondition (much like our original specification for the approximate counters required the total number of increments to be deterministic). This restriction is important: if the keys to be added are non-deterministically selected, and a client can observe the state of the skip list, it can insert a special sequence of keys in such a way so as to force a large number of comparisons to find a particular target key.

We use addSkipList to insert a key k into the skip list. The post condition in SkipAdd indicates that we now know that the added key k is in the bottom list. On the other hand, the client does not know whether the key was added to the top list or not, so in the permission for the post condition, the contents of the top list are given by some existentially quantified S'_t.

The function memSkipList checks whether a key is in the skip list. It returns a pair (b, z), where b is a boolean indicating whether the key was in the set or not, and z is the number of key comparisons performed. The rule SkipMem says that if we have the full permission for the skip list, then the boolean b indeed reflects whether the key is in the set or not, and z is in fact equal to the cost function $\text{skipcost}(S_t, S_b, k)$ we defined above. (One can also prove triples for when we have less than the full permission of the list, *i.e.*, $q < 1$.)

The rule SkipSep lets us split and join together the SkipPerm permission so that separate threads can use the skip list. Finally, SkipShift lets us do an update to convert a full SkipPerm permission in which we have added all the keys in S to the skip list back into a Prob permission in which the monadic computation is finished. This lets us prove triples of the form required for our probabilistic adequacy result.

In the mechanized proofs, I have used this specification to verify a simple client in which two threads concurrently add lists of integers to a skip list set, and then after they both finish, one looks up a key using memSkipList and returns the number of comparisons performed. Corollary 5.2 is then used to get a bound on the expected value of this number.

How are the rules in Figure 6.7 proved? The formal proof is more complex than for the counter example, but it has a similar high level structure, which I will describe briefly. There are two invariants. One is used to establish the "functional" correctness of the skip list and

[6] Here, S is a set, whereas the arguments to skiplist are lists. However, it is easy to show that if l', tl', and bl' are permutations of the lists l, tl, and bl, respectively, then skiplist $l\ tl\ bl \equiv$ skiplist $l'\ tl'\ bl'$, so it makes no difference if we treat the first argument instead as an unordered set.

SkipNew

$$\frac{\forall k \in S.\, \mathsf{INTMIN} < k < \mathsf{INTMAX}}{\mathsf{Prob}(\mathsf{skiplist}\ S\ \epsilon\ \epsilon) \vdash \mathsf{wp}\ \mathsf{newSkipList}\ \{v.\, \exists \Gamma.\, \mathsf{SkipPerm}_\Gamma(1, v, S, \emptyset, \emptyset)\}}$$

SkipAdd

$$\frac{k \in S}{\mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b) \vdash \mathsf{wp}\ \mathsf{addSkipList}\ v\ k\ \{\_.\, \exists S_t'.\, \mathsf{SkipPerm}_\Gamma(q, v, S, S_t', S_b \cup \{k\})\}}$$

SkipMem

$$\frac{\mathsf{INTMIN} < k < \mathsf{INTMAX}}{\mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S_b) \vdash \mathsf{wp}\ \mathsf{memSkipList}\ v\ k \left\{ (b, z).\ \begin{array}{l} \mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S_b) * (b = \mathsf{True} \Rightarrow k \in S_b) \\ * (b = \mathsf{False} \Rightarrow k \notin S_t \cup S_b) * (z = \mathsf{skipcost}(S_t, S_b, k)) \end{array} \right\}}$$

SkipSep

$$\mathsf{SkipPerm}_\Gamma(q + q', v, S, S_t \cup S_t', S_b \cup S_b') \dashv\vdash \mathsf{SkipPerm}_\Gamma(q, v, S, S_t, S_b) * \mathsf{SkipPerm}_\Gamma(q', v, S, S_t', S_b')$$

SkipShift

$$\mathsf{SkipPerm}_\Gamma(1, v, S, S_t, S) \vdash \Rrightarrow_{\mathcal{E}} \mathsf{Prob}(\mathsf{ret}\ (\mathsf{sort}(S_t), \mathsf{sort}(S)))$$

Figure 6.7: Specification for skip list.

maintains all the properties of the physical representation of the data structure, *i.e.,* that the two lists in the skip list are sorted, that each node in the top list points to a node with the same value in the bottom list, that there are locks protecting each node, and so on. This invariant is what one would use in regular Iris to prove that the skip list implements the expected search structure interface. It plays a role similar to that of LocInv in the counter example.

