# *Thesis Proposal*
# Verifying Concurrent Randomized Algorithms

Joseph Tassarotti

August 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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

## Abstract

Concurrency and randomization are two programming features that are notoriously difficult to use correctly. This is because programs that use them no longer behave deterministically, so programmers must take into account the set of all possible interactions and random choices that may occur. A common approach to reasoning about complex programs is to use *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.

Although logics have been developed for relational reasoning for concurrent programs and randomized programs, no logics exist for reasoning about programs that are both concurrent *and* randomized. I propose developing a program logic that supports reasoning about both of these features. Moreover, I argue that such a logic is more than just an ad-hoc combination of features: instead, the ideas of *separation* and *resources*, which are already central to modern concurrency logics, also provide a foundation for probabilistic reasoning.

# Contents

## 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 not much harder than doing a thorough pencil-and-paper proof. In large part, the work involved in these proofs is establishing that the underlying data structures and operations on them have particular properties, rather than showing that the code actually carries out the intended operations. That is, 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. To address this, a long standing research tradition has focused on developing logics and methodologies to make reasoning about effectful programs easier. A common approach to reasoning about complex programs is to use *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.

Perhaps the most famous example of a logic developed for reasoning about effectful programs is *separation logic* [62], 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 formalized an informal principle that programmers rely on all the time, namely that we can reason locally about pieces of code that operate on disjoint parts of program state because they do not interfere with one another. By formulating this principle in a precise way, separation logic makes it easier to reason about programs with state, particularly those that use pointers or references. Yang [70] 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.

Subsequent work has developed logics for various other effects. A common criticism [59] of much of the research in this area is that to reason about each new kind of effect, a whole new logic is developed, often with rather different soundness proofs and logical mechanisms. Actually, sometimes the situation is even worse than that: each new way of using one particular kind of effect seems to require developing an entirely new logic.

We can see this pattern in the work addressing two kinds of effects that are rather difficult to reason about: concurrency and randomization. O'Hearn [57] and Brookes [15] showed that separation logic could be naturally used to reason about concurrent programs. Since then, a vast number of papers and logics have been developed to handle reasoning about more and more aspects of concurrent programming. In a separate line of work, various extensions to Hoare logic and Dijkstra's weakest precondition calculus have been developed for randomized imperative programs [8, 9, 46, 50]. Many of these logics codify different reasoning patterns in probability theory such as arguments based on repeated use of the union bound, or are focused on establishing one particular property of a randomized algorithm such as expected run time.

On the concurrency side, recent research has tried to unify many previous concurrency logics. Jung et al. [40] argue convincingly that many features developed for reasoning about concurrency can be encoded in a logic built on simpler foundations. I am not aware of a similar effort in the setting of probability logics, but it is reasonable to suggest that at least some of the features of logics in the literature could be partially unified.

However, as I discuss in further detail later on, there are important algorithms that make use of *both* concurrency and randomness. For example, work-stealing scheduling [14], which is used in the runtimes of parallel languages like Cilk [25], uses randomness to efficiently schedule parallel computations and involves fine-grained concurrent interactions between threads. Moreover, the correctness proofs for these algorithms are notoriously complex: the refined analysis of work-stealing by Arora et al. [3] is over twenty pages long, and the acknowledgments thank another researcher for finding a bug in an early version. Work-stealing scheduling is not unique in this regard: the correctness proof for a concurrent binary search tree by Ellen et al. [22] is about thirty pages long.

Because of this complexity, it would be useful to have mechanized versions of these proofs. To make this feasible, we need a logic that combines features from both concurrency logics and probabilistic logics. In this document, I suggest that it is possible to develop such a logic and propose to do so in order to defend the following thesis:

**Thesis.** *Separation logic is a foundation for formal verification of the correctness and complexity of concurrent randomized programs.*

In the remainder of this document I suggest why this proposal is feasible and worth pursuing. First, I further motivate why a program logic for both concurrency and randomness would be useful by describing in more detail a number of concurrent algorithms where randomization plays a role (§2). Next, I describe some prior work on concurrent separation logic and logics for reasoning about probabilistic algorithms (§3). Then, I describe how ideas from these two lines of work could be unified in a single logic (§4). Finally, I outline a plan for carrying out the development of such a logic and argue why my past work prepares me to do so (§5).

## 2 Concurrent Randomized Algorithms

In this section, I describe 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. Before describing these algorithms in detail, let me draw a distinction with 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, since 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 – indeed, this example is already beyond the capabilities of prior work on program logics – but in the examples that follow, there is a more fundamental interaction between concurrency and randomness.

### 2.1 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 $O(\log n)$. Let us write $H_n$ for the random variable giving the height of a tree of $n$ elements generated in this manner. Then in fact, with very high probability, $H_n$ will be $O(\log n)$. To be

[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 features of the language.

precise, Devroye [17] proved that for all integers $k > \max(1, \log n)$,

$$Prob\,[H_n \geq k] \leq \frac{1}{n}\left(\frac{2e\log n}{k}\right)^k$$

For instance, this means that the probability that a tree with 1 million nodes has height greater than $100$ is less than $4 \times 10^{-19}$. 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 situation is 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. [22] 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 [2, 55]. Depending on the number of threads and the work-load, non-balancing trees can perform better than balancing ones [2].

This raises new interest in properties of non-balancing binary search trees. However, as Aspnes and Ruppert [4] 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.

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 [4] 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 [4] 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.

[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 [39, 45].

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

## 2.2 Approximate Counters

Approximate counters are another algorithm with renewed relevance in large scale concurrent systems. They were originally proposed by Morris [51] 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 since 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 in which, if the current value stored is $x$, then with probability $\frac{1}{2^x}$ we update the value to $x + 1$ (in effect, doubling our estimate of the count) and with probability $1 - \frac{1}{2^x}$ leave 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 $E[2^{C_n}] = n+1$. 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 [23] 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 [24].

Morris's counters may seem relatively unimportant today when even cell phones commonly 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. [18] 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. [18] 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.

For concreteness, I give a simplified version of the non-adaptive increment function for one of their proposals in Figure 1. I ignore overflow checking for simplicity. The function starts by generating $64$ bits uniformly at random and storing them in `rbits`. It then enters a loop in which it reads the current value of the counter into `v`. If the current value is $k$, it then checks whether the first $k$ bits of `rbits` are $0$, which occurs with probability $\frac{1}{2^k}$. If so, it attempts to increment the counter by atomically updating it to `v+1` using a CAS, and otherwise it returns. If the CAS returns the previously observed value in the counter, this means the CAS has succeeded and no other thread has done an increment in between, so again the code returns. If the CAS fails

```
void increment(unsigned int* count)
{
 unsigned long rbits = rand64();
 while(true)
 {
  unsigned int v = *count;
  unsigned int mask = (1<<v)-1;
  if(mask & rbits == 0)
  {
   if(CAS(count, v, v+1) == v)
    return;
  }
  else return;
 }
}
```

Figure 1: Simplified version of Dice et al. [18] counter.

it repeats.

The code does not generate a new random number if the CAS fails. Although Dice et al. [18] 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 [4]. Imagine $c$ threads are attempting to concurrently perform an increment, and the scheduler lets them each generate their random value of `rbits` and then pauses them. Suppose the current value in the counter is $k$. Some of the threads may have drawn values for `rbits` 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. In practice, Dice et al. [18] show on a number of benchmarks that the relative errors for their algorithms are less than $3\%$.

