Tachis: Higher-Order Separation Logic with Credits for Expected Costs

PHILIPP G. HASELWARTER, Aarhus University, Denmark
KWING HEI LI, Aarhus University, Denmark
MARKUS DE MEDEIROS, New York University, USA
SIMON ODDERSHEDE GREGERSEN, New York University, USA
ALEJANDRO AGUIRRE, Aarhus University, Denmark
JOSEPH TASSAROTTI, New York University, USA
LARS BIRKEDAL, Aarhus University, Denmark

We present Tachis, a higher-order separation logic to reason about the expected cost of probabilistic programs. Inspired by the uses of time credits for reasoning about the running time of deterministic programs, we introduce a novel notion of probabilistic cost credit. Probabilistic cost credits are a separation logic resource that can be used to pay for the cost of operations in programs, and that can be distributed across all possible branches of sampling instructions according to their weight, thus enabling us to reason about expected cost. The representation of cost credits as separation logic resources gives Tachis a great deal of flexibility and expressivity. In particular, it permits reasoning about amortized expected cost by storing excess credits as potential into data structures to pay for future operations. Tachis further supports a range of cost models, including running time and entropy usage. We showcase the versatility of this approach by applying our techniques to prove upper bounds on the expected cost of a variety of probabilistic algorithms and data structures, including randomized quicksort, hash tables, and meldable heaps. All of our results have been mechanized using Coq, Iris, and the Coquelicot real analysis library.

1 Introduction

Randomization in programming has many applications, from improving efficiency to ensuring security and privacy. In many of these applications, it is crucial to reason about the complexity and resource usage of programs. For example, when analyzing a randomized algorithm, we want to prove that it meets certain performance requirements; and when analyzing security of a cryptographic

Authors' Contact Information: Philipp G. Haselwarter, Aarhus University, Denmark, pgh@cs.au.dk; Kwing Hei Li, Aarhus University, Denmark, hei.li@cs.au.dk; Markus de Medeiros, New York University, USA, mjd9606@nyu.edu; Simon Oddershede Gregersen, New York University, USA, s.gregersen@nyu.edu; Alejandro Aguirre, Aarhus University, Denmark, alejandro@cs.au.dk; Joseph Tassarotti, New York University, USA, jt4767@nyu.edu; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

protocol, standard definitions phrase security in terms of resilience against attacks that run in probabilistic polynomial time. Since the programs are probabilistic, their running time is typically also a random variable. One of the most common and important properties to analyze for this random variable is its expected value, known as the *expected running time* of the program. More generally, we may want to reason about the expected values of other kinds of resource usage, *e.g.*, consumed entropy or memory, which leads to a more general notion of *expected cost* of a randomized program. As in the deterministic setting, sequences of operations on a data structure may alternate between being cheaper and more costly, with the trade-off that a costly operation will allow for many cheaper operations to be performed afterwards. This leads to the notion of *amortized* cost, which also has its probabilistic variant, known as *amortized expected cost*.

This work presents Tachis, a higher-order separation logic for reasoning about (amortized) expected cost of probabilistic programs. Reasoning about resource usage in separation logic is natural as it is a *logic of resources*. As pointed out by Atkey [2011], running time can be represented as a separation logic resource through the notion of a *time credit* assertion which can be used to (logically) "pay" for steps of execution. In an affine logic, resources cannot be duplicated, so an initial budget of time credits will provide an upper bound on the number of execution steps. Prior work has shown that this approach scales to sophisticated algorithms and data structures [Charguéraud and Pottier 2019; Mével et al. 2019; Pottier et al. 2024].

In this work, we build a separation logic that uses credits to reason about *expected* cost. Reasoning about expected cost poses unique challenges in comparison to reasoning about cost of deterministic programs. For example, probabilistic programs with overall finite expected cost can still have execution traces with infinite cost. Consider, *e.g.*, the program below:

$$\text{rec coinToss \_ = tick 1; if flip then () else coinToss ()}$$

Here, flip is a command that corresponds to a fair coin toss; it evaluates to either true or false, each with probability 1/2. Meanwhile, the tick 1 command is used to represent incurring a cost of 1 unit of some resource. Then, the total expected cost of executing coinToss () (*i.e.* the expected number of tick 1 commands executed) is 2.

Imagine trying to prove this bound by starting with a precondition of 2 time credits. In prior separation logics for deterministic cost bounds, the proof rule for tick 1 consumes 1 time credit. Thus on every iteration of this function we consume 1 credit, flip a coin, and then either terminate if the result is true or do the recursive call if it is false, each with probability 1/2. This means that with probability 1/4 we sample false on two iterations in a row and consume 2 credits, leaving us unable to execute any further iterations. In fact for any initial number of credits k we will need more than k credits to run the program with probability $(1/2)^k$, so no initial amount of credits will suffice. In order to use credits to reason about the expected running time in this example, we need a way to soundly *increase* the number of credits when sampling false in the coin toss.

To support this we introduce a new *expectation-preserving* credit rule, inspired by prior work on the expected running time transformer [Kaminski et al. 2016]. When reasoning about sampling instructions, this rule allows us to distribute credits across all possible outcomes as long as the expected value over all branches is the initial quantity. In particular, credits can be scaled up for certain branches, as long as this is compensated for in other branches. For instance in the example above, after executing tick 1 we will have 1 credit, so after tossing the coin we can give 0 credits to the then branch and 2 credits to the else branch. In the former branch we terminate immediately, while in the latter we have the 2 credits needed to make the recursive call.

Reasoning about *amortized* expected costs using credits does not require any additional reasoning principles. Instead, amortized reasoning can be achieved by exploiting the expressiveness of separation logic and writing specifications for data structures that store a reserve of credits to pay

for more expensive operations, should they arise. This is in analogy to the *potential method* [Cormen et al. 2009; Tarjan 1985] often employed in pen-and-paper proofs of amortized complexity of deterministic data structures, and it also generalizes to the probabilistic setting using a similar pattern as appears in prior works based on deterministic time credits [Charguéraud and Pottier 2019; Mével et al. 2019].

Because the analysis of randomized algorithms and data structures often involves different notions of cost, Tachis is parameterized by a *cost model* that assigns a cost to every command in the language. Some previous logics for cost analysis have provided indirect support for encoding cost models through primitives like the tick n command above, which can be inserted to instrument a program to track some notion of cost. However, this instrumentation approach works less well for certain natural cost measures for randomized programs, such as *entropy consumed*, where the cost of drawing a random integer uniformly from $\{0, \ldots, n\}$ is $\log_2(n + 1)$. In contrast, Tachis' parameterized approach allows us to express this kind of complex cost function in a natural way.

We emphasize that Tachis is a fully general program logic that retains all the expressiveness of standard higher-order separation logic. Furthermore, for deterministic cost-free programs we can reuse existing specifications and proofs without any changes. By mixing standard separation logic reasoning with reasoning about credits, we achieve greater expressiveness than single-purpose approaches to reasoning about expected cost.

This expressiveness is demonstrated through several case studies, including, a hashmap, a version of Quicksort parameterized by a comparator function, and an algorithm for merging multiple sorted lists, which makes use of a randomized min-heap. These examples show Tachis' support for higher-order specifications, complex data structure invariants, and compositional reasoning. Several of these case studies are beyond the scope of prior work for expected cost analysis because they combine higher-order functions with mutable state. Tachis is the first program logic for reasoning about expected costs that supports this combination of features.

Contributions. To summarize, we provide:
- The first compositional higher-order separation logic, Tachis, for reasoning about expected cost of probabilistic programs written in a probabilistic higher-order programming language with higher-order local references.
- A probabilistic counterpart of cost credits, which enable us to achieve value-dependent and expectation-preserving proof rules for compositional reasoning.
- User-defined cost models, which enable reasoning about a richer class of costs, such as expected entropy use.
- A substantial collection of case studies that demonstrates how one can modularly derive non-trivial (amortized) cost bounds of probabilistic programs.
- All of the results in this paper have been mechanized in the Coq proof assistant, using the Iris separation logic framework [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017] and the Coquelicot real analysis library [Boldo et al. 2015].

Outline. In §2 we showcase the main features of Tachis by proving expected cost bounds for the program coinToss presented earlier and another illustrative example which requires amortized reasoning. Later in §3, we recall some preliminaries and present the language RandML as well as our notion of cost model. We then introduce the Tachis logic in §4. We demonstrate how to use Tachis to establish non-trivial expected cost bounds on a range of examples in §5. Subsequently in §6, we present the model of Tachis and provide a sketch of the adequacy theorem. Finally, we discuss related and future work in §7 and conclude in §8.

2 Key Ideas

In this section, we demonstrate the features of Tachis by verifying two illustrative examples. For the first, we show how our expectation-preserving proof rule works. For the second, we show how representing cost credits as a resource enables us to reason about *amortized* expected cost for free, without needing to add any other new reasoning principles to our logic.

2.1 Repeated Coin Toss

We prove an expected cost bound for the following variant of the coin-tossing program we provided in the introduction

$$\text{rec coinToss } \_ = \text{if flip then () else coinToss ()}$$

where instead of an explicit tick operation, we use a cost model that counts every function call, *i.e.*, every function application costs 1 unit while all other reductions have cost 0 (it is known that just counting the number of function applications gives the right overall asymptotic complexity for programs written in a language like the one considered here [Charguéraud and Pottier 2019; Le Métayer 1988]). Despite this change, the expected cost is the same, as the cost-inducing operation is just moved from the tick to the function application of coinToss. Hence, it is not surprising that the cost of the program follows a geometric distribution and that the expected cost is $2 = \sum_{k=0}^{\infty} 2^{-k}$. We thus use Tachis to prove the following bound:

$$\text{EC (coinToss (), } \emptyset) \le 2$$

where EC (e, σ) denotes the expected cost of running the program e with heap σ, and $\emptyset$ denotes the empty initial heap. In §4 we will discuss the *adequacy theorem* for Tachis, which states that if we prove a Hoare triple for e in Tachis and we "own x credits in the beginning", then the expected cost of running e is less than x. Hence it suffices to prove the following specification:

$$\{\$(2)\} \text{ coinToss () } \{\text{True}\} \tag{1}$$

where the assertion $\$(x)$ in the precondition expresses that we start our proof owning x credits.

We first note that coinToss is recursive, and so we apply a proof rule for recursive function applications. Under the current cost model function applications cost 1 unit, so this rule has the following form:

$$\frac{\forall w. \{P * \$(1)\} \text{ (rec } f \ x = e) \ w \ \{Q\} \vdash \{P\} \ e[v/x] \, [(\text{rec } f \ x = e)/f] \ \{Q\}}{\vdash \{P * \$(1)\} \text{ (rec } f \ x = e) \ v \ \{Q\}} \quad \text{HT-REC}$$

This rule is relatively standard, except for the additional $\$(1)$ credits in the precondition, which we need in order to "pay" for the function application. In particular, it says that we may assume the desired specification for any recursive calls (the assumption to the left of the $\vdash$) while reasoning about the body of the function.