The second invariant functions like ProbInv from the counter example and establishes a connection between ghost resources and the status of the monadic computation. In place of the counter resources from §6.1.2, the primary resource algebra used involves resources of the form $\bullet\, S$ and $\circ\,(q, S)$, where q is again a fractional permission, but S is now a set of keys instead of an integer. The rules for these resources are similar to those for the counter resource shown in Figure 6.4, except that in place of addition and the $\leq$ ordering on integers, we use union and the subset ordering on sets. For example, we have the following two rules:

$$\boxed{\bullet\, S}^{\gamma} * \boxed{\circ\,(q, S')}^{\gamma} \vdash \ulcorner S \supseteq S' \urcorner \qquad \boxed{\circ\,(q, S)}^{\gamma} * \boxed{\circ\,(q', S')}^{\gamma} \dashv\vdash \boxed{\circ\,(q + q', S \cup S')}^{\gamma}$$

The probabilistic invariant then has the following structure:

$$\begin{aligned}&\mathsf{SkipProbInv}_{\gamma_1,\gamma_2,\gamma_3,\gamma_4}(S)\\ &\quad \triangleq \exists S_t, S_b.\, \ulcorner S_t \subseteq S_b \urcorner * (\text{ownership of resources parameterized by } S_t \text{ and } S_b \ldots)\\ &\quad * (\mathsf{Prob}(\mathsf{skiplist}\ (S \setminus S_b)\ S_t\ S_b) \vee \boxed{\circ\,(1, S_b)}^{\gamma_3})\end{aligned}$$

This says that there are two sets S_t and S_b, which represent the elements that have been inserted into the top and bottom lists in the monadic representation. The connection between these two sets and the eventual physical representation are enforced via authoritative versions of ghost resources that appear in this invariant. Finally, the invariant either contains the monadic computation resource or the full fragment of one of the ghost resources. We can see that this definition is similar to that of ProbInv. Recall that in that definition, we quantified over two counts n_1 and n_2 which appeared as arguments in the monadic computation and in ghost resources, and the invariant contained either the monadic computation or a full fragment of a ghost resource.

When proving SkipAdd, for the case where the element we are adding is not already in the list, this invariant is opened to obtain the monadic computation after acquiring the locks for the predecessor nodes. We argue that while we hold these locks, the element k we are adding cannot yet have been inserted in the monadic computation. We then couple the probabilistic choice of the concrete code to the probabilistic choice in the monadic code, resolving the nondeterminism in the latter to insert k next. The ghost resources are updated appropriately to restore the invariant, and then we verify the code that actually does the insertions. However, the latter only involves interacting with the non-probabilistic invariant. This is similar to the steps used in verifying ACounterIncr, where we used ghost resources to argue that there was at least one more pending increment to perform in the monadic computation after opening ProbInv, and then after using the coupling rule, we subsequently used LocInv to reason about the concrete step of modifying the physical counter value.

The proof of SkipShift opens the SkipProbInv invariant, but takes out the Prob assertion and restores the invariant by instead putting back the right side of the disjunct, which is implied by the precondition $\mathsf{SkipPerm}_{\Gamma}(1, v, S, S_t, S)$. Again, this is the same approach used in the proof of ACounterRead.

Chapter 7

Conclusion

7.1 Summary

This dissertation described a concurrent separation logic, Polaris, with support for probabilistic relational reasoning. The approach taken was to first adapt the notion of couplings to Varacca and Winskel's monadic model of probabilistic and non-deterministic choice. Then, the definition of weakest precondition in Iris was altered so as to require couplings between steps of a concrete program and a monadic specification. The proof of the adequacy theorem inductively assembled these per-step couplings into a complete coupling between the whole concrete program and the monadic specification. By constructing an appropriate coupling, the analysis of the monadic specification's probabilistic behavior could then be translated into results about concrete programs. Because of the way the extensions were added, all of Iris's pre-existing features for reasoning about complex concurrent programs continued to work without change. The logic was used to analyze two of the three motivating examples mentioned in Chapter 1. The results about the monad of indexed valuations, the soundness of the program logic, as well as the example proofs have all been verified in the Coq theorem prover.

7.2 Comparison with Related Work

Now that Polaris has been described in detail, we return to some of the related program logics discussed in §1.2 and compare them with Polaris. Besides Iris, whose relationship to Polaris has been discussed at length, the most closely related are pRHL [10], Quantitative Separation Logic [19], and Probabilistic Rely-Guarantee [84].

pRHL As described previously, Polaris re-uses pRHL's approach of basing relational reasoning about probabilistic programs on the concept of couplings. However, there are technical differences in how relational reasoning is done in the two logics.

First, the logics support different programming language features. pRHL is restricted to reasoning about sequential programs and does not support separating conjunction for reasoning about pointer manipulating programs. On the other hand, the soundness theorem for pRHL does not make the strong termination assumptions that Polaris requires.

Second, in Polaris, relational reasoning is done between a program in a given language and a monadic computation, whereas in pRHL one establishes a coupling between two programs expressed in the same language. In Polaris, the relationship established between the program and the monadic model is used to translate quantitative bounds on the latter to the former. Although this same translation of bounds can also be done in pRHL, the developers of pRHL (and its extensions) have used the logic to establish couplings that imply different sorts of properties as well. For example, by constructing a coupling between a program and the *same* program run on slightly different inputs, one can prove that an algorithm is differentially private [12] or that a machine learning algorithm is stable [18].