In Figure 2, I present an even simpler version of a concurrent approximate counter: here, the thread draws a random number *after* reading the current value in the counter. Moreover, if the random value indicates to go ahead and perform an increment, the thread performs a CAS but *does not try again* if the CAS fails. This too is susceptible to bias from the scheduler, since if two thread concurrently try to increment, it's possible that the counter value will only grow by at most one. But, if the current value in the counter is already large though, it's quite unlikely that two concurrent threads will *both* try to do a CAS. Hence, if the number of threads $c$ is small, we can use an adaptive algorithm in which we first use an exact counter until a certain value is reached, and then switch to this approximate variant. In particular, for $c = 2$ one can show that such an adaptive scheme has a relative bias of at most $1\%$.

```
void increment(unsigned int* count)
{
  unsigned int v = *count;
  unsigned long rbits = rand64();
  unsigned int mask = (1<<v)-1;
  if(mask & rbits == 0)
    CAS(count, v, v+1);
}
```

Figure 2: Alternative approximate counter.

## 2.3 Work-Stealing Schedulers

A third example of a concurrent randomized algorithm arises in the implementation of parallel languages like Cilk [25]. In these languages, programmers do not specify directly how to assign computation tasks to run on a set of processors. Instead, they simply specify what computations may be done in parallel, and then the language run-time has a scheduler[4]

Roughly speaking, the scheduler should try to maximize the number of processors that are working on the computation. Sometimes there will not be enough parallel tasks to keep all processors busy, while at other times there will be more parallel tasks than processors. Because the amount of parallelism is a dynamic property of the program, we cannot compute an assignment schedule offline. Therefore, it is important for the scheduler to be as efficient as possible so that scheduling does not add much overhead.

Although there are *global* greedy scheduling algorithms that compute an asymptotically optimal schedule, they are not used in practice precisely because of this overhead. Instead, a decentralized *work-stealing* scheduler is used [14]. With the work-stealing approach, each processor has a queue of work to compute. As it executes items from this queue, they may in turn generate new items. In particular, when the processor executes a part of the program that creates parallel tasks, it will select one of them to continue processing and push the others onto the queue.

At some point, a processor may complete all the items in its queue. When that happens, the processor tries to *steal* work from the queue of a randomly selected processor. Since the victim of this steal attempt could itself be accessing its queue, and other threads may simultaneously be trying to steal from the same victim, the code implementing operations on the queue must involve some synchronization mechanisms. Because locking access to the queue would have too much overhead, a number of more fine-grained concurrent implementations have been proposed (e.g. [25, 31, 52]).

In practice, this scheme has much lower overhead than the greedy scheduler. But because the victim of a steal attempt is randomly selected, it is possible that the thief may get unlucky and choose a processor which in fact also has no work. In this case, the thief will try again by selecting a new victim, but this means that it will have wasted time not performing useful

[4]Unfortunately there is a terminology conflict here. In the implementation of such schedulers, there are non-deterministic effects arising from the concurrent interaction of processors. In the description of the previous two algorithms, I have been referring to the adversary that resolves these non-deterministic choices as the scheduler, whereas now I am describing a particular kind of scheduler.

computation.

Despite this possibility, Blumofe and Leiserson [14] proved that the work-stealing strategy gives the same asymptotically optimal run-time as the global greedy scheduler *in expectation*. Their analysis, subsequently refined by Arora et al. [3], involves an abstract description of the execution of the algorithm: they speak of a queue with work nodes that are passed between threads atomically during a steal, rather than the fine-grained concurrent implementations of these queues. This analysis involves a certain kind of “balls into bins” game, a common model used in the study of load balancing algorithms and hash tables [7].

## 3 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 actual 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.

### 3.1 Hoare Logic

Most program logics for imperative programs are extensions to or otherwise based on Hoare logic [33]. 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. 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, so long as the postcondition we prove for $e_1$ matches the precondition needed by $e_2$.

Subsequently, Benton [12] 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" [19]. 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 document, I will generally present things using Hoare-style judgments, but many 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.

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 will often omit the binder so that it resembles the traditional Hoare triples. The sequencing rule then becomes a "let" rule:

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

## 3.2 Separation Logic

The language considered by Hoare in his original work was rather limited: it lacked the ability to have references or pointers between memory cells, which are needed for representing various mutable data structures such as linked lists and trees.

Unfortunately, adding these features poses several challenges. To see why, imagine we extend the logic with a primitive assertion $p \hookrightarrow v$, 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:

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

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:

$$\{\mathsf{list}(p_1, l_1) \wedge \mathsf{list}(p_2, l_2) \wedge \mathsf{disjoint}(p_1, p_2)\}\ \mathsf{append}(p_1, p_2)\ \{\mathsf{list}(p_1, l_1 +\!\!+\ l_2)\} \tag{1}$$

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:

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

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. We might try to use the rule of consequence to fix this, since the precondition here certainly implies the precondition of 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:

$$\frac{\{p \hookrightarrow v\}\ p := v'\ \{p \hookrightarrow v'\}}{\{p \hookrightarrow v \wedge p \hookrightarrow v\}\ p := v'\ \{p \hookrightarrow v' \wedge p \hookrightarrow v\}}$$

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, Yang in an extension to Hoare logic called *separation logic* [62], in which the assertions are a form of the substructural logic of Bunched Implications developed by O'Hearn and Pym [56]. 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, since 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:

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

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 $R$.

Unlike normal conjunction, $P \not\vdash P * P$, since 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.

## 3.3 Concurrent Separation Logic

O'Hearn [57] realized that this 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 $e_1 \ || \ e_2$ for the concurrent composition of $e_1$ and $e_2$, he proposed the following rule:

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

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 [15] 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 §2, use these basic atomic operations directly instead of higher-level primitives like locks. Since the initial work by Brookes and O'Hearn, a number of logics have been developed that incorporate support for reasoning about these more fine-grained uses of concurrency (e.g. [20, 26, 38, 54, 58, 68], among many others).

Parkinson [59] 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. Dinsdale-Young et al. [21] 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. [40] in a logic called Iris [41, 47], 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. [67] and subsequently used and extended in [48, 65] 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"). Because this encoding is key to what I propose in §4, I will describe how it works. One starts by defining an assertion $\mathsf{source}(\mathrm{i}, \mathrm{e})$ which says that in the source program execution, thread identifier $i$ is executing the expression $e$. Then there are assertions like $p \hookrightarrow_{\mathsf{s}} v$ which describe the source program's heap. Finally, one has rules for simulating steps of that source program, such as:

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

which executes an assignment in the source program, where the $\Rrightarrow$ connective, called a "viewshift", 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.

## 3.4 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 [61] 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.

Kozen [46] 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. [50] 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. Kaminski et al. [43] extend this logic with the ability to count the number of steps taken by the program so as to derive expected time bounds.

An alternative approach is to not introduce probabilities at the level of assertions at all. Barthe et al. [9] 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 while this system may be less expressive, it is easier to use for certain examples.

Another system that avoids probabilistic assertions is Probabilistic Relational Hoare Logic (pRHL) [8, 10]. This 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 $\mu_1$ and $\mu_2$, respectively, then $(\mu_1, \mu_2) \in L(Q)$.

It is not immediately obvious why knowing that $\mu_1$ and $\mu_2$ are related by $L(Q)$ is useful. The key is that the existence of a joint distribution $\mu$ satisfying the above three conditions can often tell us useful information about the two distributions $\mu_1$ and $\mu_2$. 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:

$$\forall r, \mathit{Prob}\,[X \geq r] \ \geq\ \mathit{Prob}\,[Y \geq r]$$

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 $\mathit{Prob}\,[Y \geq r]$, it suffices to bound $\mathit{Prob}\,[X \geq r]$. 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]:

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

where $\mu_1$ and $\mu_2$ 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 $\forall x \in A_1.\, \mu_1(x) = \mu_2(f(x))$.