Since we are considering a cost model where all reductions other than function application incur no costs, we have "standard" proof rules for non-function application steps, *e.g.*, the following rule.

$$\frac{\vdash \{P * b = \text{true}\} \ e_1 \ \{Q\} \qquad \vdash \{P * b = \text{false}\} \ e_2 \ \{Q\}}{\vdash \{P\} \text{ if } b \text{ then } e_1 \text{ else } e_2 \ \{Q\}} \quad \text{HT-IF}$$

Coming back to our example, note that instead of exactly $\$(1)$ credits for the precondition of the HT-REC rule, we have $\$(2)$. Hence we need to split our credits up to pay for this function application step via the following rule for credits $\$(x_1) * \$(x_2) \dashv\vdash \$(x_1 + x_2)$. Thus we split $\$(2)$ into $\$(1) * \$(1)$, where the first $\$(1)$ is used to pay for the function application and the second

is passed on to reason about the remainder of the program (the body of the function call). After splitting the credits and applying HT-REC to Equation (1), we are left to prove

$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(1)\} \text{ if flip then } () \text{ else coinToss } () \{\text{True}\}$$

Next, we reason about the coin toss (the flip). We need to consider the expected cost for the cases where the toss returns false and where it returns true. We observe that the two cases have different expected cost; therefore, we need to split our credits in an "expectation-preserving" manner across the two branches, as alluded to in the introduction. To do so, we use the *expectation rule* for flip:[1]

$$\frac{\frac{1}{2} \cdot X_2(\text{true}) + \frac{1}{2} \cdot X_2(\text{false}) \leq x_1}{\{\$(x_1)\} \text{ flip } \{b. \$(X_2(b))\}} \text{ HT-FLIP-EXP}$$

This rule states that for $X_2 : \mathbb{B} \to \mathbb{R}_{\geq 0}$ we may split $\$(x_1)$ into $\$(X_2(\text{false}))$ and $\$(X_2(\text{true}))$ for the two branches of flip, if their weighted sum is less than or equal to x_1. This encodes the intuition that we distribute credits in a expectation-preserving way, where the expected value over all branches is bounded above by the initial quantity. Applying this rule in our example, from the $\$(1)$ in the precondition we can pass $\$(0)$ to the then branch and $\$(2)$ to the else branch. That is, we pick $X_2(b) \triangleq \text{if } b \text{ then } 0 \text{ else } 2$ which is a valid choice since $\frac{0}{2} + \frac{2}{2} = 1 \leq 1$. By HT-FLIP-EXP and an application of the standard HT-BIND rule, it then suffices to show:

$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(X_2(b))\} \text{ if } b \text{ then } () \text{ else coinToss } () \{\text{True}\}$$

From here, we can apply the HT-IF rule, which performs a case split on b to verify the rest of the two branches depending on the return value of flip, *i.e.*,

$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(X_2(b)) * b = \text{true}\} () \{\text{True}\}$$
$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(X_2(b)) * b = \text{false}\} \text{ coinToss } () \{\text{True}\}$$

By symbolic simplification of the preconditions above, it thus suffices to show

$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(0)\} () \{\text{True}\}$$
$$\{\$(1) * \$(1)\} \text{ coinToss } () \{\text{True}\} \vdash \{\$(2)\} \text{ coinToss } () \{\text{True}\}$$

The first program immediately returns (), which trivially satisfies the postconsition True, without incurring any additional cost, finishing the case. For the second, we apply the assumption after splitting the $\$(2)$ credits as $\$(1) * \$(1)$.

If one chooses to count only the number of calls to flip, one can prove the same specification in a similar manner. Indeed, the proof rules and soundness of Tachis are agnostic to the definition of the cost model, and users can define their own cost model, as we shall see in §3.3.

To summarize, we demonstrated how one uses Tachis to verify the expected cost of a simple coin-tossing program. In this one example, we highlighted how one can

(1) Use credits, a separation logic resource that enjoys all the usual separation logic rules (such as the frame rule), to keep track of the cost incurred as we step through the program.
(2) Use expectation-preserving composition to reason about the expected cost across multiple branches after sampling operations.

---

[1]We provide a generalized expectation rule for more general uniform sampling in §4.2.

2.2 Repeated Amortized Operations

Next, we consider an example of amortized expected time reasoning introduced by Batz et al. [2023] to motivate their amortized ERT calculus. We will see that just by combining credits with standard separation logic reasoning rules, we can handle the amortized reasoning in this example.

Let $e_1 \oplus e_2$ be notation for if flip then e_1 else e_2 and consider the program

$$\text{op} \triangleq (\text{tick}(!\ell) \oplus \ell \leftarrow 0); \ell \leftarrow (!\ell) + 1$$

Here the operator $\text{tick}(!\ell)$ indicates a cost of $!\ell$, where ℓ is some memory location and $!$ is the dereferencing operator. If location ℓ contains a natural number n then with probability $\frac{1}{2}$, op either has cost n or resets ℓ to 0. In both cases, ℓ is subsequently incremented. As pointed out by Batz et al., it is easy to show that if ℓ contains n, the expected cost of the operation is $\frac{n}{2}$, i.e.,

$$\left\{\$\left(\tfrac{n}{2}\right) \; * \; \ell \mapsto n\right\} \text{op} \left\{\exists m. \ell \mapsto m\right\}$$

where $\ell \mapsto n$ is the standard *points-to* connective from separation logic stating that ℓ contains the value n. However, computing the cost of a *sequence* of operations in general is much less obvious. For example, one can show that the expected cost of (op; op) is $\frac{3n}{4} + \frac{1}{4}$ and for (op; op; op) it is $\frac{7n}{8} + \frac{5}{8}$ which does not lead to an obvious pattern. However, repeating the operation m times has *amortized* cost $n + m$, which Batz et al. establish using a dedicated amortized expected running time calculus. In Tachis, such amortized expected reasoning emerges from using credits with standard separation logic reasoning, just as prior work has shown that deterministic credits enable deterministic amortized bounds [Atkey 2011; Charguéraud and Pottier 2019].

To show the amortized specification, we first establish that op has amortized cost 1, i.e., we show

$$\{\$(1) \; * \; \pi\} \text{op} \{\_. \pi\}$$

where $\pi = \exists n. \ell \mapsto n * \$(n)$. Intuitively, π contains the *potential* needed to evaluate the operation which changes according to the contents of ℓ. The precondition of the specification thus gives us a total of $n + 1$ credits, where n is the content of ℓ. The key part of the proof is applying the HT-FLIP-EXP rule using $X_2(b) \triangleq$ if b then $2n + 1$ else 1. If flip evaluates to false, we reset ℓ to 0 and thus no longer need the n credits stored in π for that branch. If flip evaluates to true, we have to both spend n credits and also maintain π with n credits, so we need $2n$ credits in total. As $\frac{2n+1}{2} + \frac{1}{2} = n + 1$, the expectation rule allows us, intuitively, to bring the n credits from the else branch to the then branch, as the credits needed in the two branches average out. In both cases, we will have the $\$(1)$ supplied by the precondition left over, which we need to store back as potential, since the content of ℓ is incremented and this re-establishes π as a postcondition.

It is now straightforward to show an amortized specification of m repetitions of op. Let repeat $m\, f$ be a program that invokes the f procedure m times. By applying a general higher-order specification of repeat and the specification of op we show

$$\{\$(n + m) \; * \; \ell \mapsto n\} \text{ repeat } m\ (\lambda\_. \text{op}) \{\text{True}\}$$

as a simple corollary and thus an amortized expected cost of $n + m$.

To summarize, this example illustrates how the resource-based representation plus the expressivity of the ambient higher-order separation logic enables us to reason about amortized expected cost without the need for any other specialized rules. In general, by "depositing" excess credits in a data structure's potential assertion (like the π above), and then "withdrawing" those stored credits to pay down expensive operations, we can capture standard patterns of amortized reasoning. The HT-FLIP-EXP rule further allows us to average stored credits across branches to do amortized expected reasoning.

3 Preliminaries

In this section, we first recall some useful definitions from probability theory. We then present the syntax of RandML, the high-order probabilistic ML-like language our programs are written in, as well as its operational semantics. Lastly, we present the notion of cost models, the device that allows us to define a more general class of costs depending on the application, and we define the expected cost of a program according to a cost model.

3.1 Probabilities

A *discrete subdistribution* (henceforth simply *distribution*) on a countable set A is a function $\mu : A \to [0, 1]$ such that $\sum_{a \in A} \mu(a) \leq 1$. The collection of distributions on A is denoted by $\mathcal{D}(A)$.

A (non-negative, real) *random variable* with respect to a distribution $\mu : \mathcal{D}(A)$ is a function $X : A \to \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ denotes the non-negative real numbers. The *expectation* or *expected value* of a random variable X w.r.t. μ is defined as $\mathbb{E}_\mu[X] \triangleq \sum_{a \in A} \mu(a) \cdot X(a)$ if the sum converges, in which case we say that X has finite expectation.

The collection of distributions can be equipped with a monad structure as usual:

LEMMA 1 (DISCRETE DISTRIBUTION MONAD). *Let $\mu \in \mathcal{D}(A)$, $a \in A$, and $f : A \to \mathcal{D}(B)$. Then*

$$\text{bind}(f, \mu)(b) \triangleq \sum_{a \in A} \mu(a) \cdot f(a)(b) \qquad \text{ret}(a)(a') \triangleq \begin{cases} 1 & \text{if } a = a' \\ 0 & \text{otherwise} \end{cases}$$

gives a monadic structure to $\mathcal{D}$. We write $\mu \ggg f$ for $\text{bind}(f, \mu)$.