It does not seem difficult to adjust pRHL so that one could reason relationally about programs written in two different languages, or between a program and a monadic model. Conversely, Polaris could also be adapted to support relational reasoning about programs written in the same language. Rather than using ghost resources to model monadic computations, one would instead use them to model expressions from the probabilistic language itself. This approach has been previously used for relational reasoning about non-probabilistic programs in Iris [43, 75, 115].

A more substantive difference is that pRHL uses a Hoare *quadruple* of the form

$$\{P\}\, e_1 \sim e_2\, \{Q\}$$

where the two programs being reasoned about both appear in the four-part judgment. In contrast, Polaris extends a unary logic by encoding the monadic computation as an assertion that can appear in the pre and post-condition. As explained in §5.2, this representation is used instead of quadruples because in the concurrent setting, it is not clear how to state the fork rule with quadruples. Because multiple threads need to concurrently couple steps with the monadic computation, it makes sense to view it as a shared ghost resource.

Quantitative Separation Logic The Quantitative Separation Logic of Batz et al. [19] is a non-concurrent separation logic for reasoning about sequential probabilistic programs. Instead of using relational reasoning, it uses the quantitative style of Morgan et al. [86]. That is, assertions are functions from memory states to non-negative real numbers representing probabilities or expected values. The weakest precondition $\mathsf{wp}\ e\ \{P\}$ is a function such that if e is executed in state σ, then the expected value of P on the state of the program after execution is equal to $\mathsf{wp}\ e\ \{P\}(\sigma)$.

The sequential language considered by Batz et al. [19] has while loops, memory that can store integer values, and recursive procedure calls. In contrast, the example language we have instantiated Polaris with in this dissertation has support for concurrency, higher-order functions, and higher-order store. Although the examples verified in Chapter 6 do not use higher-order functions or higher-order store directly, the example client verified using the approximate counter specification does use higher-order functions.

The soundness theorem of Quantitative Separation Logic applies to programs that do not necessarily terminate, unlike Polaris. One of the examples considered by Batz et al. is a program that probabilistically extends a list so that the final length is geometrically distributed. They point out that this program does not terminate in a bounded number of steps, so that it lies

outside the scope of Polaris. However, when reasoning about programs with loops or recursive procedure calls, the rules of Quantitative Separation Logic discussed by Batz et al. only appear to allow one to obtain *upper bounds* on expected values[1]. For example, the verification of the geometric list example just described only shows that the expected value of the length will be ≤ 1. The other examples considered by Batz et al. involving loops similarly only establish bounds on expected values. In contrast, when reasoning about purely sequential programs with Polaris, $\mathbb{E}^{\mathsf{max}}$ and $\mathbb{E}^{\mathsf{min}}$ of the corresponding monadic model will be identical, so that Polaris can be used to obtain exact expected values of recursive programs when they terminate in a bounded number of steps.

Quantitative Separation Logic features the following version of the frame rule (with additional side conditions omitted):

$$\mathsf{wp}\ e\ \{P\} * Q \leq \mathsf{wp}\ e\ \{P * Q\}$$

Batz et al. suggest that this is the natural re-formulation of the traditional frame rule, $\mathsf{wp}\ e\ \{P\} * Q \vdash \mathsf{wp}\ e\ \{P * Q\}$, since the inequality ordering $\leq$ is the quantitative analogue of $\vdash$. As they point out, one cannot expect this inequality to be an equality, because $\mathsf{wp}\ e\ \{P * Q\} \not\vdash \mathsf{wp}\ e\ \{P\}$ in traditional separation logic. However, the fact that the quantitative frame rule is only an inequality appears problematic for modular reasoning. For instance, the verification of the geometric list example mentioned above gives an upper bound on length. If we now try to re-use this result to reason about executing the list program as part of a larger program, the frame rule gives an inequality in the *opposite* direction than the one required. Because Polaris uses relational reasoning to establish a connection to a monadic model whose quantitative properties are analyzed separately, this issue does not arise. Thus, clients of the data structures verified in Chapter 6 can use the specifications proved there to obtain the desired bounds.

Probabilistic Rely-Guarantee McIver et al. [84] develop a probabilistic version of rely-guarantee logic. They model concurrent probabilistic programs using a probabilistic variant of concurrent Kleene algebra [56]. McIver et al. do not fix a syntax for a concrete language, but explain how iteration, conditional statements, and parallel composition are modeled using the Kleene algebra operations and reasoned about using the logic. It is unclear whether this formalism could be extended to handle higher-order functions, dynamically allocated state, or fork-join concurrency, which are all supported by Polaris. Additionally, as previously described in §1.2.3, one shortcoming of rely-guarantee logic is that it does not support state-local reasoning, because the rely and guarantee conditions have to be checked against the program's global state. The probabilistic extensions to rely-guarantee do not address this limitation. It is not clear how one could modularly verify data structures like the concurrent skip list, where clients ought to be able to use the specification without having to be concerned about the internal representation of the data structure. Because Polaris builds on Iris, we are able to re-use Iris's features for modular reasoning about such data structures.