Barthe et al. [11] noted that the quadruples of pRHL imply the existence of a *coupling* between the distributions of states of $e_1$ and $e_2$, a well-known proof technique in probability theory. This observation explains why pRHL is useful for establishing certain results, and suggested further uses and extensions to the logic. pRHL and variants of it have been used to verify the correctness of certain differential privacy procedures.

Despite some of the impressive results established using these logics, to the best of my knowledge they have not been used to verify properties about non-trivial data structures like search trees or skip lists. Moreover, the languages under consideration are often rather limited: they do not have higher-order functions, data-types other than numbers and booleans, or allocation of memory, and generally the only means of iteration are while loops. In the next section, I propose the development of a logic for a language that would include these features, as well as concurrency, by incorporating probabilistic reasoning into concurrent separation logic.

[5]The version of this rule in the work by Barthe et al. [8] is different because their $\mathsf{draw}$ command mutates an assignable rather than returning a value. For consistency with the rest of this section, I describe a simplified rule for an expression based language.

# 4 Combining Concurrent and Probabilistic Reasoning

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

In this section, I describe the design of a logic that would make the formal verification of the examples from §2 feasible. I propose to develop this logic for my dissertation in order to defend the thesis statement from the introduction. Since the proposed work is to develop the logic, by necessity I cannot yet present it in detail here. Instead, I start by describing some choices about what features and ideas such a logic should be based on. I then sketch an approach to handling these features by modifying the semantic model of Iris, building on earlier work I have done extending Iris. Finally, I show how a proof about approximate counters might be done in such a logic.

## 4.1 Design

I believe that such a logic should be based on concurrent separation logic. 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. In particular, my goal would be to build on Iris [40], either by finding some way to encode my logic directly in Iris, as has been done for logics for weak memory concurrency [42], or by developing a small set of extensions to the model of Iris, as I have done with my collaborators for another program logic [65].

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.

The first question is whether the language of assertions should be interpreted probabilistically, as in [43, 46, 50], or whether there should be particular assertions or judgments for speaking about probabilities as in [8, 9, 61]. Regardless of any merits of the former approach, I believe there are serious issues with trying to develop such a "quantitative" model of assertions for a separation logic. It seems hard to reconcile an approach that interprets assertions as numbers representing the probability that they are true with the separation logic view of assertions as resources that can be transferred between threads: What does it mean to say that with probability 1/2, $p \hookrightarrow v$, and what does that license the "owner" of this resource to do? Even if a useful notion of probabilistic ownership could be given, the existing semantic models of logics like Iris are rather sophisticated, using constructions in the category of complete bisected ultrametric spaces [13] to solve recursive domain equations in order to support certain impredicative features. It would be hard to adapt these models to a quantitative view of assertions. Although there is on-going work on developing categorical models for the semantics of higher-order probabilistic programs which might be useful in this regard [30, 32, 37], I think there are still too many open questions to make it feasible to consider this direction. For that reason, I will pursue the latter approach.

The second question is to what extent the logic should incorporate explicit reasoning about probabilities. On one end, work in the style of Ramshaw [61] features explicit assertions of the form $\mathsf{Fr}(P) = p$ which says that $P$ holds with "frequency" $p$. Hence, one may reason about probabilities explicitly throughout the verification. Alternatively, in Barthe et al. [9] probabilities appear only as annotations in judgments of the form $\vdash_p \{P\}\ e\ \{Q\}$, where $p$ is some probability value that is updated according to the rules of the logic. On the other end is a relational logic like pRHL in which no probabilities appear explicitly, and the only time probabilistic reasoning enters is "locally" in the premise of the rule for reasoning about when the two programs draw a random sample.

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.**

   If we look at the analysis of concurrent binary trees by Aspnes and Ruppert [4], the authors analyze a mathematical abstraction of the algorithm. They talk about an adversary playing a probabilistic game where it gets to affect the order that nodes are inserted into the tree, rather than a scheduler preempting the execution of a thread at a particular moment. Similarly, the analysis of approximate counters given by Flajolet [23] immediately recasts the problem in terms of a simple Markov chain.

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

   When it comes to random variables describing aspects of probabilistic algorithms, there are many properties of interest: the expected value, variance, tail-bounds, rate of convergence to an asymptotic distribution, and so on. Hence, we will almost certainly carry out multiple proofs about such algorithms. However, we do not want to re-prove basic facts each time about the correctness of the algorithm, 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. [9] an attempt at 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 [59] 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 unified synthetic logic.

## 4.2 Encoding in Iris

Having argued in favor of a relational approach in the style of pRHL encoded in Iris, let me now describe tentatively how such a logic could be developed.

### Operational Semantics for Concurrent Randomized Languages

The Iris logic is parameterized by a small-step concurrent operational semantics for a given language that one wants to reason about. Thus, the first step is to give an operational semantics for a concurrent randomized language that is expressive enough to capture the kinds of algorithms I described in previous sections. However, as I have mentioned, the semantics of higher-order probabilistic programming languages can be extremely subtle and is the subject of much ongoing work.

But, by restricting our attention only to discrete distributions and terminating programs, we can avoid these complexities while still being able to analyze the kinds of algorithms described in §2. In Appendix A I describe how to take a small-step semantics for a deterministic language and extend it with concurrency and randomized choice. The resulting semantics specifies a per-thread small step relation $e; \sigma \xrightarrow{p} e'; \sigma'$ which says that from state $\sigma$, expression $e$ steps with probability $p > 0$ to $e'$ and state $\sigma'$. This is then lifted to a concurrent semantics by parameterizing by a scheduler which chooses at each point which thread takes the next step. For a given scheduler, if the execution of a program always terminates without faulting, the semantics induces a discrete probability distribution on terminating values and states.

### Adding Couplings to Iris

Once we have an operational semantics for a probabilistic language, we need to extend Iris with pRHL-like features needed for probabilistic relational reasoning. Recall that the key rule in pRHL was the "coupling rule":

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

where $f \blacktriangleleft \mu_1, \mu_2$ means $\forall x \in A_1.\, \mu_1(x) = \mu_2(f(x))$. That is, when the two programs make a random choice, the premise requires us to exhibit a kind of weighted bijection $f$ between the outcomes of the samples, and subsequently we reason as if whenever the left program drew $x$, the right program drew $f(x)$. One way to interpret this rule is that we are constructing a kind of simulation relation between the two programs that is sensitive to probabilities. (The developers of pRHL mention an explicit connection to research in probabilistic process algebras, where various notions of probabilistic bisimulation have been developed [16]).

To see how something similar might be done in concurrent separation logic, recall the discussion of how relational reasoning was previously done by Turon et al. [67], in which resources representing threads were manipulated using the view-shift connective $\Rrightarrow$. In my prior work [65], we noted that one limitation of the approach of Turon et al. [67] for relational reasoning in concurrent separation logic was that the triple $\{\mathsf{source}(i, e_1)\}\ e_2\ \{v.\, \mathsf{source}(i, v)\}$ only implied something about the relationship between $e_1$ and $e_2$ when $e_2$ terminated, because it was based on a logic of partial correctness. In practice, one often wants to prove an additional property: if $e_2$ diverges, then $e_1$ should also diverge[6].

In order to strengthen the meaning of triples so that they would imply this additional property about non-termination, we extended Iris with an additional connective $P \Rrightarrow Q$ which indicated that *at least* one step of the source program was executed when transforming from resources satisfying $P$ to ones satisfying $Q$. Then the meaning of triples was changed so that whenever the target program took a step, at least one step of the source program had to be simulated. As a consequence, we obtained rules like:

$$\frac{P \Rrightarrow Q}{\{P * x \hookrightarrow v\}\ x := w\ \{Q * x \hookrightarrow w\}}$$

where the premise $P \Rrightarrow Q$ reflects the requirement that a step of the source program be simulated. Then, from a derivation of $\{\mathsf{source}(i, e_1)\}\ e_2\ \{v.\, \mathsf{source}(i, v)\}$, it followed that if $e_2$ diverged, $e_1$ also had a diverging execution.