3.2 Language Definition and Operational Semantics

The syntax of RandML, the language we consider in this paper, is defined by the grammar below.

$$v, w \in Val ::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid () \mid \ell \in Loc \mid \text{rec } f\, x = e \mid (v, w) \mid \text{inl } v \mid \text{inr } v$$
$$e \in Expr ::= v \mid x \mid \text{rec } f\, x = e \mid e_1\, e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid$$
$$(e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{inl } e \mid \text{inr } e \mid \text{match } e \text{ with inl } v \Rightarrow e_1 \mid \text{inr } w \Rightarrow e_2 \text{ end} \mid$$
$$\text{allocn } e_1\, e_2 \mid !\, e \mid e_1 \leftarrow e_2 \mid e_1[e_2] \mid \text{rand } e \mid \text{tick } e$$
$$K \in Ectx ::= - \mid e\, K \mid K\, v \mid \text{allocn } e\, K \mid \text{allocn } K\, v \mid !\, K \mid e \leftarrow K \mid K \leftarrow v \mid \text{rand } K \mid \text{tick } K \mid \ldots$$
$$\sigma \in State \triangleq Loc \xrightarrow{\text{fin}} Val \qquad \rho \in Cfg \triangleq Expr \times State$$

The term language is mostly standard: allocn $e_1\, e_2$ allocates a new array of length e_1 with each cell containing the value returned by e_2, $!\, e$ dereferences the location e evaluates to, and $e_1 \leftarrow e_2$ assigns the result of evaluating e_2 to the location that e_1 evaluates to. We also write $\ell[i]$ for offsetting location ℓ by i. Reading from and writing v to an array a at offset i is thus written as $!\, a[i]$ and $a[i] \leftarrow v$ respectively. We often refer to a recursive function value rec $f\, x = e$ by its name f. We introduce syntactic sugar for lambda abstractions $\lambda x.\, e$ defined as rec $\_\, x = e$, let-bindings let $x = e_1$ in e_2 defined as $(\lambda x.\, e_2)\, e_1$, sequencing $e_1; e_2$ defined as let $\_ = e_1$ in e_2, and references ref e defined as allocn 1 e. The Boolean operation flip is syntactic sugar for (rand 1 == 1). The tick instruction has no operational meaning (tick n simply reduces to ()) but can be used by cost models to explicitly specify the cost of a computation in the program, see, *e.g.*, $cost_{\text{tick}}$ in §3.3.

To define full program execution, we define $\text{step}(\rho) \in \mathcal{D}(Cfg)$, the distribution induced by the single step reduction of configuration $\rho \in Cfg$. The semantics is standard, we first define head reductions and then lift it to reduction in an evaluation context K. All non-probabilistic constructs reduce deterministically as usual, *e.g.*, $\text{step}(\text{if true then } e_1 \text{ else } e_2, \sigma) = \text{ret}(e_1, \sigma)$ and

step(tick z, σ) = ret((), σ). The probabilistic choice rand N reduces uniformly at random, *i.e.*,

$$\text{step}(\text{rand } N, \sigma)(n, \sigma) \triangleq \begin{cases} \frac{1}{N+1} & \text{for } n \in \{0, 1, \dots, N\}, \\ 0 & \text{otherwise.} \end{cases}$$

With the single step reduction step$(-, -)$ defined, we now define a stratified execution probability $\text{exec}_n : Cfg \rightarrow \mathcal{D}(Val)$ by induction on n:

$$\text{exec}_n(e, \sigma) \triangleq \begin{cases} \lambda v. \, 0 & \text{if } e \notin Val \text{ and } n = 0, \\ \text{ret}(e) & \text{if } e \in Val, \\ \text{step}(e, \sigma) \ggg \text{exec}_{(n-1)} & \text{otherwise.} \end{cases}$$

The probability that a full execution, starting from configuration ρ, reaches a value v is taken as the limit of its stratified approximations, which exists by monotonicity and boundedness:

$$\text{exec}(\rho)(v) \triangleq \lim_{n \to \infty} \text{exec}_n(\rho)(v)$$

We simply write $\text{exec}(e)$ as notation for $\text{exec}(e, \sigma)$ if $\text{exec}(e, \sigma)$ is the same for all states σ. The termination probability of an execution from a configuration ρ is denoted by $\text{exec}_{\Downarrow}(\rho) \triangleq \Sigma_{v \in Val} \text{exec}(\rho)(v)$.

3.3 Cost Models and Expected Cost

A *cost model* is a non-negative real-valued function $cost : Expr \rightarrow \mathbb{R}_{\geq 0}$ that is invariant under evaluation contexts in the sense that if $e \notin Val$ then $cost(K[e]) = cost(e)$ for all K and e. We use cost models to associate costs (non-negative real values) to one step of evaluating an expression, *i.e.*, $cost(e)$ represents the cost incurred by the next transition of e. At its simplest, a cost model can associate a constant cost to all expressions, and hence to all reduction steps. For instance, the cost model $(\lambda\_. \, 1)$ is obviously invariant under evaluation contexts.

For a given cost model $cost$, the *expected cost* of executing e for n steps from starting state σ is defined as follows.

$$\text{EC}_n^{cost}(e, \sigma) \triangleq \begin{cases} 0 & \text{if } n = 0 \text{ or } e \in Val, \\ cost(e) + \mathbb{E}_{\text{step}(e,\sigma)}\left[\text{EC}_m^{cost}\right] & \text{if } n = m + 1. \end{cases} \tag{2}$$

In this definition, we consider EC_m^{cost} as a random variable w.r.t. the distribution induced by stepping the current configuration. Concretely, $\mathbb{E}_{\text{step}(e,\sigma)}\left[\text{EC}_m^{cost}\right] = \sum_{\rho \in Cfg} \text{step}(e, \sigma)(\rho) \cdot \text{EC}_m^{cost}(\rho)$. Note that our language semantics ensures that $\text{step}(e, \sigma)(\rho)$ is non-zero at finitely many configurations, so this expected value is always finite.

The *expected cost* for full program execution is defined as $\text{EC}^{cost} \triangleq \sup_{n \in \omega} \text{EC}_n^{cost}$, where we take $\text{EC}^{cost} \triangleq \infty$ if the sequence is not bounded.

Examples of cost models. Since cost models have to respect evaluation contexts, they are usually defined in term of the current head redex of an expression. Given a (reducible) expression e, there exists a unique maximal evaluation context K and expression e' such that we can decompose e as $K[e']$ with e' reducible using a head reduction step. Let decomp : $Expr \rightarrow Expr$ denote the function that computes e' from e. One can show that if a cost model $cost$ is defined in terms of decomp, then $cost$ automatically satisfies the requirement of context-invariance.

For example, we can define the following cost functions.

$$cost_{all} \triangleq \lambda\_. \ 1$$

$$cost_{app} \triangleq \lambda e. \ 1 \qquad \text{if } decomp(e) = e_1 \ e_2 \qquad \text{for some } e_1, e_2, \text{ and} \qquad 0 \quad \text{otherwise.}$$

$$cost_{rand} \triangleq \lambda e. \ \log_2(N+1) \quad \text{if } decomp(e) = \text{rand } N \quad \text{for some } N, \text{ and} \qquad 0 \quad \text{otherwise.}$$

$$cost_{tick} \triangleq \lambda e. \ |z| \qquad \text{if } decomp(e) = \text{tick } z \quad \text{for some } z \in \mathbb{Z}, \text{ and} \quad 0 \quad \text{otherwise.}$$

As previously mentioned, $cost_{all}$ associates a constant cost to all reduction steps. The cost model $cost_{app}$ was used in the example in §2, and counts the number of function applications. This is known to behave asymptotically as the total number of reduction steps, but it has the advantage that it allows to abstract away some concrete implementation details. $cost_{rand}$ counts the entropy cost incurred by sampling commands. Recall that sampling uniformly out of a set of N elements has an entropy cost of $\log_2 N$. Generating high-quality random bits can be expensive, particularly if they must be cryptographically secure, so entropy usage of a computation is an important cost consideration [Lumbroso 2013]. Finally, $cost_{tick}$ counts the cost of all tick instructions. This is in line with previous works, and allows for cost analysis through code annotations.

4 The Expected Cost Logic

The Tachis logic for expected cost is built on top of the Iris base logic [Jung et al. 2018], a state-of-the-art higher-order separation logic. On top of the base logic, we add a predicate that denotes ownership of credits and a notion of Hoare triple that can be used to write specifications of programs. Hoare triples can then be established using a set of program logic rules and mapped to a concrete bound on expected cost using an adequacy theorem.

In this section we discuss each of these aspects in more detail but save the proof of soundness for §6. Below, we parameterize the development in terms of an arbitrary cost function called *cost* that satisfies the conditions discussed in §3.3.

4.1 Base Logic of Propositions

The Iris base logic is an expressive higher-order separation logic framework with support for advanced features such as higher-order ghost state [Jung et al. 2016], impredicative invariants [Svendsen and Birkedal 2014], and guarded recursion, all of which are paramount for reasoning about higher-order programs written in realistic languages. While important, we only discuss these more specialized connectives as necessary to focus on the novelties of this work. A selection of Tachis propositions are shown below.

$$P, Q \in iProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x. \ P \mid \exists x. \ P \mid$$

$$P * Q \mid P \twoheadrightarrow Q \mid \ell \mapsto v \mid \$(x) \mid \{P\} \ e \ \{v. Q\} \mid \dots$$

With the exceptions of the credit assertion $\$(x)$ and Hoare triples $\{P\} \ e \ \{v.Q\}$, all propositions have the usual meaning as in (non-probabilistic) separation logic. The $\$(x)$ assertion denotes ownership of $x \in \mathbb{R}_{\geq 0}$ credits. This is a separation logic resource subject to a set of rules, the most important of which is the credit splitting rule $\$(x_1 + x_2) \dashv\vdash \$(x_1) * \$(x_2)$. That is, owning $x_1 + x_2$ credits is equivalent to owning both x_1 and x_2 credits. Splitting credits allows us, *e.g.*, to pass a subprogram the exact amount of credits it needs while keeping the remaining credits in reserve. The logic is affine, which means that we may "throw away" resources and credits in particular, but they cannot be duplicated or generated from nothing. This ensures that the initial amount of credits assumed by a program provides a sound upper bound on its cost.

A Hoare triple $\{P\} \ e \ \{v. Q\}$ is a proposition that states a specification for a program e. Here P denotes the precondition and Q a postcondition with a free variable v, which morally captures

the return value of e. The Tachis logic is *impredicative*, *i.e.*, both Hoare triples and their pre- and postconditions are first-class propositions of the logic, and in particular they may contain Hoare triples as well. This is important for giving general and compositional specifications to higher-order programs as showcased in §5. In order to discuss the examples in §5, and for any user of Tachis, it is only important to understand what Hoare triples imply (through the adequacy theorem) and how they are proven (using the program logic rules) rather than how they are defined in §6.

4.2 Program Logic

Soundness of the program logic is captured by the following adequacy theorem.

THEOREM 2 (ADEQUACY). *Let x be a non-negative real number and let φ be a predicate on values. If $\vdash \{\$(x)\} e \{\varphi\}$ then for any state σ,*

(1) EC $(e, \sigma) \leq x$, and
(2) $\forall v \in Val.\ exec(e, \sigma)(v) > 0 \implies \varphi(v)$.

The theorem says that by proving a Hoare triple specification of e that assumes initial ownership of x credits, then the expected cost of executing e is bounded above by x. Moreover, any value in the output distribution satisfies φ. Recall that we assume an ambient *cost* function which the concrete definition of EC $(\cdot, \cdot)$ depends on.

Tachis Hoare triples satisfy a range of structural and computational rules. Notably, both the structural rules and the computational rules look almost identical to the corresponding rules found in separation logics for reasoning about non-probabilistic programs, with the addition of cost tracking through credits. We emphasize that this is a strength of our approach and proofs of non-probabilistic programs, *e.g.*, data structures, can readily be reused if the cost model permits.