[1]Quantitative Separation Logic also has a weakest *liberal* precondition for partial correctness reasoning, in which the loop-rules give lower bounds, but this can only be used to reason about probabilities of events, rather than arbitrary expected values.

7.3 Future Work

Polaris is far from the definitive solution to the challenges of reasoning about probabilistic concurrent programs. There are many interesting problems that could be explored by building on the work presented here. In this section we survey some possibilities.

7.3.1 Instantiation with Other Languages

Polaris, like the original Iris, is parameterized by a fairly generic programming language. We have instantiated it with a single simple ML-like language, which was sufficient to express our examples. However, it would be interesting to consider other languages, particularly those which have already been used with Iris. For example, Krogh-Jespersen et al. [76] use Iris to reason about a programming language for distributed systems, and verify a centralized load-balancing server program. The focus of this verification is about functional correctness, *i.e.,* showing that the load balancer actually properly assigns tasks without affecting their results. However, an important consideration for such systems is to ensure that the load is indeed well-balanced. Some of the most important practical ways of efficiently doing load balancing tasks involve randomized algorithms [9, 111]. By adapting the work of Krogh-Jespersen et al. to use our probabilistic extension of Iris, we would be able to reason about such algorithms. For non-probabilistic operations of the language, the proof rules derived using WP-LIFT-STEP should work automatically, and we would only have to derive new coupling rules for any added probabilistic constructs. There are many other randomized algorithms used in distributed systems which would be interesting to verify [81, 105].

The language we have considered used sequentially consistent shared memory. However, modern processors only offer weaker consistency guarantees by default [1, 109], and special synchronization instructions need to be used to restore sequential consistency. A number of systems programming languages now have memory models that, like hardware, are weaker than sequential consistency by default. These weak memory effects are difficult to account for, and several program logics have been developed to reason about programs written in the setting of these weaker memory models [36, 110, 112, 123, 125]. One of these, iGPS [67], is encoded as a logic on top of Iris. If the encoding was instead done on top of our probabilistic extension to Iris, it might be possible to also incorporate probabilistic reasoning in iGPS. This could be used to analyze weak memory versions of probabilistic data structures like hash tables [120].

7.3.2 Alternative Monads

The probabilistic adequacy theorems in §5.3 do not rely in an essential way on specific properties of the monad M_{NI} or non-deterministic couplings. Other than the monad laws, the main facts used are BIND and RET to construct couplings for whole programs from step-wise couplings, and then Theorem 2.8 is used to actually make conclusions about expected values from the existence of a coupling.

This makes it possible to consider alternative monads for the combination of probabilistic and non-deterministic choice, along with suitable notions of coupling for them. In §2.6 the monad of Tix et al. [119] was mentioned. Using this monad instead might also allow us to

consider schedulers that can themselves make randomized choices, instead of the deterministic schedulers described in §5.1.

On a simpler level, Iris and Polaris can of course always be used to reason about non-concurrent programs. This might be useful because only recently have Batz et al. [19] developed a probabilistic separation logic, and the language considered in that work does not have features like higher order state, which Polaris supports. To reason about sequential code, we would not need to represent non-deterministic choice in our monad, and we could use the standard Giry monad [48] for probabilistic choice, which would let us reason about programs that sample from distributions that are not necessarily discrete. Prior work has already formalized analyses of algorithms expressed using this monad (*e.g.,* [37]). Alternatively, one could use the monad of measures on the category of quasi-Borel spaces [54]. Sato et al. [104] have recently presented a logic for reasoning about programs whose denotational semantics is given using this category.

7.3.3 Termination

The probabilistic adequacy results, Theorem 5.1 and Corollary 5.2, only apply to programs with schedulers under which they are guaranteed to terminate in a bounded number of steps. One generalization would be to consider programs which merely terminate with probability 1, and without an a priori bound on the number of steps taken. An approach to extending Corollary 5.2 to this case would be to consider for each n the finite approximation of the program which takes a maximum of n steps and returns some arbitrary default value if the program has not terminated. Something like Corollary 5.2 would then apply to this. If as n goes to infinity, the probability of non-termination goes to 0, the expected value of these approximations would converge to that of the full program[2]. Hence, bounds that hold for all approximations would hold for the limit.