Semantically, we encoded this by equipping the resource monoids with a binary relation $\curvearrowright$, which represented the abstract idea of "stepping" a resource. Then the definition of the Hoare triple required the resources in the precondition to perform a transition along this relation every time the target program took a step. For the particular resources representing source threads, this $\curvearrowright$ relation was based on the operational semantics of the source language.

Now, in the probabilistic setting, we can instead consider a stepping relation for resources that is weighted by probabilities, just like that of the operational semantics. Then we can change the definition of Hoare triples so that if a target thread is in a state where its set of transitions has distribution $\mu$, we can require that there be a weighted-bijection $f$ from the transitions in $\mu$ and the distribution of transitions that the current resources owned by the thread can make. For such a definition, we can imagine that the following rule would be sound:

$$\frac{f \blacktriangleleft \mu_1, \mu_2}{\{\mathsf{source}(i, \mathsf{draw}(\mu_2))\}\ \mathsf{draw}(\mu_1)\ \{x.\, \mathsf{source}(i, f(x))\}}$$

which would be the appropriate reformulation of the pRHL rule in the concurrent setting.

## Beyond pRHL-style specifications

However, the kind of relational reasoning provided by pRHL does not exactly correspond to the mode of use that I argued for in §4.1. First, I said that what we want to do is establish a relation between the code and a simple mathematical description of a stochastic process, but in pRHL the "specification language" is the same imperative randomized language. We can partially address this in the extension to Iris that I have proposed by simply using a resource monoid that represents source threads expressed in a simpler functional language.

A second issue is that in the above rule, both programs have to be at the point where they are making a random choice that can be coupled. For example, if we have the following two programs:

[6]In fact, concurrent programs can diverge for trivial reasons if the scheduler never assigns certain threads to run, so we considered a notion of fair divergence, in which every thread that does not terminate takes infinitely many steps.

$$\begin{array}{ll} \textsf{let } x = \textsf{draw}(\mu_1) \textsf{ in} \qquad\qquad & \textsf{let } y = \textsf{draw}(\mu_2) \textsf{ in} \\ \textsf{let } y = \textsf{draw}(\mu_2) \textsf{ in} & \textsf{let } x = \textsf{draw}(\mu_1) \textsf{ in} \\ (x, y) & (x, y) \end{array}$$

Then we would like to be able to prove that the distribution of return values are the same. The problem is that if $\mu_1 \neq \mu_2$, then we will not (in general) be able to use the coupling rule because we cannot exhibit a coupling when the left program draws from $\mu_1$ and the right program draws from $\mu_2$. To handle this in pRHL, there is a rule like:

$$\frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2' \qquad \{P\}\ e_1 \sim e_2\ \{Q\}}{\{P\}\ e_1' \sim e_2'\ \{Q\}}$$

which permits swapping expressions in a quadruple for "equivalent" ones. The logic also has various rules for establishing equivalences, which would allow one to permute the order of the samples in the above examples. While this works in the sequential setting, in the concurrent setting with references, it is not obvious when two expressions are equivalent. Even something as simple as swapping the order of two expressions may change program behavior if other threads can access memory modified by the expressions. After all, part of the reason we want to use a program logic with support for relational reasoning is precisely to be able to establish these kinds of equivalences.

Finally, the most obvious way to specify programs in pRHL is not very compositional. To see the problem, suppose we tweak the concurrent binary search tree data structure so that each node in the tree contains an approximate counter which estimates how many children it has. Then, we would like to prove that the shape of the tree is related to the stochastic model proposed by Aspnes and Ruppert [4], and the counts in each node are distributed according to some Markov chain characterizing approximate counters. We would like to re-use some prior proof about approximate counters, but it is not very natural to specify this combined data structure as being generated by some monolithic stochastic process, since the shape of the tree does not depend on the counters.

To address these issues, I propose to make use of the fact that probabilistic choice can be regarded as a monadic operation [28, 60]. In the discrete setting, $\textsf{return}\ x$ is the trivial distribution yielding $x$ with probability 1, and the monadic bind $(x \leftarrow \mu \textsf{ in } F(x))$ is interpreted as the distribution formed by first sampling a value $x$ from $\mu$ and then proceeding as the distribution $F(x)$.

With this in mind, we can define a new assertion of the form $v \leftarrow_i \mu$, where $\mu$ is a distribution, $v$ is some value in the support of that distribution, and $i$ is a kind of identifier used for bookkeeping. Informally, we can read this as saying "$v$ was generated by sampling from $\mu$", where the $i$ is used to distinguish from other occasions in which a sample from distribution $\mu$ is generated. This assertion is used as part of the following rules:

$$\textsf{True} \Rrightarrow \exists i.\, (v \leftarrow_i \textsf{return}\ v) \qquad\qquad \frac{f \blacktriangleleft \mu_2, G(v)}{\{v \leftarrow_i \mu_1\}\ \textsf{draw}(\mu_2)\ \{x.\, f(x) \leftarrow_i (y \leftarrow \mu_1 \textsf{ in } G(y))\}}$$

The first rule is simple, particularly if we pretend the view-shift is just an implication: it says that we can immediately conclude any value $v$ was generated by the degenerate distribution that always returns $v$. The second rule is how we re-express the pRHL coupling rule in terms of this new assertion. As a precondition, we know that $v$ was generated by $\mu_1$. Then, when the physical program proceeds to sample some value $x$ from $\mu_2$, we can conclude that a value $f(x)$ was generated by first sampling some value $y$ from $\mu_1$ and then sampling from $G(y)$, so long as $f$ couples $\mu_2$ and $G(v)$. All the other usual rules from Iris (ideally) carry over unchanged.

How do we model the $v \leftarrow_i \mu$ assertion semantically in Iris? I believe this can be done by considering resource monoids where the elements are distributions, specified in this monadic style, along with a "trace" that records the sequence of samples drawn at each monadic bind step. Then the "stepping" relation on this monoid corresponds to adjoining a new sample to the end of the trace.

We write $\mathsf{init}(i)$ for the assertion $() \leftarrow_i \mathsf{return}\ ()$, which represents a resource that has not yet generated any samples. The soundness statement of such a logic would say (among other things) that if we can prove $\{\mathsf{init}(i)\}\ e\ \{x.\, x \leftarrow_i \mu\}$, then under every scheduler in which $e$ always terminates, its return value is distributed according to $\mu$. The benefit of this reformulation is that we do not have to describe once and for all the entirety of the randomized specification in the precondition.

## Non-determinism and randomized choice

With what has been described so far, however, we cannot yet handle reasoning about the algorithms in §2. Before seeing where the problem arises, first consider the following simple program:

$$\begin{array}{l||l} \mathsf{lock}(l) & \mathsf{lock}(l) \\ r := \mathsf{bernoulli}(p) & r := \mathsf{bernoulli}(p) \\ \mathsf{unlock}(l) & \mathsf{unlock}(l) \end{array}$$
$$!r$$

There are two threads, which each acquire a lock $l$ and draw a sample from a Bernoulli distribution with parameter $p$, and then store this in $r$ before releasing the lock. After both threads finish, we return the contents of $r$.

Call this program $e$. Then, using the usual mechanisms from Iris and the extensions described so far, we can derive $\{\mathsf{init}(i)\}\ e\ \{x.\, x \leftarrow_i \mathsf{bernoulli}(p)\}$, which by the intended soundness result, would say that the return value of $e$ has distribution $\mathsf{bernoulli}(p)$.