A selection of structural and computational rules are shown below.

$$\frac{\vdash \{P\}\, e\, \{v.Q\} \qquad \vdash \forall v.\, \{Q\}\, K[v]\, \{R\}}{\vdash \{P\}\, K[e]\, \{R\}}\ \text{HT-BIND} \qquad\qquad \frac{\vdash \{P\}\, e\, \{Q\}}{\vdash \{P * R\}\, e\, \{Q * R\}}\ \text{HT-FRAME}$$

$$\frac{}{\vdash \{\ell \mapsto v * \$(cost(!\,\ell))\}\ !\,\ell\ \{w.\, w = v * \ell \mapsto v\}}\ \text{HT-LOAD}$$

$$\frac{}{\vdash \{\$(cost(\mathsf{rand}\,N))\}\ \mathsf{rand}\,N\ \{n.\, 0 \leq n \leq N\}}\ \text{HT-RAND}$$

$$\frac{e_f = (\mathsf{rec}\,f\,x = e) \qquad \forall w.\, \{P * \$(cost(e_f\,w))\}\, e_f\,w\,\{Q\} \vdash \{P\}\, e[v/x][e_f/f]\, \{Q\}}{\vdash \{P * \$(cost(e_f\,v))\}\, e_f\,v\,\{Q\}}\ \text{HT-REC}$$

Notice in particular that HT-BIND is the "usual" bind rule. This fact relies on the requirement that cost models are invariant under evaluation contexts, *i.e.*, the cost of evaluating e in context K is the same as the cost of evaluating e in isolation. The frame rule HT-FRAME is the traditional separation-logic frame rule. Since credits are assertions, they can be framed using this rule just like any other separation logic resource.

The computational rule HT-LOAD is used to reason about read operations. It requires ownership of the location ℓ as denoted by the points-to connective $\ell \mapsto v$. In addition, we are required to own $cost(!\,\ell)$ credits to pay for the (possible) cost of the read operation. Similar computational rules hold for store and allocation operations. The rule HT-REC can be used to reason about recursive functions. The rule requires ownership of enough credits to pay for the function invocation and provides a Hoare triple specification for reasoning about recursive calls of e_f. While the rule allows

the cost to depend on the actual function and its arguments, we only consider cost models that assign a uniform cost to all function applications.

The sampling operation rand N satisfies a computational rule HT-RAND similar to HT-LOAD where $cost(\text{rand } N)$ credits are required in the precondition. However, a key novelty of this work is that it also satisfies the rule HT-RAND-EXP shown below which allows for *expectation-preserving composition*. The rule distributes credits among all possible outcomes of a sampling in an expectation-preserving manner. Specifically, suppose we are given x_1 credits in the precondition of a rand N operation which samples uniformly from the set $\{0, \ldots, N\}$. Given that n is the outcome of the sampling, the rule says that we can supply $X_2(n)$ credits for the remainder of the program, as long as the cost of rand N and the *weighted* sum of $X_2 : \{0, \ldots, N\} \to \mathbb{R}_{\geq 0}$ is at most x_1.

$$\frac{cost(\text{rand } N) + \sum_{n=0}^{N} \frac{X_2(n)}{N+1} \leq x_1}{\{\$(x_1)\} \text{ rand } N \{n.\ \$(X_2(n)) \ * \ 0 \leq n \leq N\}} \text{ HT-RAND-EXP}$$

4.3 Tactic Support

One might worry that the $\$(cost(e))$ preconditions in all of the rules add significant overhead to carrying out proofs in Tachis, even in a cost model where most of these costs are 0. To address this, the Coq formalization of Tachis extends the Iris Proof Mode [Krebbers et al. 2018] which provides tactic support for interactively reasoning about judgments in the Iris separation logic. For example, when verifying a program and applying HT-LOAD, the tactics automatically decompose the expression in the goal into its evaluation context and head redex, apply HT-BIND, and locate the corresponding points-to connective in the resource context before applying HT-LOAD. We adapt these tactics to automatically determine the cost of the redex and locate sufficient credits in the resource context. For the cost models we have considered, managing credits is no more of a nuisance than managing points-to connectives, even for models like $cost_{\text{all}}$ where credits must be provided for all execution steps.

5 Case Studies

In this section, we present a collection of case studies where we apply Tachis to reason about expected cost. The examples come from a spectrum of applications, each presenting various challenges, *e.g.*, amortized reasoning, local state, and compositionality. We demonstrate the usability of Tachis by proving concise, non-trivial, and modular specifications for each of these examples.

5.1 Coupon Collector

The coupon collector's problem is a famous puzzle in probability theory. The problem goes as follows: given $n > 0$ distinct coupons in a bag, what is the expected number of random draws with replacement you need to do to draw every coupon at least once? A standard pen-and-paper analysis starts by observing that the total number of draws D can be rewritten as the sum $D = \sum_{i=1}^{n} D_i$ where D_i is the random variable representing the number of draws to draw the ith coupon after having drawn $i - 1$ distinct coupons previously. By linearity of expectation, we have $\mathbb{E}[D] = \mathbb{E}\left[\sum_{i=1}^{n} D_i\right] = \sum_{i=1}^{n} \mathbb{E}[D_i]$. Then, each D_i follows a geometric distribution with expectation $\frac{n}{n-(i-1)}$ and hence $\mathbb{E}[D] = \sum_{i=1}^{n} \frac{n}{n-(i-1)} = n \cdot (\sum_{i=1}^{n} \frac{1}{i}) = n \cdot H(n)$ where $H(x)$ represents the harmonic sum $\sum_{i=1}^{x} \frac{1}{i}$. As we shall see, the frame rule allows us to carry out a proof in Tachis that is quite similar in structure to this standard analysis.

We implement the coupon collection process as the RandML program shown in Figure 1. The implementation initially allocates an array arr of length n where each cell is initialized to false; the array tracks the coupons that have been collected so far. The program repeatedly samples a coupon

```
rec repeatDraw arr cnt =                          collector ≜ λ_.
    if cnt = 0 then ()                                let arr = allocn n false in
    else let k = rand (n − 1) in                      repeatDraw arr n
        if ! arr[k] then repeatDraw arr cnt
        else arr[k] ← true;
            repeatDraw arr (cnt − 1)
```

Fig. 1. An implementation of the coupon collector.

k by executing rand $(n − 1)$. If arr[k] is true, the coupon has already been drawn, so the process repeats. Otherwise, we mark arr and repeat the process with cnt decremented by 1. The process stops when cnt reaches 0 which signifies that n distinct coupons have been drawn.

Following the analysis shown above, we expect to prove the following specification using a cost model that assigns cost 1 to rand N and 0 to everything else.

$$\{\$(n \cdot H(n))\} \text{ collector } () \text{ \{True\}}$$

We start by proving a specification for the repeatDraw loop.

$$\left\{ \begin{array}{l} 0 < \text{cnt} \le n * \text{length } l = n * \text{size } s = n - \text{cnt } * \\ \text{lsmatch } l \, s * \$(n \cdot H(\text{cnt})) * \text{arr} \mapsto^* l \end{array} \right\} \text{ repeatDraw arr cnt \{True\}}$$

Here arr $\mapsto^* l$ denotes ownership of an array starting at location arr and the list of its contents l. The predicate lsmatch asserts that that ith index of the list l is true if and only if i is in the set s. In essence, the specification states that if there are cnt distinct coupons which we have not drawn before, we need to provide exactly $n \cdot H(\text{cnt})$ credits to execute repeatDraw arr cnt.

The key part of the proof is the sampling step, where we need to distribute the $n \cdot H(\text{cnt})$ credits depending on the result of the rand operation, *i.e.*, we need to figure out what postcondition $\Phi_?$ to use when deriving the Hoare triple $\{\$(n \cdot H(\text{cnt}))\}$ rand n $\{\Phi_?\}$ with HT-RAND-EXP.

Notice that $n \cdot H(\text{cnt}) = \sum_{i=1}^{\text{cnt}} D_i$. Looking back at our pen-and-paper proof, by linearity of expectations, the cost to get the ith coupon is only accounted for by D_i, and all D_j with $i \ne j$ are independent. Using the frame rule, we can temporarily "set aside" the credits needed for D_j:

$$\frac{\left\{\$\left(\frac{n}{\text{cnt}}\right)\right\} \text{ rand } n \left\{\Phi'_?\right\}}{\left\{\$\left(\frac{n}{\text{cnt}}\right) * \$(n \cdot H(\text{cnt} − 1))\right\} \text{ rand } n \left\{\Phi'_? * \$(n \cdot H(\text{cnt} − 1))\right\}}$$

We proceed by the expectation-preserving composition rule and assign $\frac{n}{\text{cnt}}$ credit to those branches where the result of rand n is already in s and 0 otherwise, *i.e.*,

$$\left\{\$\left(\tfrac{n}{\text{cnt}}\right)\right\} \text{ rand } n \left\{x. \left(\$\left(\tfrac{n}{cnt}\right) \wedge x \in s\right) \vee \left(\$(0) \wedge x \notin s\right)\right\}$$

as $cost(\text{rand } n) + \frac{\text{size } s}{n} \cdot \frac{n}{\text{cnt}} + \frac{n - \text{size } s}{n} \cdot 0 = 1 + \frac{(n − \text{cnt})}{n} \cdot \frac{n}{\text{cnt}} \le \frac{n}{\text{cnt}}$ as required. The remainder of the proof follows by symbolic execution.

While the coupon collector has been verified using previous program logics for expected run time analysis, such as the ERT calculus of Kaminski et al. [2016], the structure of the proof in that calculus is quite different from the pen-and-paper analysis. In contrast, Tachis's support for framing allows for reasoning about the expectation of each D_i, much as in the pen-and-paper proof. Existing automated resource analysis tools, such as eco-imp [Avanzini et al. 2020], can only provide a (non-optimal) bound for the cost of the coupon collector, whereas the specification established by Tachis above is a tight bound.

$$
\begin{array}{ll}
\textsf{rec repeatSwap } v\ i = & \textsf{shuffle} \triangleq \lambda\ v. \\
\quad \textsf{if } i \leq 0 \textsf{ then } v & \quad \textsf{let } i = (\textsf{List.length } v) - 1 \textsf{ in} \\
\quad \textsf{else let } j = \textsf{rand } i \textsf{ in} & \quad \textsf{repeatSwap } v\ i \\
\qquad \textsf{let } v' = \textsf{List.swap } v\ i\ j \textsf{ in} & \\
\qquad \textsf{repeatSwap } v'\ (i - 1) &
\end{array}
$$

Fig. 2. An implementation of the Fisher-Yates shuffle.