A more challenging task is to develop a logic to *prove* that concurrent randomized programs terminate with probability 1 when run under certain kinds of schedulers. Developing concurrent separation logics for proving termination (and related liveness properties) for non-probabilistic concurrent programs is already challenging and is the subject of much recent work [29, 57, 79, 115]. A core idea behind the cited work in this area is to have a ghost resource that is forced to "shrink" each time the program takes a step. For example, one might start with some finite number of "tokens", and then the definition of weakest precondition is changed so that each time a reduction happens one must give up one of these tokens[3]. Then, if we establish a weakest precondition starting with n tokens, the program must terminate in no more than n steps. Tokens can be transferred between threads like any other resource in order to account for the fact that actions by one thread may cause another to take additional steps.

Ngo et al. [92] have adapted this idea to reason about the expected resource usage of probabilistic sequential programs. Instead of counting each step, there is a special expression $\mathsf{tick}(n)$ which is thought of as costing n "resources" to execute, and it is the total of these costs that are tracked by requiring n tokens to be given up when this expression is executed. If the program

[2]This argument hinges on the fact that the function f in the statement of Corollary 5.2 is bounded on the values returned by the terminating program.

[3]Much like how in Chapter 5 the definition of weakest precondition was changed to require the existence of step-wise couplings

makes a probabilistic choice[4] $\mathsf{flip}(z_1, z_2)$, then there is a rule that lets us transform n tokens in the precondition into n_t and n_f tokens in the postconditions for the cases where $\mathsf{flip}(z_1, z_2)$ evaluates to true and false respectively, subject to the conditions

$$n = \frac{z_1}{z_2} n_t + (1 - \frac{z_1}{z_2}) n_f$$

and $n_t, n_f \geq 0$. Thus, the expected number of tokens (and hence, the pending number of tick costs) is bounded by n. This is useful to reason about situations where the pending resource use varies based on the outcome of the randomized choice.

More generally, in place of a finite number of tokens, the resources can be programs expressed in some language, and instead of giving up a token, we could require this ghost resource program to take a step each time the concrete program does. If the program used as the ghost resource is guaranteed to terminate, then so too must the concrete program. To adapt this idea to the probabilistic setting, we could again require step-wise couplings between the ghost program and the concrete program. However, instead of the form of coupling that was used in Chapter 5, which always permitted a "dummy" step in which no reduction actually happened in the ghost monadic code, we could develop some stricter notion.

7.3.4 Stronger Specifications

Whenever one carries out proofs in a program logic, one must scrutinize exactly what has been proved. This is especially so when verifying a data structure, rather than a whole program, because one has to consider whether the resulting specification is sufficient to reason about a client using the data structure. For this reason, we have used the specifications in Chapter 6 to verify some simple clients. Of course, it is almost certainly true that what we have proved is not sufficient for all clients. Some stronger specifications could be proved, but others may lie beyond the scope of what is expressable in Polaris. We now consider some limitations and possible extensions.

First, the system presented in Chapter 5 only allows us to establish a single coupling at a time. Naturally, if a client program used several probabilistic components (say, several distinct approximate counters and a skip list), we might want to conclude properties for all of them at once. It would be straightforward to index the $\mathsf{Prob}(\mathcal{I})$ assertion by an index i, and then allow different couplings to be established for each of the different indices, separately.

However, a more challenging situation arises when one wants to reason about the combination of two interdependent probabilistic data structures. For instance, suppose we had a probabilistic skip list with approximate counters tracking the number of nodes in the top and bottom lists, and modified the code so that if the counters ever reported that too many nodes were in the top list, a thread would acquire a global lock and "re-balance" the skip list. We might then want to bound the probability that a re-balance operation occurs. This would require reasoning about both the probability that the skip list became too skewed, and the probability that the counter estimation is off, triggering a spurious re-balancing. We would want to reason

[4]The primitive in the language considered by Ngo et al. [92] is a probabilistic if-statement, but the idea is the same.

about these two probabilities separately as much as possible, not unnecessarily re-doing parts of the analysis of these two data structures. Unfortunately, this does not seem possible with the logic we have presented, even if we make the extension described in the previous paragraph. Part of the problem seems to be that many probabilistic properties and analyses are inherently non-compositional. For instance, even something as simple as the expected value of a product of two random variables is not equal to the product of their expectations, unless they are independent. For this reason, it is not so common to see the analysis of "whole systems" composed of many interacting probabilistic algorithms.

The reader may find the previous paragraph disappointing. After all, isn't the point of separation logic to achieve composable proofs and local reasoning? If probabilistic reasoning is not so compositional to begin with, why do we need separation logic? To answer those questions, I would argue that even if compositional reasoning about the *probabilistic behaviors* of interacting randomized data structures is not so common or very feasible, separation logic still provides benefits. For example, it enables us to give specifications like the ones in Chapter 6, so that verification of a client using a skip list does not need to consider the internal representations of the two sublists. Moreover, the implementations of randomized data structures may themselves use several simpler concurrent data structures like queues as subcomponents, and we would therefore want to re-use proofs of these components.