However, consider the following variant $e'$:

$$\begin{array}{l||l} r := \mathsf{bernoulli}(p) & r := \mathsf{bernoulli}(p) \end{array}$$
$$!r$$

which does not use locks to control access to $r$. Then it appears to be possible to only prove triples like the following[7] about $e'$:

$$\{\mathsf{init}(i) * \mathsf{init}(i')\}\ e'\ \{x.\, x \leftarrow_i \mathsf{bernoulli}(p) \vee x \leftarrow_{i'} \mathsf{bernoulli}(p)\}$$

[7]Of course, I have not *proved* that certain other triples are underivable.

or

$$\{\mathsf{init}(i)\}\ e'\ \{x.\, \exists v.\, (x, v) \leftarrow_i \mu_1 \vee (v, x) \leftarrow_i \mu_1\}$$

where

$$\begin{aligned}\mu_1 \triangleq\ & y \leftarrow \mathsf{bernoulli}(p)\ \mathsf{in}\\ & z \leftarrow \mathsf{bernoulli}(p)\ \mathsf{in}\\ & (y, z)\end{aligned}$$

neither of which are covered by the simple soundness theorem I mentioned above: in the first, we have that $x$ is distributed according to one of two different identifiers; in the second there is only one identifier, but we do not know which of two values was returned.

In fact, we should not be able to conclude that the return value's distribution is $\mathsf{bernoulli}(p)$, because it is not so for all schedulers. In particular, for instance, the scheduler can bias the return value by running both threads until they have drawn their random variable, then scheduling them so that the maximum of the two will be written last, causing the maximum to be the value read at the end[8]. Nevertheless, the scheduler's power is not limitless, since in the end it has to choose from one of the two Bernoulli values, so for instance, the probability that the final value will be $1$ is still at most $3/4$.

This is the kind of situation that arises in the examples like the concurrent approximate counters and the binary tree. Hence, we need a soundness theorem that can say something (weaker) in these cases as well. The idea is to use a monad that combines both non-deterministic (that is, selection among alternatives by an adversary) and probabilistic choice. There are many kinds of such monads [27, 29, 66, 69]; in our case the construction of Varacca and Winskel [69] seems sufficient. They have shown that a denotational semantics for a simple probabilistic language with non-deterministic choice using their monad is adequate with respect to an operational semantics modeled by an adversarial scheduler. For simplicity, we can think of this monad as yielding a set of distributions, corresponding to different schedulers. Writing $x \,\Box\, y$ for the non-deterministic choice operation (that is, the selection by an adversary) between $x$ and $y$, then the monadic encoding we want to relate $e'$ to is:

$$\begin{aligned}\mu' \triangleq\ & y \leftarrow \mathsf{bernoulli}(p)\ \mathsf{in}\\ & z \leftarrow \mathsf{bernoulli}(p)\ \mathsf{in}\\ & (\mathsf{return}\ y) \,\Box\, (\mathsf{return}\ z)\end{aligned}$$

which says that two random samples are drawn, and then a scheduler gets to pick from between the two. To make use of this non-deterministic choice operation, we have new view-shift rules:

$$(v \leftarrow_i \mu_1) \Rrightarrow (v \leftarrow_i \mu_1 \,\Box\, \mu_2) \qquad\qquad (v \leftarrow_i \mu_2) \Rrightarrow (v \leftarrow_i \mu_1 \,\Box\, \mu_2)$$

The soundness statement of such a logic would then be amended to say that if we can prove $\{\mathsf{init}(i)\}\ e\ \{x.\, x \leftarrow_i \mu\}$, then for each scheduler in which $e$ always terminates, its return value is distributed according to some distribution in $\mu$.

[8]In $e$, the scheduler could not bias things in this way because the random values were drawn after the lock was acquired, at which point the scheduler could no longer re-order the two threads.

A natural question to ask is: what have we gained by relating a program like $e'$ to $\mu'$, since $\mu'$ involves exactly the kind of non-determinism in the original $e'$? Moreover, if Varacca and Winskel [69] have given a denotational semantics for a probabilistic language with choice, why not try to adapt their model to give a denotational semantics directly for our language and use this semantics directly, rather than passing through the program logic?

First, it would be difficult to extend the model of Varacca and Winskel [69] to handle features like higher-order state. Second, even if we had such a model, we will want to recover all the tools from concurrent separation logic for reasoning about the parts of the program that are really about fine-grained concurrent interaction. And, while in the case of $e'$, we indeed have not gained much, in the case of $e$ and $\mu$, $\mu$ is already simpler than the representation we would get directly in the denotational semantics, particularly if the lock is implemented in terms of something like a CAS. In the end, the monadic encoding here only has probabilistic and non-deterministic choice, without other effects like general recursion and state. So, in establishing a connection between a program and one of these monadic encodings, we can indeed simplify the description of its behavior.

Admittedly, the discussion above is not entirely precise, but if we already knew with certainty how to modify the semantics of Iris to encode probabilistic reasoning, there would be little work left to propose. However, I have explained a connection between pRHL and my prior work that suggests it is feasible to develop a program logic with the characteristics I have argued for in this section.

## 4.3 Specifying Approximate Counters

To further explore the proposed logic, I return to the concurrent approximate counters described in Figure 2 of §2, in which the thread does not try again if the CAS fails. Since I do not yet have a full set of sound rules for this logic, I will not yet *prove* the correctness of this algorithm. Rather, the goal of this section is to to show how some of the features I have described make it natural to *specify* the correctness of this algorithm.

In the pencil-and-paper treatments of approximate counters (even in the non-concurrent case), usually one defines $C_n$ to be the random variable giving the value of the counter after $n$ increments, and then studies the properties of $C_n$. But the number of times the increment function is called is an intensional property of the program, not some externally visible behavior. Moreover, the number of times increment is called may itself be a random variable. In general, the specification of concurrent data structures is a subtle issue, and adding randomness only further complicates the matter.

To help gain some intuition about what the specification should be, we can look at how a "normal" concurrent counter is typically specified in a logic like Iris. Figure 3 gives a specification for an exact concurrent counter that can be used by at most two threads concurrently. The triples refer to a predicate $\mathsf{counter}(l, n, v)$. The argument $l$ is a pointer to the counter, while $n$ is the number of times an increment has been done on this counter *using this permission*, and $v$ represents a lower bound on the current value in the counter.

$$\{\mathsf{True}\}\ \mathsf{newCounter}\ ()\ \{l.\, \mathsf{counter}(l, 0, 0) * \mathsf{counter}(l, 0, 0)\}$$

$$\{\mathsf{counter}(l, n, v)\}\ \mathsf{incr}\ l\ \{\mathsf{counter}(l, n + 1, v + 1)\}$$

$$\{\mathsf{counter}(l, n, v)\}\ \mathsf{read}\ l\ \{v'.\, v' \geq v * \mathsf{counter}(l, n, v')\}$$

$$\begin{array}{l}\{\mathsf{counter}(l, n_1, v_1) * \mathsf{counter}(l, n_2, v_2)\}\\ \quad \mathsf{read}\ l\\ \{v.\, v = n_1 + n_2 * \mathsf{counter}(l, n_1, v) * \mathsf{counter}(l, n_2, v)\}\end{array}$$

Figure 3: Specification for exact counter accessible by at most two operations.

The counter has three operations: newCounter, incr, and read. The triple for newCounter says that it returns a pointer $l$ to the new counter, as well as permission to access this new counter in the form of two $\mathsf{counter}(l, 0, 0)$ assertions. Only two are returned because we are considering a counter that can be accessed by only two threads at once.

The incr triple is straight-forward: if we have permission to access a counter at $l$ in the form of a $\mathsf{counter}(l, n, v)$ assertion, then after doing the increment we get back the assertion $\mathsf{counter}(l, n + 1, v + 1)$. The $n + 1$ records the fact that we have used this permission to do an additional increment, and the lower bound is raised to $v + 1$ because even if we do not know the exact value in the counter, it has increased by one.