5.2 Fisher-Yates Shuffle

In this section, we use the $cost_{\mathrm{rand}}$ model to reason about the *expected entropy cost* of the *Fisher-Yates shuffle* [Durstenfeld 1964; Toutenburg 1971]. The Fisher-Yates shuffle is a classical randomized algorithm that generates a random permutation of a sequence by repeatedly sampling an element from the sequence without replacement. Figure 2 shows a RandML implementation of the algorithm. Starting from the last index i of a list v, we repeatedly sample an index j from rand i and swap the i-th and j-th elements of the list.

The Fisher-Yates shuffle is unbiased in the sense that every permutation is equally likely. Given that the input list is of length N, there is a total of $N!$ possible permutations, assuming each element in the list is distinct. Since each permutation is equally likely in an unbiased shuffle, no algorithm for unbiased shuffling can consume less than $\log_2(N!)$ entropy. We prove that the Fisher-Yates shuffle achieves this optimal entropy consumption in expectation, by proving an upper bound of $\log_2(N!)$ on its expected entropy cost. In particular, we show

$$
\left\{ \textsf{isList}(l, v) * \$\left(\log_2(|l|!)\right) \right\} \textsf{ shuffle } v \left\{ v'.\ \exists l'.\ \textsf{isList}(l', v') * l \equiv_p l' \right\}.
$$

Here $\textsf{isList}(l, v)$ is a standard representation predicate which means v corresponds to the mathematical list l, and $l \equiv_p l'$ denotes that l is a permutation of l'. The proof of the specification is mostly straightforward: for any natural number N, we have $\log_2(N!) = \sum_{i=2}^{N} \log_2(i) = \sum_{i=1}^{N-1} cost(\textsf{rand } i)$, which is exactly the cost needed to pay for each rand operation for a list of length N.

This example demonstrates the benefit of parameterizing by different cost models, instead of trying to use tick to instrument a program to register its total cost. In general, the $\log_2(i+1)$ entropy cost of calling rand i may be an irrational real number. Thus, instrumenting the program with a $\textsf{tick}(\log_2(i+1))$ to incur this cost would require adding primitives to the language for computing the $\log_2$ in the argument to tick.

5.3 Batch Sampling

It is a folklore result that the entropy cost of generating samples from various distributions can be reduced by generating samples in batches, instead of one at a time [Han and Hoshi 1997; Lumbroso 2013]. Samples that are generated in batches can be stored in a buffer and then returned one by one, only regenerating when the buffer is exhausted. In this section, we consider a concrete example described by Mennucci [2010] of how batching can reduce per-sample entropy. As we will see, reasoning about the buffering of batched samples is natural with amortized cost analysis.

Suppose we would like to simulate uniform sampling from a set of three elements, *e.g.*, the set $\{0, 1, 2\}$ but the only probabilistic primitive that we have at our disposal is rand 1, a probabilistic flip. One approach is to execute rand 1 twice and decode the combined output as a 2-digit binary

```
rec prefetch mem =                 initSampler ≜ let mem = ref 0 in
    let v = flipN 8 in                          let cnt = ref 0 in
    if v < 243 then                             λ _. (if ! cnt == 0 then
        mem ← v                                      prefetch mem; cnt ← 5);
    else prefetch m                             let v = ! mem in
                                                cnt ← ! cnt − 1;
                                                mem ← v `quot` 3;
                                                v `rem` 3
```

Fig. 3. An implementation of batch sampling.

number. If this number equals 3 we repeat the process, otherwise we return the decoded number.

$$\text{rec sampleThree } \_ = \text{let } v = (\text{rand } 1) + 2 * (\text{rand } 1) \text{ in}$$
$$\text{if } v < 3 \text{ then } v \text{ else sampleThree } ()$$

Since $cost(\text{rand } 1) = \log_2(1 + 1) = 1$, one can show that the expected entropy cost of one execution of sampleThree is $\frac{8}{3}$ by proving the following specification, which is mostly straightforward.

$$\left\{ \$ \left(\tfrac{8}{3} \right) \right\} \text{ sampleThree } () \{ n. \, 0 \leq n \leq 2 \}$$

Next, we consider a batch sampling scheme shown in Figure 3, which generates five rand 2 samples at once. The key advantage of sampling in batches is that the *amortized* expected cost will be strictly smaller than $\frac{8}{3}$, so over a long sequence of queries, it will be cheaper than sampleThree. When called, the function initSampler initializes state needed by the sampler, and then returns a batched sampling function that can be invoked to generate samples. This returned function uses a helper function prefetch, which executes rand 1 eight times, intuitively corresponding to sampling a natural number strictly below $2^8 = 256$. If the number is strictly below $3^5 = 243$, we can encode it as a 5-digit ternary number, where each of the digits represents an independent rand 2 sample. Otherwise, if the number is ≥ 243, we repeat the process of executing eight rand 1 samplings until the result is below 243. The result of prefetch is cached in the location mem, and then each of the generated digits are returned for the next 5 queries. The location cnt stores the number of queries left before a new batch is required. With this approach, we only incur a large cost at the start of every five queries, while the remaining four queries incur 0 cost, as we are only reading out the corresponding result from the batched sampling.

After an initial cost of $4 \cdot \frac{256 \cdot 8}{243 \cdot 5}$ to initialize the sampler, the amortized cost of each subsequent query is $\frac{256 \cdot 8}{243 \cdot 5}$ which is strictly smaller than $\frac{8}{3}$, the expected cost of our original sampleThree program. We establish this claim by showing the higher-order specification

$$\left\{ \$ \left(4 \cdot \tfrac{256 \cdot 8}{243 \cdot 5} \right) \right\} \text{ initSampler } \left\{ f. \, \pi * \left\{ \$ \left(\tfrac{256 \cdot 8}{243 \cdot 5} \right) * \pi \right\} f \, () \{ n. \, (0 \leq n < 3) * \pi \} \right\}$$

where

$$\pi \triangleq \exists \text{cnt}, c, \text{mem}, m. \, \text{cnt} \mapsto c \, * \, c < 5 \, * \, \text{mem} \mapsto m \, * \, m < 3^c \, * \, \$ \left((4 - c) \cdot \tfrac{256 \cdot 8}{243 \cdot 5} \right)$$

The π assertion serves two purposes: (1) it relates the locations cnt and mem and their contents, and (2) it serves as a reserve that stores the credits provided in previous invocations to initSampler. The reserve is increased by the intermediate queries that have actual cost 0—but non-zero amortized cost—and spent all at once for the expensive operations happening every fifth query.

We omit the details of the proof which, as in previous examples, relies on choosing how to distribute credits during the batch sampling through the expectation-preserving composition rule. The key take-away of this example is that by treating cost as a resource in separation logic, advanced reasoning about amortized expected cost is naturally supported.

5.4 Hash Map

In this example, we show that Tachis scales to reasoning about realistic data structures.

A hash map (also known as a hash table or a hash set) is a data structure widely used for implementing dictionaries, caches, and sets. We implement a hash map that supports two operations: insertion and lookup. Our hash map contains an array, and each array index contains a pointer to a linked list (called a bucket), which stores a list of elements. To insert an element, the hash map uses a hash function to hash the element, and appends it to the bucket with the array index of the hash. To perform a lookup on an element, the hash map hashes the element and traverses the bucket with the array index of the hash to determine whether it is present in the bucket.

In this example, we establish cost bounds on inserting and looking up elements in our hash map implementation. We use the $cost_{tick}$ model, where we incur $\$(1)$ whenever we perform an access to memory by manually adding a tick 1 operation after each dereference.

Our hash map utilizes a model of the idealized hash function (Figure 4) under the uniform hash assumption [Bellare and Rogaway 1993], which assumes that the hash function hf from a set of keys K to values V behaves as if, for each key k, the hash $hf(k)$ is randomly sampled from a uniform distribution over V, independently of all other keys. We implement this model by first initializing an empty mutable map hf. When we want to hash a key k, we first check the map to determine whether there is an element under that key. If so, we return the hash value stored in $hf(k)$. Otherwise, we sample a value uniformly from $V = \{0, \ldots, n\}$, store the value in hf with key k, and return it.

$$
\begin{aligned}
\text{compute\_hash} \triangleq \lambda\ hf\ k.\ &\text{match get } hf\ k \text{ with} \\
&\quad \text{Some}(v) \Rightarrow v \\
&\quad | \text{ None} \quad \Rightarrow \text{let } v = \text{rand } n \text{ in set } hf\ k\ v\ ;\ v \\
&\quad \text{end}
\end{aligned}
$$

Fig. 4. An idealized hash function implementation.

Although the implementation of the hash function involves location accesses, we do not include tick operations in our hash implementation. This is because our implementation only serves to simulate the effects of a realistic hash function, which in real life, usually incurs constant cost for each hash operation.

The two main operations supported by the hash map are insert and lookup as shown in Figure 5. The function insert hashes an element v with the hash function hf to get the index idx of the bucket that v should go into and accesses the bucket idx with a dereference, incurring one tick. Finally, insert stores v at end of the linked list representing the bucket after checking for the presence of v to avoid duplication. The function lookup similarly computes idx by hashing v, accessing the bucket idx with a load and hence incurring one tick cost, and finally performing a lookup by traversing the linked list. We omit the (standard) code for the linked list functions LinkedList.insert and LinkedList.lookup, but note that since they traverse the entire linked list, and memory dereferencing costs 1 in our model, the functions incur $\$(len)$ credits for each invocation where len is the length of the linked list.

insert $\triangleq \lambda\, h\, v.$ lookup $\triangleq \lambda\, h\, v.$

     let $(l, hf) = h$ in   let $(l, hf) = h$ in

     let $idx = $ compute_hash $hf\, v$ in   let $idx = $ compute_hash $hf\, v$ in

     let $b = \,!\,l[idx]$ in   let $b = \,!\,l[idx]$ in

     tick 1;   tick 1;

     $(l +_\ell idx) \leftarrow$ LinkedList.insert $b\, v$ LinkedList.lookup $b\, v$

Fig. 5. A hash map implementation.

Let us consider the costs of insert and lookup in the case where an element has never been hashed by a hash function hf before. For insert, when we first hash the element, since it has not been hashed by hf before, the idealized hash function randomly picks an index idx from $V = \{0, \ldots, n\}$. Let the cost of insert in the case that the element is hashed to idx by hf be $C(idx)$. By stepping through the program, we can show that $C(idx) = 1 + \mathsf{hmLength}(h, idx)$, where $\mathsf{hmLength}(h, idx)$ is notation for the length of the linked list representing the bucket with index idx. Hence the expected cost of insert is $\mathbb{E}_\mu[C]$, where μ is the uniform distribution on $\{0, \ldots, n\}$. By linearity of expectations, we have $\mathbb{E}_\mu[C] = \mathbb{E}_\mu[1] + \mathbb{E}_\mu[\mathsf{hmLength}(h, idx)] = 1 + \frac{\mathsf{size}(h)}{n+1}$ where $\mathsf{size}(h)$ is shorthand for the total number of elements stored in the hash map (this is the sum of lengths of all the linked list buckets). The time complexity of lookup is exactly the same, as instead of adding an element to the end of a linked list in insert, we still traverse the linked list comparing our element with each value stored by the linked list.