An additional limitation of the specifications proved in Chapter 6 is that they essentially only apply to "stable" states of the data structure: we can only make conclusions about expected values of the counter when we can guarantee there are no on-going concurrent writes; similarly, bounds on the number of comparisons when searching in the skip list can only be obtained when we can prove there are no concurrent insertions. These results could be strengthened somewhat, *e.g.*, by proving that the number of comparisons to find a key k which is already in the list is non-decreasing as other keys are added. However, informally, one would like the variable n appearing in the bound on the expected number of comparisons in (6.3) to be the number of keys less than k *at the time the search was done.* But the number of keys in the list during the search is not entirely well defined if there are concurrent insertions. Instead, we might try to consider the number of keys after the current "batch" of insertions is done and there is some gap before additional insertions begin. The idea of considering groups of operations separated by gaps of time occurs in the analysis of non-probabilistic concurrent data structures, and there is a weaker condition than linearizability known as *quiescent consistency* whose definition is given in terms of such gaps. Sergey et al. [108] have developed ways of reasoning about quiescent consistent data structures in a separation logic, and it might be possible to adapt their approach to the probabilistic setting.

Different results from the theory of couplings and variants of couplings have been used to extend pRHL [14, 15, 16, 59]. It would be interesting to see to what extent these different kinds of couplings can be defined in the presence of non-deterministic choice and used in Polaris. Aguirre et al. [3] have shown that a kind of step-indexed model can be used to reason about more general kinds of couplings (so-called "shift couplings"). Because Polaris is step-indexed, it might be possible to adapt these ideas to reason about shift-couplings in Polaris.

Stronger bounds on probabilistic behavior could be obtained by taking more advanced results from probability theory and seeing how they can be applied to non-deterministic probabilistic computations, as was done in §2.4. Of course, to keep everything machine checked, such

results would have to be mechanized, but by now a number of important results in probability theory have been formalized in various theorem provers [2, 8, 58, 60].

Bibliography

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. (cited on p. 108).

[2] Reynald Affeldt and Manabu Hagiwara. Formalization of Shannon's theorems in SSReflect-Coq. In *ITP*, pages 233–249, 2012. (cited on p. 112).

[3] Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. Relational reasoning for markov chains in a probabilistic guarded lambda calculus. In *ESOP*, pages 214–241, 2018. (cited on p. 111).

[4] Andrew Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001. (cited on p. 47).

[5] Maya Arbel and Hagit Attiya. Concurrent updates with RCU: search tree as an example. In *PODC*, pages 196–205, 2014. (cited on p. 3).

[6] James Aspnes and Eric Ruppert. Depth of a random binary search tree with concurrent insertions. In *DISC*, pages 371–384, 2016. (cited on pp. 3, 4, 6, and 7).

[7] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009. (cited on p. 25).

[8] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *J. Autom. Reasoning*, 59(4):389–423, 2017. (cited on p. 112).

[9] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999. (cited on p. 108).

[10] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *Mathematics of Program Construction (MPC)*, pages 1–6, 2012. (cited on pp. 2, 16, and 105).

[11] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. Relational reasoning via probabilistic coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 387–401, 2015. (cited on pp. 37, 38, and 79).

[12] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 55–67, 2016. (cited on p. 106).

[13] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. A program logic for union bounds. In *ICALP*, pages 107:1–107:15, 2016. (cited on pp. 2, 15, and 21).

[14] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving uniformity and independence by self-composition and coupling. In *LPAR*, 2017. (cited on pp. 16, 37, and 111).

[15] Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. *-liftings for differential privacy. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 102:1–102:12, 2017. (cited on p. 111).

[16] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. In *POPL*, pages 161–174, 2017. (cited on pp. 16, 37, and 111).

[17] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. An assertion-based program logic for probabilistic programs. In *ESOP*, pages 117–144, 2018. (cited on pp. 15 and 21).

[18] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving expected sensitivity of probabilistic programs. *PACMPL*, 2(POPL):57:1–57:29, 2018. (cited on p. 106).

[19] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Quantitative separation logic. *CoRR*, abs/1802.10467, 2018. URL `http://arxiv.org/abs/1802.10467`. (cited on pp. 17, 105, 106, 107, and 109).

[20] Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, pages 119–140, Berlin, Heidelberg, 1969. Springer Berlin Heidelberg. ISBN 978-3-540-36091-9. (cited on p. 27).

[21] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004. (cited on pp. 10, 38, and 79).

[22] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. `https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`, 2018. (cited on pp. 41 and 64).

[23] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015. (cited on p. 32).

[24] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005. (cited on p. 59).

[25] 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. (cited on pp. 59 and 89).

[26] Stephen D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. *Electr. Notes Theor. Comput. Sci.*, 158:123–150, 2006. (cited on pp. 2 and 13).

[27] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN 978-0-262-02665-9. URL http://mitpress.mit.edu/books/certified-programming-dependent-types. (cited on p. 25).