There are two triples for read, based on whether we have both counter permissions. The first says that if we merely have $\mathsf{counter}(l, n, v)$ and read from the counter at $l$, it returns a value $v'$ which is greater than or equal to $v$. Since the counter value can only be incremented, not decremented, any future reads will have to return a value at least as big as $v'$, so in the postcondition our permission is modified so that the lower bound is now $v'$. Since we only have one permission, some other thread can still be performing other increments, so this *monotonicity* property is all we can hope to say.

In the second read rule, we have the two permissions $\mathsf{counter}(l, n_1, v_1)$ and $\mathsf{counter}(l, n_2, v_2)$. In this case we know that exactly $n_1 + n_2$ increments have been done prior to our read, and no other increments can be done concurrently while reading, so the value we read has to be exactly $n_1 + n_2$.

The specification for the approximate concurrent counter is shown in Figure 4. Now, the triples refer to a predicate $\mathsf{approxCounter}_i(l, n, v)$, where we now index by an identifier $i$ used for coupling.

The triple for newApproxCounter is similar to the exact case, except it consumes as a precondition some initial identifier $i$. The increment rule is also similar, but we do not increase the lower bound $v$: after all, it may be the case that the value in the counter does not change as a result of this increment.

Again, we have two triples for approxRead, based on whether we have both counter permis-

$$\{\mathsf{init}(i)\}$$
$$\quad \mathsf{newApproxCounter}\ ()$$
$$\{l.\ \mathsf{approxCounter}_i(l,0,0) * \mathsf{approxCounter}_i(l,0,0)\}$$

$$\{\mathsf{approxCounter}_i(l,n,v)\}\ \mathsf{approxIncr}\ l\ \{\mathsf{approxCounter}_i(l,n+1,v)\}$$

$$\{\mathsf{approxCounter}_i(l,n,v)\}\ \mathsf{approxRead}\ l\ \{v'.\ v' \geq v * \mathsf{approxCounter}_i(l,n,v')\}$$

$$\{\mathsf{approxCounter}_i(l,n_1,v_1) * \mathsf{approxCounter}_i(l,n_2,v_2)\}$$
$$\quad \mathsf{approxRead}\ l$$
$$\left\{\begin{array}{l} v.\ v \leftarrow_i \mathsf{approxDist}(n_1+n_2) * v \geq \max(v_1,v_2) \\ \quad * \exists i'.\ \mathsf{approxCounter}_{i'}(l,n_1,v) * \mathsf{approxCounter}_{i'}(l,n_2,v) \end{array}\right\}$$

Figure 4: Specification for approximate counter accessible by at most two concurrent threads.

sions. In the first case, when we only have one permission, we read some value $v'$ which is greater than or equal to the current lower bound, and we update the lower bound in the postcondition accordingly. This captures the fact that the approximate counter is still monotone.

For the second rule, the precondition has both of the permissions $\mathsf{approxCounter}_i(l,n_1,v_1)$ and $\mathsf{approxCounter}_i(l,n_2,v_2)$. In this case we know that exactly $n_1 + n_2$ increment operations have been called. Therefore, we conclude that the value read is distributed according to approxDist, which is described in Figure 5. The monadic description of approxDist captures the fact that each increment may happen by itself, or there may be two concurrent increments. In the postcondition, we get back the counter permissions again, but this time with a different identifier: this is because we have given back the $v \leftarrow_i \mathsf{approxDist}(n_1+n_2)$ resource in the other half of the postcondition, so this identifier $i$ cannot be further manipulated in subsequent increments[9].

[9]It should be possible to formulate a stronger specification that allows multiple reads in which we have a conclusion about the distribution, but this one is simple and still captures some essential properties.

$$
\begin{aligned}
\mathsf{approxDist}(n) &\triangleq k \leftarrow \mathsf{approxDistAux}(n)\ \mathsf{in} \\
&\quad\ \mathsf{return}\ 2^k - 1 \\
\mathsf{approxDistAux}(0) &\triangleq 0 \\
\mathsf{approxDistAux}(1) &\triangleq 1 \\
\mathsf{approxDistAux}(n+2) &\triangleq \begin{pmatrix} k \leftarrow \mathsf{approxDistAux}(n+1)\ \mathsf{in} \\ p \leftarrow \mathsf{bernoulli}(1/2^k)\ \mathsf{in} \\ \mathsf{if}\ p\ \mathsf{then} \\ \quad k+1 \\ \mathsf{else} \\ \quad k \end{pmatrix} \square \begin{pmatrix} k \leftarrow \mathsf{approxDistAux}(n)\ \mathsf{in} \\ p_1 \leftarrow \mathsf{bernoulli}(1/2^k)\ \mathsf{in} \\ p_2 \leftarrow \mathsf{bernoulli}(1/2^k)\ \mathsf{in} \\ \mathsf{if}\ p_1 \vee p_2\ \mathsf{then} \\ \quad k+1 \\ \mathsf{else} \\ \quad k \end{pmatrix}
\end{aligned}
$$

Figure 5: Distributions for approximate counter.

# 5 Additional Proposed Work

In the introduction, I proposed the following thesis statement:

**Thesis.** *Separation logic is a foundation for formal verification of the correctness and complexity of concurrent randomized programs.*

The preceding section proposed a possible design of a concurrent separation logic for probabilistic relational reasoning. However, the development of such a logic alone would not be enough to defend my thesis statement: one must demonstrate that the logic is actually useful by verifying examples. I now discuss the additional pieces of work that are needed to make it feasible to verify such examples and mention the examples I hope to verify.

## 5.1 Mechanization of Logic

Many of the modern program logics I have discussed above have some complicated aspects. Not only are their soundness proofs often quite intricate, but the proof rules themselves have subtle side conditions and premises. This makes it difficult to check if they're being used correctly in a pencil and paper derivation!

For that reason, I consider it essential to not only mechanize the soundness proof of the proposed logic, but also to do so in a way that makes it possible to reason *internally* in the logic using a theorem prover.

Fortunately, the Iris project already features a complete mechanization of Iris in Coq and a "proof mode" for constructing derivations in the logic almost as if one were using a specialized proof assistant. I have already mechanized a serious extension to Iris and its proofmode as part of prior work [65], so I believe it will be feasible to mechanize my proposed logic by building on the mechanization of Iris.

## 5.2 Mechanized Probability Theory

The logic I have proposed is for relational reasoning. Its purpose is to show that some abstract model of a program captures its behavior appropriately. In the end, to establish concrete results about the program, one must then actually reason about this abstract model. To do so, it is useful to have mechanized results from probability theory. For example, the analysis of concurrent binary tree depth by Aspnes and Ruppert [4] uses Doob's Optional Stopping Theorem. The textbooks by Motwani and Raghavan [53] and Mitzenmacher and Upfal [49] give an overview of the kinds of results from probability theory that are useful in the analysis of algorithms.

There has been extensive prior work mechanizing probability theory in theorem provers (e.g. [1, 5, 6, 34, 35, 36], among others). However much of this work has been done in theorem provers other than Coq, and since I am building on the formalization of Iris in Coq, I cannot use much of it directly. Moreover, many of the theorems used in the analysis of algorithms have not been mechanized at all. Therefore, I will need to mechanize some of these theorems myself, though I cannot hope to do all of them. My focus will be on the theorems that are most essential to the particular examples and properties I will consider. In unpublished work [63, 64], I have already mechanized a small library for discrete probability, and used it to mechanize and extend a theorem due to Karp [44] that is useful for the analysis of randomized divide-and-conquer algorithms. I have used this result to mechanize high probability bounds for the costs of sequential and parallel Quicksort, the height of binary search trees, and the number of rounds needed in a distributed leader election protocol.