To verify the above claims, we prove that the expected cost of inserting and looking up an element not in the hash map is proportional to the size of the hash map, *c.f.*, the specifications below. In these specifications, the predicate $\mathsf{isHashMap}(h, m_1, m_2)$ intuitively asserts that the hash map h is valid according the two mathematical maps m_1 and m_2. The map m_1 models the map implementing the hash function and m_2 models the linked lists of each bucket of the hash map.

$$\left\{ v \notin \mathsf{dom}(m_1) * \mathsf{isHashMap}(h, m_1, m_2) * \$\left(1 + \tfrac{\mathsf{size}(h)}{n+1}\right) \right\}$$

     insert $h\, v$

$$\{\_.\, \exists i.\, \mathsf{isHashMap}(h, m_1[v \leftarrow i], m_2[i \leftarrow m_2[i] \,+\!+\, [v]])\}$$

$$\left\{ v \notin \mathsf{dom}(m_1) * \mathsf{isHashMap}(h, m_1, m_2) * \$\left(1 + \tfrac{\mathsf{size}(h)}{n+1}\right) \right\}$$

     lookup $h\, v$

$$\{b.\, b = \mathsf{false} * \exists i.\, \mathsf{isHashMap}(h, m_1[v \leftarrow i], m_2)\}$$

In addition, we also prove that by fixing the maximum size of the hash map, we can show an amortized specification for the insertion of new elements into the hash map[2], where the cost of insertion is a fixed constant x.

$$\{v \notin \mathsf{dom}(m_1) * \mathsf{size}(h) < \mathsf{MAX} * \mathsf{isAmortizedHashMap}(h, m_1, m_2) * \$(x)\}$$

     insert $h\, v$

$$\{\_.\, \exists i.\, \mathsf{isAmortizedHashMap}(h, m_1[v \leftarrow i], m_2[i \leftarrow m_2[i] \,+\!+\, [v]])\}$$

---

[2]There is no natural amortized specification analogue of lookup, as the function does not change the size or structure of the hash map, and hence each invocation has the same cost.

In textbook proofs of the case where an element v has previously been hashed (an thus exists in the hash table), one proves that the cost of insert (and lookup) is also proportional to the size of the hash table, assuming that v is chosen uniformly across all elements stored in the hash table. This assumption is not directly expressible within Tachis, and we leave this problem as future work.

5.5 Randomized Quicksort

A common technique in randomized algorithm design is *divide-and-conquer*, where a program decomposes a problem into several "smaller" subproblems, often based on a random choice. A cost analysis of a divide-and-conquer algorithm often involves first modeling the program using a recurrence relation, and then deriving upper bounds or closed forms for the model mathematically. Distinguishing the *modeling* and *bounding* stages can be advantageous for mechanized program verification, since the recurrence relations in the *modeling* step are typically exact (eliminating the re-engineering work from guessing a bound that is "too loose") and more straightforward to establish. To show how the *expectation rule* can simplify cost bound proofs for divide-and-conquer algorithms, we verify an implementation of randomized quicksort in Tachis.

```
rec qSort cmp xs =
    let n = List.length xs in
    if n < 1 then xs else
        let i = rand (n − 1) in
        let (pivot, rest) = List.remove_nth xs i in
        let xslt = List.filter (λx. not (cmp pivot x)) rest in
        let xsgt = List.filter (λx. cmp pivot x) rest in
        let (xslt′, xsgt′) = (qSort cmp xslt, qSort cmp xsgt) in
        List.append xslt′ (List.cons pivot xsgt′)
```

Fig. 6. An implementation of Quicksort.

Figure 6 depicts a higher-order RandML implementation of quicksort. The program operates on functional lists of generic values, using a function cmp to compare them. As is common in the analysis of sorting algorithms we will only count the costs incurred by comparisons; we can parametrically calculate the total expected cost incurred by cmp by using the $cost_{tick}$ model since qSort contains no tick statements. Let R be a decidable, total order on a type A, and let ι be an injection from A to *Val*. We will use ι to specify that RandML values are ordered according to R. We define a predicate isPureComp$^\iota$ to specify that cmp correctly computes comparisons with respect to the order R, incurring a bounded cost k:

$$\text{isPureComp}^\iota(c, R, k) : iProp \triangleq \forall(x, y : A), \{\$(k)\}\ c\ (\iota\ x)\ (\iota\ y)\ \{b.\ b = R\ x\ y\}$$

Given a constant $m \geq 0$ define $\mathrm{t}_m(n)$ by the recurrence relation

$$\mathrm{t}_m(n) \triangleq \begin{cases} 0 & n = 0 \\ mn + \frac{2}{n} \sum_{i=0}^{n-1} \mathrm{t}_m(i) & n > 0 \end{cases}$$

A standard mathematical derivation (which we will omit here, but which is formalized in our Coq development) shows that $\mathrm{t}_m(n)$ is bounded above by $(n+1) \sum_{i=1}^{n} \frac{2m}{i+1}$. It is well known that the harmonic series is $O(\log n)$ so establishing a cost bound of $\mathrm{t}_m(n)$ for any m is enough to obtain the familiar $O(n \log n)$ asymptotic result.

We can now move on to specifying and verifying our implementation of qSort. Let L be a list of values of type A without duplicates, and let v be a value. By applying ι to each element, define the predicate $\mathrm{isList}(L, v)$ to hold when v is a value-level reflection of L. We then seek to prove that

$$\{\$(\mathrm{t}_{2k}(|L|)) \ast \mathrm{isList}(L, v) \ast \mathrm{NoDup}(L) \ast \mathrm{isPureComp}^\iota(\mathrm{cmp}, R, k)\}$$

$$\mathrm{qSort\ cmp\ } v$$

$$\left\{v'. \exists L'.\ L \equiv_p L' \ast \mathrm{Sorted}(L') \ast \mathrm{isList}(L', v')\right\}$$

The predicates in the postcondition state the functional correctness of quicksort, namely that the returned list is a sorted permutation of the input. Since the functional correctness is irrelevant for the runtime bound we will elide that part of the argument here, and just focus on explaining the cost accounting. We proceed by induction using HT-REC. Set $N = |L|$ and, after eliminating the $N = 0$ edge case, split $\$(\mathrm{t}_{2k}(N))$ into $\$(kN + kN)$ and $\$\left(\frac{2}{N} \sum_{i=0}^{N-1} \mathrm{t}_{2k}(i)\right)$. We will spend the former credit to execute the comparisons in the two List.filter statements, as in our implementation their cost will be the same regardless of the pivot element selected by rand. We will eventually use the latter credit to pay for the induction hypothesis during the recursive calls to qSort.

When the sampled pivot is close to the largest or smallest element in the list the latter credit is not be enough to pay for the recursive calls (conversely, we have a credit excess when the pivot is near the median). To ensure that we can apply the induction hypothesis in all cases we must redistribute the credits at the statement rand $(n - 1)$ based on the rank of the pivot. Define

$$\mathrm{ix\_rk}_L(i) \triangleq |\{j \in L \mid j < L[i]\}|$$

$$\mathrm{rk\_rev}_L(i) \triangleq N - 1 - i$$

The function $\mathrm{ix\_rk}_L$ converts an index i in L into the rank of $L[i]$, and the function $\mathrm{rk\_rev}_L$ relates the number of elements greater than the pivot to the rank of the pivot itself. Crucially both functions are bijections on $\{0, \ldots, N - 1\}$, the sample space of rand $(N - 1)$, so post-composing a sum over that space with $\mathrm{ix\_rk}_L$ or $\mathrm{rk\_rev}_L$ will not change its value. Using our bijections, we can define an credit distribution function.

$$d_L(i) = (\mathrm{t}_{2k} \circ \mathrm{ix\_rk}_L)(i) + (\mathrm{t}_{2k} \circ \mathrm{rk\_rev}_L \circ \mathrm{ix\_rk}_L)(i)$$

We can show that the mean of d_L is exactly the credit remaining after framing out $\$(kN + kN)$, meaning we can apply HT-RAND-EXP using d_L at rand $(n - 1)$:

$$\frac{1}{N} \sum_{i=0}^{N-1} d_L(i) = \frac{1}{N} \sum_{i=0}^{N-1} (\mathrm{t}_{2k} \circ \mathrm{ix\_rk}_L)(i) + \frac{1}{N} \sum_{i=0}^{N-1} (\mathrm{t}_{2k} \circ \mathrm{rk\_rev}_L \circ \mathrm{ix\_rk}_L)(i)$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} \mathrm{t}_{2k}(i) + \frac{1}{N} \sum_{i=0}^{N-1} \mathrm{t}_{2k}(i) = \frac{2}{N} \sum_{i=0}^{N-1} \mathrm{t}_{2k}(i)$$

Therefore, when rand $(n - 1)$ selects the pivot index i, we can use $\$(kN + kN + d_L(i))$ to verify the continuation. Since $\mathrm{ix\_rk}_L(i) = \mathrm{length}\ lt$ and $(\mathrm{rk\_rev}_L \circ \mathrm{ix\_rk}_L)(i) = \mathrm{length}\ gt$ this is precisely enough credit to filter the lists and then apply the induction hypothesis, completing the proof.