[28] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4. (cited on p. 18).

[29] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *ESOP*, pages 176–201, 2016. (cited on p. 109).

[30] Jerry den Hartog and Erik P. de Vink. Verifying probabilistic programs using a Hoare like logic. *Int. J. Found. Comput. Sci.*, 13(3):315–340, 2002. (cited on p. 15).

[31] Luc Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2 (3):597–609, 1992. (cited on p. 8).

[32] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *SPAA*, pages 43–52, 2013. (cited on pp. 5 and 6).

[33] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. (cited on pp. 10 and 51).

[34] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010. (cited on p. 14).

[35] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, 2013. (cited on p. 14).

[36] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 448–475, 2017. (cited on p. 108).

[37] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. Verified analysis of random trees. In *ITP*, 2018. (cited on p. 109).

[38] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010. (cited on p. 3).

[39] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007. (cited on p. 14).

[40] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. (cited on p. 19).

[41] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985. (cited on p. 4).

[42] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN 978-0-521-89806-5. (cited on p. 4).

[43] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 442–451, 2018. (cited on p. 106).

[44] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, pages 388–402, 2010. (cited on p. 14).

[45] N. D. Gautam. The validity of equations of complex algebras. *Archiv für mathematische Logik und Grundlagenforschung*, 3(3):117–124, Sep 1957. (cited on p. 27).

[46] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *ICFP*, pages 2–14, 2011. (cited on pp. 25 and 40).

[47] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. (cited on p. 50).

[48] Michèle Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85, 1982. (cited on p. 109).

[49] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382, 2011. (cited on p. 19).

[50] Jean Goubault-Larrecq. Continuous previsions. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, pages 542–557, 2007. (cited on p. 39).

[51] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994. (cited on p. 18).

[52] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list (brief announcement). In *OPODIS*, 2006. (cited on p. 8).

[53] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990. (cited on p. 18).

[54] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *LICS*, pages 1–12, 2017. (cited on p. 109).

[55] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969. (cited on p. 10).

[56] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011. (cited on p. 107).

[57] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013. (cited on p. 109).

[58] Johannes Hölzl and Armin Heller. Three chapters of measure theory in Isabelle/HOL. In *ITP*, pages 135–151, 2011. (cited on p. 112).

[59] J. Hsu. Probabilistic Couplings for Probabilistic Reasoning. *ArXiv e-prints*, October 2017. (cited on p. 111).

[60] Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, Cambridge University, May 2003. (cited on p. 112).

[61] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In *ESOP*, 2012. (cited on p. 14).

[62] Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *J. Comput. Syst. Sci.*, 16(3):301–322, 1978. (cited on p. 3).

[63] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983. (cited on pp. 14 and 17).

[64] 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 *POPL*, pages 637–650, 2015. (cited on pp. 2, 14, 20, 41, 59, and 67).

[65] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, 2016. (cited on pp. 14, 41, and 67).

[66] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Accepted for publication in JFP*, 2018. (cited on pp. 41, 42, 46, 47, 48, 55, 67, and 72).

[67] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP*, volume 74 of *LIPIcs*, pages 17:1–17:29, 2017. (cited on p. 108).

[68] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *ESOP*, pages 364–389, 2016. (cited on p. 15).

[69] Gary D. Knott. *Deletion in Binary Storage Trees*. PhD thesis, Stanford University, May 1975. (cited on p. 3).

[70] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981. (cited on p. 77).

[71] Dexter Kozen. A probabilistic PDL. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 291–297, 1983. (cited on pp. 2 and 15).

[72] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, pages 696–723, 2017. (cited on pp. 14, 41, and 67).

[73] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017. (cited on p. 65).

[74] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. In *ICFP*, 2018. (cited on p. 65).

[75] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*, pages 218–231, 2017. (cited on pp. 14, 79, and 106).

[76] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, and Lars Birkedal. Aneris: A logic for node-local, modular reasoning of distributed systems, 2018. (cited on p. 108).

[77] Leslie Lamport. Composition: A way to make proofs harder. In *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, pages 402–423, 1997. (cited on p. 18).

[78] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X. (cited on p. 18).

[79] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, pages 65:1–65:10, 2014. (cited on p. 109).

[80] T. Lindvall. *Lectures on the Coupling Method*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002. ISBN 9780486421452. (cited on pp. 16 and 37).

[81] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4. (cited on pp. 17 and 108).

[82] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992. ISBN 978-3-540-97664-6. (cited on p. 18).

[83] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN 978-0-387-94459-3. (cited on p. 18).

[84] Annabelle McIver, Tahiry M. Rabehaja, and Georg Struth. Probabilistic rely-guarantee calculus. *Theor. Comput. Sci.*, 655:120–134, 2016. (cited on pp. 17, 20, 105, and 107).