Based on this experience, I expect to be able to mechanize results like Doob's Optional Stopping Theorem without excessive difficulty by restricting my attention to discrete probability theory. This suffices for the algorithms under consideration, and avoids the need to develop extensive results in measure theory.

## 5.3 Examples

Finally, I will use the logic and the library of probability theory results to verify some examples. As a warm-up, I will start by showing that the total number of comparisons performed by a concurrent version of Quicksort, which forks threads to recursively sort parts of the list, is stochastically dominated by the number performed by a purely functional version. As I have mentioned before, this example is not especially interesting because the threads manipulate entirely disjoint parts of the heap, but it would exercise some features of the logic and would connect to my already completed mechanized analysis of purely functional Quicksort.

Subsequently, I would like to verify properties of the concurrent binary search trees and approximate counters that I described in §2.

An interesting "stretch" goal would be to verify the complexity of a work stealing scheduler. However, even simply verifying the functional correctness of such a scheduler would already be quite involved, so it may not be feasible to do the whole analysis as part of this thesis.

## 5.4 Timeline

Tentatively, I envision the following schedule for completing the proposed work:

- **July – October:** Develop extensions to Iris and mechanize soundness proof of logic. Verify simple examples like concurrent Quicksort which do not require additional results in probability theory.
- **November – February:** Verify expected value results for concurrent approximate counters. Begin developing results like Doob's stopping theorem and Azuma-Hoeffding inequality needed for other examples.
- **March – April:** Verify depth bounds of concurrent binary search trees.
- **May – Fall 2018:** Write dissertation.

# Acknowledgments

I thank my advisor and the rest of my committee for their mentorship and for agreeing to evaluate this proposal. I would also like to thank Ralf Jung for innumerable conversations about concurrent separation logic and its semantics. I am grateful to Jean-Baptiste Tristan for suggesting the example of approximate counters and for feedback on these ideas.

This research was conducted in part with U.S. Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a; and with a gift from Oracle Labs. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these organizations.

# Appendices

## A Concurrent Randomized Operational Semantics

In this appendix I describe how to take a small-step operational semantics for a deterministic language and extend it with concurrent threads and probabilistic choice. This is not novel or difficult, and for example, similar semantics are described by [69], but I include it here for completeness.

For simplicity, we only add the ability to draw from the $\mathsf{bernoulli}(\frac{1}{2})$ distribution as a primitive, but additional primitive distributions could also be added. We start with a small-step semantics specified by:

$$e; \sigma \to e'; \sigma'$$

which says that from state $\sigma$, expression $e$ steps to $e'$ with updated state $\sigma'$. Next, we extend the language with an expression $\mathsf{flip}$. We define a judgment $e; \sigma \xrightarrow{p} e'; \sigma'$ as follows:

$$\frac{e; \sigma \to e'; \sigma}{e; \sigma \xrightarrow{1} e'; \sigma} \qquad \mathsf{flip}; \sigma \xrightarrow{\frac{1}{2}} 1; \sigma \qquad \mathsf{flip}; \sigma \xrightarrow{\frac{1}{2}} 0; \sigma$$

(I assume here that the original language has expressions for integers $0$ and $1$). This judgment says that from state $\sigma$, expression $e$ steps with probability $p > 0$ to $e'$ and state $\sigma'$. When $p$ is 1, we simply write $e; \sigma \to e'; \sigma'$, because then the semantics coincides with the original deterministic one. We assume that for all $e; \sigma$ and $e'; \sigma'$, there is at most one $p$ such that $e; \sigma \xrightarrow{p} e'; \sigma'$. We write $\mathsf{pstep}(e; \sigma, e'; \sigma')$ for the function which is $p$ if $e; \sigma \xrightarrow{p} e'; \sigma'$ and $0$ otherwise. For each $\sigma$ and $e$ we have that either for all $e'$ and $\sigma'$, $\mathsf{pstep}(e; \sigma, e'; \sigma') = 0$, indicating $e; \sigma$ is stuck and cannot reduce (signifying an error), or else:

$$\sum_{e', \sigma'} \mathsf{pstep}(e; \sigma, e'; \sigma') = 1$$

This means that if the thread is not stuck, the reduction relation induces a distribution on the set of configurations which it can step to. With the exception of the $\mathsf{draw}$ primitive, the per-thread semantics are otherwise deterministic.

For concurrency, we extend the language with expressions of the form $\mathsf{fork}\{e\}$. We then lift this semantics to a relation between pairs consisting of a *list* of threads and a state, indexed by which thread in the list takes a step:

$$\frac{e_i; \sigma \xrightarrow{p} e_i'; \sigma'}{[e_1, \dots, e_i \dots, e_n]; \sigma \xrightarrow[i]{p} [e_1, \dots, e_i' \dots, e_n]; \sigma'} \qquad \frac{e_i = K[\mathsf{fork}\{e_{\mathrm{f}}\}]}{[e_1, \dots, e_i, \dots, e_n]; \sigma \xrightarrow[i]{1} [e_1, \dots, K[()], \dots, e_n, e_{\mathrm{f}}]; \sigma}$$

The first rule says that if the $i$th thread can perform a step with some probability $p$, then the list of threads can perform the corresponding step with probability $p$. The second rule says that if the $i$th thread is equal to $K[\mathsf{fork}\{e_f\}]$ for some evaluation context $K$ and this thread steps, then with probability $1$ a new thread is created with expression $e_f$ and the value $()$ is returned in the evaluation context.

We call a pair of a list of threads and a state a *configuration*. A configuration is said to have terminated if every thread is a value. A configuration $[e_1, \dots, e_n]; \sigma$ is *safe* if each $e_i$ is either a value or can take a step. Given configurations $R$ and $R'$, we write $\mathsf{pstep}_i(R, R')$ for the obvious function which gives the probability $p$ that $R \xrightarrow[i]{p} R'$ if one exists, and $0$ otherwise. A *trace* is either an initial configuration $R$ or a finite sequence of steps, $R_1 \xrightarrow[i_1]{p_1} R_2 \xrightarrow[i_2]{p_2} \dots \xrightarrow[i_n]{p_n} R_{n+1}$. Given a trace $H$, we write $H \xrightarrow[i]{p} R$ for the trace represented by appending an additional step by thread $i$ with probability $p$ to $R$. The last configuration in a trace is denoted by $\mathsf{curr}(H)$. A trace is safe if every configuration appearing in it is safe, and a trace is terminated if the last configuration in it is terminated. The *weight* of a trace is defined by:

$$\mathsf{weight}(H) = \begin{cases} 1 & \text{if } H = R \\ p_1 \cdots p_n & \text{if } H = R_1 \xrightarrow[i_1]{p_1} \dots \xrightarrow[i_n]{p_n} R_{n+1} \end{cases}$$

Notice that for fixed $R$ and $i$, if the $i$th thread of $R$ is not stuck then:

$$\sum_{R'} \mathsf{pstep}_i(R, R') = 1$$

so that having fixed a choice of which thread steps "next", the semantics defines a distribution on the configurations which $R$ steps to. It is the job of the *scheduler* to resolve the choice of which thread should step next at any given point. A scheduler $\varphi$ is a function from traces to thread indices, with the restriction that if $\mathsf{curr}(H) = [e_1, \dots, e_n]; \sigma$ then $1 \leq \varphi(H) \leq n$ and $e_{\varphi(H)}$ is not a value. This means that the scheduler can only select a valid thread index and cannot try to step a value. The scheduler may inspect not only the current state and each thread, but also the sequence of prior steps. Of course, in practice, schedulers do not really deeply inspect the current state of the program and the entire history of execution when selecting the next thread to run, but we are conservatively over-approximating the power of the scheduler[10].