Verifying the Expected Entropy Cost for Quicksort. Using Tachis to establish a recurrence relation for a cost bound and then separately analyzing the recurrence leads to more modular proofs and proof re-use. For example, when we change cost models, often the overall *structure* of a recurrence relation will not change, and so the same proof can be adapted to establish a recurrence under this different model, even if analyzing the asymptotics of the recurrence or computing a closed form would be quite different. To illustrate this, we verify a recurrence for the expected

$$
\begin{aligned}
\text{meld} \triangleq\ &\mathsf{rec}\ \text{meld}\ \ \text{cmp}\ v_1\ v_2 = \\
&\mathsf{match}\ v_1\ \mathsf{with}\ \ \mathsf{None} \Rightarrow v_2\ |\ \mathsf{Some}\ \mathsf{h}_1 \Rightarrow \\
&\mathsf{match}\ v_2\ \mathsf{with}\ \ \mathsf{None} \Rightarrow v_1\ |\ \mathsf{Some}\ \mathsf{h}_2 \Rightarrow \\
&\mathsf{let}\ ((\mathsf{r}_{\min},\ (\mathsf{hl}_{\min},\ \mathsf{hr}_{\min})),\ \mathsf{h}_{\max}) = \\
&\quad (\mathsf{if}\ \mathsf{cmp}\ (\mathsf{fst}\ \mathsf{h}_1)\ (\mathsf{fst}\ \mathsf{h}_2)\ \mathsf{then}\ (\mathsf{h}_1, \mathsf{h}_2)\ \mathsf{else}\ (\mathsf{h}_2, \mathsf{h}_1))\ \mathsf{in} \\
&\mathsf{if}\ \mathsf{flip} \\
&\quad\ \mathsf{then}\ \mathsf{Some}\ (\mathsf{r}_{\min},\ (\text{meld}\ \mathsf{cmp}\ \mathsf{hl}_{\min}\ \mathsf{Some}(\mathsf{h}_{\max}),\ \mathsf{hr}_{\min})) \\
&\quad\ \mathsf{else}\ \mathsf{Some}\ (\mathsf{r}_{\min},\ (\mathsf{hl}_{\min},\ \text{meld}\ \mathsf{cmp}\ \mathsf{hr}_{\min}\ \mathsf{Some}(\mathsf{h}_{\max}))) \\
&\mathsf{end}\ \mathsf{end} \\[4pt]
\text{insert} \triangleq\ &\lambda\mathsf{cmp}, \mathsf{r}, v.\ (\mathsf{r} \leftarrow \text{meld}\ \mathsf{cmp}\ (!\,\mathsf{r})\ \mathsf{Some}(v, (\mathsf{None}, \mathsf{None}))) \\[4pt]
\text{remove} \triangleq\ &\lambda\mathsf{cmp}, \mathsf{r}. \\
&\mathsf{match}\ !\,\mathsf{r}\ \mathsf{with} \\
&\quad \mathsf{None} \Rightarrow \mathsf{None} \\
&\quad |\ \mathsf{Some}\ (v,\ (\mathsf{hl},\ \mathsf{hr})) \Rightarrow (v \leftarrow \text{meld}\ \mathsf{cmp}\ \mathsf{hl}\ \mathsf{hr});\ \mathsf{Some}\ v \\
&\mathsf{end}
\end{aligned}
$$

Fig. 7. An implementation of randomized meldable heaps.

amount of entropy consumed by the random choices in qSort. Define a recurrence relation $e(n)$ which represents the entropy required to sort a list of length n:

$$
e(n) \triangleq \begin{cases} 0 & n = 0 \\ \log_2(n) + \frac{2}{n}\sum_{i=0}^{n-1} e(i) & n > 0 \end{cases}
$$

Given a comparison function with zero entropy cost, an argument directly analogous to the time complexity proof shows that $\$(e(n))$ suffices to execute qSort on lists of length n under the $cost_{\mathrm{rand}}$ model. The proofs primarily differ by where they spend credit: the $cost_{\mathrm{tick}}$ proof spends credit inside List.filter, whereas the $cost_{\mathrm{rand}}$ proof spends credit to execute rand. The details of the expectation-preserving transfer of credits across branches and other steps of program execution are otherwise nearly identical, even reusing much of their lemmas. It is known that $e(n)$ is asymptotically linear [Cover and Thomas 2006, p. 89-91], but solving this recurrence or computing the asymptotics of $e(n)$ is quite different from the analysis of $t_m(n)$, the recurrence for the cost of comparisons. Thus, if the logic had required solving or bounding the recurrence at the same time as we analyzed the program, we would not be able to re-use as much of the proof.

5.6 Randomized Heaps

A *randomized meldable heap* [Gambin and Malinowski 1998] is an implementation of a binary heap, which use randomization to perform insert and remove using an expected $O(\log(n))$ comparisons. The key to this performance is the meld operation, which recursively combines two meldable heaps by melding the heap with greater root into a random child of the heap with lesser root. Choosing a child of the lesser heap at random ensures the *expected* size of a heap in the recursive call to meld is halved, regardless of how unbalanced the children may be. This is essential to obtaining the expected cost bound. Below, we verify an implementation of meldable heaps against a modular heap specification, using expectation preserving composition to obtain expected cost bounds.

$$\text{isComp}(K, \text{cmp}, \text{hasKey}) \triangleq \exists R : K \to K \to \mathbb{B}, x : \mathbb{R}_{\geq 0}. \, \text{PreOrder}(R) \land \text{Total}(R) \land$$
$$\{\text{hasKey}(k_1, v_2) * \text{hasKey}(k_2, v_2) * \$(x)\}$$
$$\text{cmp } v_1 \, v_2$$
$$\{b. \, b = R(k_1, k_2) * \text{hasKey}(k_1, v_2) * \text{hasKey}(k_2, v_2)\}$$

Fig. 8. A specification for an abstract comparator.

$$\text{isComp}(K, \text{cmp}, \text{hasKey}) \Rightarrow$$
$$\exists \text{isHeap} : List(K) \to Val \to iProp, X_i, X_r : \mathbb{N} \to \mathbb{R}_{\geq 0}.$$
$$(\forall n, m. \, n \leq m \Rightarrow X_i(n) \leq X_i(m)) \land (\forall n, m. \, n \leq m \Rightarrow X_r(n) \leq X_r(m))$$
$$\land \, \{\text{True}\} \, \text{new} \, () \, \{v. \, \text{isHeap}([], v)\}$$
$$\land \, \{\text{isHeap}(l, v) * \text{hasKey}(k, w) * \$(X_i(|l|))\} \, \text{insert} \, v \, w \, \{\_. \, \exists l'. \, \text{isHeap}(l', v) * l \equiv_p (k :: l')\}$$
$$\land \, \{\text{isHeap}(l, v) * \$(X_r(|l|))\}$$
$$\quad \text{remove} \, v$$
$$\left\{ w. \begin{array}{l} (w = \text{None} \, * \, l = [] \, * \, \text{isHeap}([], v)) \\ \lor \, (\exists u, k, l'. \, w = \text{Some} \, u \, * \, l \equiv_p (k :: l') \, * \, \min(k, l) \, * \, \text{hasKey}(k, u) \, * \, \text{isHeap}(l', v)) \end{array} \right\}$$

Fig. 9. An abstract specification for a min-heap.

[Figure 7](#) depicts a RandML implementation of meldable heaps, and [Figure 9](#) outlines our target specification. Our heap specification is parameterized by a *comparator*, a RandML program which we specify (in [Figure 8](#)) to implement a total preorder using a bounded cost x. The comparator specification also includes a resourceful representation predicate $\text{hasKey} : K \to Val \to iProp$, connecting comparable elements of type K to a particular RandML values.[3] In [§5.7](#) we will apply the heap specification with a concrete comparator instance, demonstrating the modularity of cost bounds proven in Tachis.

For the remainder of this section, we assume that A, cmp, and key_{cmp} satisfy the predicate $\text{isComp}(A, \text{cmp}, \text{key}_{\text{cmp}})$ with some cost bound $k \geq 0$ and total preorder R.

Define a meta-level type BinaryTree of binary trees at type A, and a function treeToList which produces a list of the elements stored in a binary tree in an unspecified order. We represent a binary tree in RandML as either None when the tree is empty, or Some $(v, (l, r))$ when the tree is a node with value v and children l and r. By lifting key_{cmp} to the nodes of this structure we obtain a representation predicate for binary trees, denoted $\text{isBinaryTree} : \text{BinaryTree} \to Val \to iProp$.

We define the predicate heap, which holds when a meta-level binary tree is heap-ordered with respect to R. Together, we can define a specification isMeldHeapVal for RandML-level heaps in terms of their list of elements L: the value must represent a binary tree, which is a heap with respect to R, and whose elements are a permutation of L.

$$\text{isMeldHeapVal}(v, L) \triangleq \exists b. \, \text{isBinaryTree}(b, v) * \text{heap } b * L \equiv_p \text{treeToList}(b)$$

We can also lift this specification to references, which we use as our instance for isHeap in [Figure 9](#).

$$\text{isMeldHeapRef}(\ell, L) \triangleq \exists h. \, \ell \mapsto h * \text{isMeldHeapVal}(h, L)$$

---

[3]We remark that this differs from the representation predicate in [§5.5](#), since our implementation of List.filter does not respect linearity, and qSort is specified against a partial order, rather than a preorder.

Under this definition it is straightforward to implement and verify new, which allocates a reference to a new empty heap.

Moving on, we will prove a specification for meld in Tachis. We define the cost associated to melding a heap of size n with the following function:

$$t^{\text{meld}}(n) \triangleq \begin{cases} 0 & n = 0 \\ 2k(1 + \log_2(n)) & n > 0 \end{cases} \tag{3}$$

Using $|\cdot|$ to denote both the length of a list and the size of a heap, we define a Hoare triple for meld which encompasses both the cost bound and the functional correctness.

$$\left\{ \text{isMeldHeapVal}(v_1, L_1) * \text{isMeldHeapVal}(v_2, L_2) * \$\left(k + t^{\text{meld}}(|L_1|) + t^{\text{meld}}(|L_2|)\right) \right\}$$

$$\text{meld } v_1 \, v_2$$

$$\left\{ v'. \exists L. \, \text{isMeldHeapVal}(v', L) * L \equiv_p L_1 + L_2 \right\}$$

We proceed by HT-REC. After handling base cases, the body of meld compares the roots of the two heaps using cmp, incurring a cost of k credits. We denote the minumum- and maxumum-rooted heap $\mathsf{h_{min}}$ and $\mathsf{h_{max}}$ respectively, by heap ordering we know that no elements of $\mathsf{h_{max}}$ are lesser than the root $\mathsf{r_{min}}$ of $\mathsf{h_{min}}$. At this point we own $t^{\text{meld}}(|\mathsf{h_{max}}|)$ and $t^{\text{meld}}(1 + |\mathsf{hl_{min}}| + |\mathsf{hr_{min}}|)$ credits. One can observe that depending on the balance of $\mathsf{h_{min}}$ this may not be enough credits to directly apply the inductive hypothesis; as we alluded to in the introduction, we will obtain the correct number of credits using an expectation-preserving composition.

$$\mathsf{tc}^{\text{dist}}(s) \triangleq \begin{cases} k + t^{\text{meld}}(|\mathsf{hl_{min}}|) & s = 0 \\ k + t^{\text{meld}}(|\mathsf{hr_{max}}|) & s = 1 \end{cases}$$

In our development we show that $t^{\text{meld}}(|\mathsf{h_{min}}|)$ is enough credit to apply the expectation rule with distribution function $\mathsf{tc}^{\text{dist}}$, in particular:

$$[k + t^{\text{meld}}(|\mathsf{hl_{min}}|) + (k + t^{\text{meld}}(|\mathsf{hr_{min}}|))]/2 \le t^{\text{meld}}(1 + |\mathsf{hl_{min}}| + |\mathsf{hr_{min}}|)$$

Therefore by framing, when rand 1 samples s we will have total credit balance $t^{\text{meld}}(|\mathsf{h_{max}}| + \mathsf{tc}^{\text{dist}}(s))$. Suppose without loss of generality that rand 1 samples 0. Spending $t^{\text{meld}}(|h_{\text{max}}|) + k + t^{\text{meld}}(|hl_{\text{min}}|)$ credits, we can apply the inductive hypothesis to evaluate the recursive call, meld cmp $\mathsf{hl_{min}}$ Some($\mathsf{h_{max}}$). The resulting value is certainly a heap with no elements greater than r, as none of its inputs were, so our return value also does not violate the heap order. Furthermore, the returned heap has precisely the same elements as $L_1 + L_2$ up to permutation. This establishes the postcondition.