[85] Michael W. Mislove. Nondeterminism and probabilistic choice: Obeying the laws. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, pages 350–364, 2000. (cited on pp. 39 and 40).

[86] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996. (cited on pp. 2, 15, and 106).

[87] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21 (10):840–842, 1978. (cited on p. 4).

[88] Hiroshi Nakano. A modality for recursion. In *LICS*, 2000. (cited on p. 49).

[89] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. (cited on p. 25).

[90] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014. (cited on p. 14).

[91] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014. (cited on p. 3).

[92] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*, pages 496–512, 2018. (cited on pp. 109 and 110).

[93] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999. (cited on p. 12).

[94] P.W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1):271–307, 2007. (cited on pp. 2, 13, and 53).

[95] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Analysis of the expected search cost in skip lists. In *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 11-14, 1990, Proceedings*, pages 160–172, 1990. (cited on p. 8).

[96] Matthew J. Parkinson. The next 700 separation logics - (invited paper). In *VSTTE*, pages 169–182, 2010. (cited on pp. 2, 14, and 21).

[97] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types. (Inductive Definitions in Type Theory).* 1996. URL https://tel.archives-ouvertes.fr/tel-00431817. (cited on p. 22).

[98] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *POST*, pages 53–72, 2015. (cited on p. 25).

[99] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997. (cited on p. 19).

[100] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990. (cited on p. 7).

[101] Lyle Harold Ramshaw. *Formalizing the Analysis of Algorithms.* PhD thesis, Stanford University, 1979. (cited on p. 15).

[102] Robert Rand and Steve Zdancewic. VPHL: A verified partial-correctness logic for probabilistic programs. *Electr. Notes Theor. Comput. Sci.*, 319:351–367, 2015. (cited on p. 15).

[103] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002. (cited on pp. 1 and 12).

[104] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu. Formal verification of higher-order probabilistic programs. *CoRR*, abs/1807.06091, 2018. URL http://arxiv.org/abs/1807.06091. (cited on p. 109).

[105] Roberto Segala. *Modeling and verification of randomized distributed real-time systems.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. URL http://hdl.handle.net/1721.1/36560. (cited on pp. 17, 18, and 108).

[106] Roberto Segala. The essence of coin lemmas. *Electr. Notes Theor. Comput. Sci.*, 22:188–207, 1999. (cited on pp. 17 and 18).

[107] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995. (cited on p. 17).

[108] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110, 2016. (cited on p. 111).

[109] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010. (cited on p. 108).

[110] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. A separation logic for fictional sequential consistency. In *ESOP*, pages 736–761, 2015. (cited on p. 108).

[111] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003. (cited on p. 108).

[112] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. A separation logic for a promising semantics. In *ESOP*, pages 357–384, 2018. (cited on p. 108).

[113] Wouter Swierstra. A Hoare logic for the state monad. In *TPHOLs*, pages 440–451, 2009. (cited on p. 25).

[114] Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *CoRR*, abs/1802.02951, 2018. URL http://arxiv.org/abs/1802.02951. (cited on p. xi).

[115] Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*, pages 909–936, 2017. (cited on pp. 14, 79, 106, and 109).

[116] Iris Team. Iris 3.0 documentation, 2017. URL http://plv.mpi-sws.org/iris/appendix-3.0.pdf. (cited on pp. xi, 41, and 67).

[117] The Coq Development Team. The Coq proof assistant, version 8.7.2, February 2018. URL https://doi.org/10.5281/zenodo.1174360. (cited on p. 22).

[118] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runst. *PACMPL*, 2(POPL):64:1–64:28, 2018. (cited on p. 72).

[119] Regina Tix, Klaus Keimel, and Gordon D. Plotkin. Semantic domains for combining probability and non-determinism. *Electr. Notes Theor. Comput. Sci.*, 222:3–99, 2009. (cited on pp. 39, 40, and 108).

[120] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*, 2011. (cited on p. 108).

[121] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. Efficient training of LDA on a GPU by mean-for-mode estimation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 59–68, 2015. (cited on p. 91).

[122] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013. (cited on pp. 14 and 79).

[123] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707, 2014. doi: 10.1145/2660193.2660243. URL http://doi.acm.org/10.1145/2660193.2660243. (cited on p. 108).

[124] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007. (cited on p. 14).

[125] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *OOPSLA*, 2013. (cited on p. 108).

[126] Eelis van der Weegen and James McKinna. A machine-checked proof of the average-case complexity of quicksort in Coq. In *TYPES*, pages 256–271, 2008. (cited on p. 25).

[127] Daniele Varacca. The powerdomain of indexed valuations. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, page 299, 2002. (cited on p. 39).

[128] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113, 2006. (cited on pp. 21, 25, 27, 28, 30, 39, 40, and 105).

[129] Hongseok Yang. Relational separation logic. *TCS*, 375(1–3):308–334, 2007. (cited on p. 2).