We write $\mathsf{tstep}_\varphi(H, H')$ for the judgment expressing that under scheduler $\varphi$, the trace $H$ can step to $H'$:

$$\frac{\varphi(H) = i \qquad \mathsf{curr}(H) = R \qquad R \xrightarrow[i]{p} R'}{\mathsf{tstep}_\varphi(H, H \xrightarrow[i]{p} R')}$$

Let $\mathsf{tstep}^*_\varphi$ be the reflexive transitive closure of this relation.

[10]Some probabilistic algorithms assume the scheduler is weaker than this. For instance, some assume that the scheduler is entirely oblivious: it cannot inspect the configuration at all. We can incorporate this into the above setting by only proving theorems that quantify over a restricted set of schedulers, e.g. for oblivious schedulers we can consider only those $\varphi$ for which there exists some $\sigma$ such that $\varphi(H) = \sigma(\mathsf{length}(H))$.

Suppose under a given scheduler $\varphi$, no execution of $R$ diverges. That is, there is no infinite sequence of traces $H_1, H_2, ...$ such that $\mathsf{tstep}_\varphi(R, H_1)$ and $\mathsf{tstep}_\varphi(H_i, H_{i+1})$ for all $i$. Furthermore, assume every $H'$ such that $\mathsf{tstep}_\varphi(R, H')$ holds is safe. Then every execution terminates, and because the language only has bounded non-determinism, by König's lemma there are only finitely many terminated traces $H$ for which $\mathsf{tstep}^*_\varphi(R, H)$. Denote this set of traces by $\Omega$. Define $P : \mathcal{P}(\Omega) \to \mathbb{R}$ by:

$$P(A) = \sum_{H \in A} \mathsf{weight}(H)$$

Then $P(\Omega) = 1$ and $(\Omega, \mathcal{P}(\Omega), P)$ forms a discrete probability space. With respect to this probability space, $\mathsf{curr}$ is a discrete random variable giving the final configuration of a random execution of $R$.

# Bibliography

[1] Reynald Affeldt and Manabu Hagiwara. Formalization of Shannon's theorems in SSReflect-Coq. In *ITP*, pages 233–249, 2012. 5.2

[2] Maya Arbel and Hagit Attiya. Concurrent updates with RCU: search tree as an example. In *PODC*, pages 196–205, 2014. 2.1

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001. 1, 2.3

[4] James Aspnes and Eric Ruppert. Depth of a random binary search tree with concurrent insertions. In *DISC*, pages 371–384, 2016. 2.1, 2.2, 1, 4.2, 5.2

[5] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Sci. Comput. Program.*, 74(8):568–589, 2009. 5.2

[6] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *CoRR*, abs/1405.7012, 2014. URL `http://arxiv.org/abs/1405.7012`. 5.2

[7] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 593–602, 1994. 2.3

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, pages 1–6, 2012. 1, 3.4, 5, 4.1

[9] 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. 1, 3.4, 4.1, 3

[10] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving uniformity and independence by self-composition and coupling. In *LPAR*, 2017. 3.4

[11] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. In *POPL*, pages 161–174, 2017. 3.4

[12] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004. 3.1

[13] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comput. Sci.*, 411(47):4102–4122, 2010. 4.1

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 1, 2.3

[15] Stephen D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. *Electr. Notes Theor. Comput. Sci.*, 158:123–150, 2006. 1, 3.3

[16] Yuxin Deng and Wenjie Du. Logical, metric, and algorithmic characterisations of probabilistic bisimulation. *CoRR*, abs/1103.4577, 2011. URL `http://arxiv.org/abs/1103.4577`. 4.2

[17] Luc Devroye. A note on the height of binary search trees. *J. ACM*, 33(3):489–498, 1986. 2.1

[18] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *SPAA*, pages 43–52, 2013. 2.2, 1

[19] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. 3.1

[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010. 3.3

[21] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *POPL*, 2013. 3.3

[22] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010. 1, 2.1

[23] Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985. 2.2, 1

[24] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009. ISBN 978-0-521-89806-5. 2.2

[25] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998. 1, 2.3

[26] 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. 3.3

[27] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *ICFP*, pages 2–14, 2011. 4.2

[28] 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. 4.2

[29] 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. 4.2

[30] Jean Goubault-Larrecq and Achim Jung. QRB, QFS, and the probabilistic powerdomain. *Electr. Notes Theor. Comput. Sci.*, 308:167–182, 2014. 4.1

[31] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work

stealing deque. *Distributed Computing*, 18(3):189–207, 2006. 2.3

[32] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. *CoRR*, abs/1701.02547, 2017. URL `http://arxiv.org/abs/1701.02547`. 4.1

[33] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969. 3.1

[34] Johannes Hölzl. *Construction and stochastic applications of measure spaces in higher-order logic*. PhD thesis, Technical University Munich, 2013. 5.2

[35] Johannes Hölzl. Markov processes in isabelle/hol. In *CPP*, pages 100–111, 2017. 5.2

[36] Johannes Hölzl and Armin Heller. Three chapters of measure theory in isabelle/hol. In *ITP*, pages 135–151, 2011. 5.2

[37] Daniel Huang and Greg Morrisett. An application of computable distributions to the semantics of probabilistic programming languages. In *ESOP*, pages 337–363, 2016. 4.1

[38] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In *ESOP*, 2012. 3.3

[39] Arne T. Jonassen and Donald E. Knuth. A trivial algorithm whose analysis isn't. *J. Comput. Syst. Sci.*, 16(3):301–322, 1978. 2

[40] 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. 1, 3.3, 4.1

[41] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, 2016. (to appear). 3.3

[42] 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*, page To Appear, 2017. 4.1

[43] 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. 3.4, 4.1

[44] Richard M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):1136–1150, 1994. 5.2

[45] Gary D. Knott. *Deletion in Binary Storage Trees*. PhD thesis, Stanford University, May 1975. 2

[46] 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. 1, 3.4, 4.1

[47] 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. 3.3

[48] 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. 3.3

[49] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms*

*and probabilistic analysis*. Cambridge University Press, 2005. ISBN 978-0-521-83540-4. 5.2

[50] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996. 1, 3.4, 4.1

[51] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21 (10):840–842, 1978. 2.2

[52] Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *ASPLOS*, pages 413–426, 2014. 2.3

[53] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. ISBN 0-521-47465-5. 5.2

[54] 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. 3.3

[55] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014. 2.1

[56] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999. 3.2

[57] P.W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 375(1):271–307, 2007. 1, 3.3

[58] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005. 3.3

[59] Matthew J. Parkinson. The next 700 separation logics - (invited paper). In *VSTTE*, pages 169–182, 2010. 1, 3.3, 3

[60] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002. 4.2

[61] Lyle Harold Ramshaw. *Formalizing the Analysis of Algorithms.* PhD thesis, Stanford University, 1979. 3.4, 4.1

[62] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002. 1, 3.2

[63] Joseph Tassarotti. Probabilistic recurrence relations for work and span of parallel algorithms. *CoRR*, abs/1704.02061, 2017. URL `http://arxiv.org/abs/1704.02061`. 5.2

[64] Joseph Tassarotti and Robert Harper. Verified tail bounds for randomized programs, 2017. URL `https://www.cs.cmu.edu/~rwh/papers/tail-bounds/paper.pdf`. 5.2

[65] Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In *ESOP*, pages 909–936, 2017. 3.3, 4.1, 4.2, 5.1

[66] 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. 4.2

[67] 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. 3.3, 4.2

[68] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007. 3.3

[69] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113, 2006. 4.2, A

[70] Hongseok Yang. Relational separation logic. *TCS*, 375(1–3):308–334, 2007. 1