Because insert and remove both make one call to meld it is straightforward to prove the remaining specifications in Figure 9 from the specification for meld, with the insertion and removal cost bounds $X_i(N) = k + t^{\text{meld}}(N)$ and $X_r(N) = k + 2t^{\text{meld}}(N)$.

Leutgeb et al. [2022] describe a tool that can derive the cost bound for melding automatically. However, the tool's input language is a first-order functional language that does not have mutable state. Thus, it cannot handle the parameterized comparator we consider here, which is essential for an example that makes use of the heaps that we describe next in §5.7.

5.7 K-way Merge

The *k-way merge* problem [Cormen et al. 2009] consists of merging k sorted lists to produce a single sorted list with the same elements. One way to solve the problem is by maintaining a min-heap of the k lists, each keyed by the head of the (sorted) list [Bentley 1986]. The procedure then

```
rec repeatRemove h r =                    merge ≜ λzss. let h = new () in
    match remove h with                       List.iter (insert h) zss;
      Some zs ⇒                               let r = ref (List.new ()) in
        match List.head zs with               repeatRemove h r;
          Some z ⇒                            List.reverse ! r
            r ← List.cons z (! r)
            insert h (List.tail zs);
            repeatRemove h r
        | None ⇒ repeatRemove h r
        end
    | None ⇒ ()
    end
```

Fig. 10. A implementation of k-way merge.

repeatedly extracts the minimal element from the heap, adding the head element to the output buffer, and reinserting the remaining list into the heap. Figure 10 shows a RandML implementation of the algorithm that uses *some* min-heap implementation with operations new, insert, and remove. To showcase the modularity of Tachis, we verify the k-way merge implementation in isolation, assuming *only* the existence of an implementation that satisfies the abstract min-heap specification shown in Figure 9. In the end, we compose this specification with the meldable heap specification from §5.6 and obtain the expected $O(n \log k)$ cost bound in a fully modular way.

To verify merge, we instantiate the isComp interface from Figure 8 with a comparator that compares lists by comparing their head elements. The comparator induces cost 1 per invocation by including a tick 1 statement. Using the abstract min-heap specification, we prove

$$\left\{ \text{isList}(zss, v) * (\forall zs \in zss.\ \text{Sorted}_{\leq}(zs)) * \$\left(x_{\text{merge}}\right) \right\}$$
$$\quad \text{merge } v$$
$$\left\{ w. \exists zs.\ \text{isList}(zs, w) * zs \equiv_p \text{concat}(zss) * \text{Sorted}_{\leq}(zs) \right\}$$

where $x_{\text{merge}} = (k + n) \cdot X_i(k) + X_r(0)$ denoting $|\text{concat}(zss)|$ by n and $|zss|$ by k. Note that X_i and X_r are abstract cost functions unknown to the specification of merge. The proof of the specification contains no probabilistic reasoning but manages the cost as an abstract entity required as a precondition for the heap operations. The k-way merge proof showcases the extensive modularity supported by Tachis both in terms of logical abstractions, cost, and implementations.

6 The Semantic Model and the Soundness Theorem

The semantic model of Tachis is simplified by working with *weakest preconditions* instead of Hoare triples directly. The two notions are well-known to be equivalent, and we use a standard definition [Jung et al. 2018, §6] of Hoare triples in terms of weakest preconditions:

$$\{P\}\ e\ \{Q\} \triangleq \Box(P \twoheadrightarrow \text{wp } e\ \{Q\})$$

With this definition in hand, we will now turn our attention to the development of the weakest precondition and then prove adequacy of Tachis for cost and functional correctness.

6.1 The Weakest Precondition, Cost Credits, and the Expected Cost Modality

The weakest precondition associated to an expression e_1 for a postcondition Φ is defined as a guarded recursive fixed point.[4] The *cost interpretation* $\$(x)$ and $\$_\bullet(x)$ and the *expected cost modality* ECM constitute the main novelty of our model.

$$
\begin{aligned}
\mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \triangleq\ & (e_1 \in \mathit{Val} \wedge \Rrightarrow_{\mathcal{E}} \Phi(e_1)) \\
& \vee\, (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, x_1.\, S(\sigma_1) * \$_\bullet(x_1) \mathrel{-\!\!*}{}_{\mathcal{E}}\!\Rrightarrow_{\emptyset} \\
& \quad \mathrm{ECM}(\underbrace{e_1,\ \sigma_1,}_{\rho_1}\ x_1,\ \underbrace{(\lambda\, e_2, \sigma_2, x_2\,.\, \triangleright_{\emptyset}\!\Rrightarrow_{\mathcal{E}} (S(\sigma_2) * \$_\bullet(x_2) * \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\})))}_{Z}
\end{aligned}
$$

where $\mathrm{ECM}(\rho_1, x_1, Z) \triangleq \exists\, (X_2 : \mathit{Cfg} \to \mathbb{R}_{\geq 0})\,.$

$$
\mathrm{red}(\rho_1) * \exists r.\, \forall \rho_2.\, X_2(\rho_2) \leq r * \tag{4}
$$

$$
\mathit{cost}(\rho_1) + \textstyle\sum_{\rho_2 \in \mathit{Cfg}} \mathrm{step}(\rho_1)(\rho_2) \cdot X_2(\rho_2) \leq x_1 * \tag{5}
$$

$$
\forall \rho_2.\, \mathrm{step}(\rho_1)(\rho_2) > 0 \mathrel{-\!\!*} Z(\rho_2, X_2(\rho_2)) \tag{6}
$$

If e_1 is a value and $\Phi(e_1)$ holds, then we can prove the first branch of the disjunction. Otherwise, under the assumption that the *state interpretation* $S(\sigma_1)$ and the *cost interpretation* $\$_\bullet(x_1)$ hold, we have to prove that the *expected cost modality* ECM holds for the current configuration (e_1, σ_1) and cost x_1. The state interpretation ties the physical state of the program, which evolves according to the operational semantics, to the logical presentation of the heap, *i.e.*, the $\ell \mapsto v$ proposition. The last argument to ECM, let us call it Z, contains a guarded recursive occurrence of the weakest precondition under a later modality $\triangleright$, which ensures existence of the guarded fixpoint.

The cost credit resource algebra. The cost interpretation $\$_\bullet(x_1)$ connects the cost budget $\$(x)$ that a user of the logic manipulates to the cost incurred by the reductions of the program e_1 according to the operational semantics. This ensures that the cost credits evolve in a way that matches the cost of the program. We achieve this connection by defining a new (unital) resource algebra $\mathsf{Auth}(\mathbb{R}_{\geq 0}, +)$ of non-negative real numbers, where $\$_\bullet(x)$ denotes the authoritative view, and $\$(x)$ stands for ownership of a fragmental view. The construction comes with an *agreement* rule $\$(x_1) * \$_\bullet(x_2) \vdash x_1 \leq x_2$, a rule that allows *spending* credits by updating $\$(x_1) * \$_\bullet(x_1 + x_2)$ to $\$(x_2)$, and a rule for *acquiring* credits by updating $\$_\bullet(x_1)$ to $\$(x_2) * \$_\bullet(x_1 + x_2)$. The cost splitting rule from §4.1 follows similarly from the laws for $(\mathbb{R}_{\geq 0}, +)$.

The expected cost modality. The flexibility of Tachis to split a cost budget so long as it is preserved in expectation stems from the way the cost credits are handled in the expected cost modality ECM. The existential quantifier over X_2 allows us to specify a family of costs indexed by configurations. The first two clauses (4) of ECM serve bureaucratic purposes: $\mathrm{red}(\rho_1)$ enforces that ρ_1 is reducible (hence no verified program will get "stuck"), and r, an upper bound to X_2, is required for the existence of the sum in the following line.

The heart of the matter is the weighted sum (5). It requires us to distribute the currently available cost budget x_1 between the cost for stepping the current configuration $\mathit{cost}(\rho_1)$ and the cost for the remainder of the program $X_2(\rho_2)$. Crucially, the cost for ρ_2 is *weighted* by the probability of stepping to it from ρ_1.[5]

Finally, by (6), one has to prove that for any configuration ρ_2 reachable in one step, the predicate Z holds with the corresponding cost $X_2(\rho_2)$. Since we applied ECM in the definition of the weakest

---

[4]We have greyed out the "masks" $\mathcal{E}$ and "update modalities" $\Rrightarrow_{\mathcal{E}}$, since they are (1) orthogonal to the issues discussed here and (2) our use of them in the definition of the weakest precondition is standard; see, *e.g.*, Jung et al. [2018] for details.
[5]The sum thus only "counts" configurations that are reachable from ρ_1, as the probability would be 0 otherwise.

precondition with $Z(e_2, \sigma_2, x_2) = \triangleright(S(\sigma_2) * \$_\bullet(x_2) * \text{wp } e_2 \{\Phi\})$, this amounts to carrying on proving the weakest precondition of the new expression e_2 with the updated physical state σ_2 and cost budget $x_2 = X_2(e_2, \sigma_2)$.

6.2 Soundness for Cost and Correctness

With the definition of the weakest precondition in hand, we can now prove that Tachis is a sound method for establishing cost bounds and functional correctness of RandML programs.

THEOREM 3 (ADEQUACY). *Let x be a non-negative real number and let φ be a predicate on values. If $\$(x) \vdash \text{wp } e \{\varphi\}$ then for any state σ,*

(1) $\text{EC}(e, \sigma) \leq x$, and
(2) $\forall v \in Val.\ \text{exec}(e, \sigma)(v) > 0 \implies \varphi(v)$.

Theorem 2 then follows directly from the definition of Hoare triples.

PROOF. We will focus on the salient steps of the proof of (1); (2) is similar and details can be found in the accompanying Coq formalization. Since $\text{EC}(e, \sigma)$ is defined as the supremum of $\text{EC}_n(e, \sigma)$ over $n \in \omega$, it suffices to show $\text{EC}_n(e, \sigma) \leq x$ for all n. This in turn follows the standard soundness theorem of the Iris step-indexed logic [Jung et al. 2018] if we can derive $\vdash \triangleright^n \text{EC}_n(e, \sigma) \leq x$ in Iris.

We proceed by induction on n. The base case, $\vdash \text{EC}_0(e, \sigma) \leq x$, is trivial since $\text{EC}_0(e, \sigma) = 0$. Consider the inductive step with $n = m + 1$ and $e \notin Val$. In this case, the assumption amounts to

$$\text{ECM}(e, \sigma, x, (\lambda e_2, \sigma_2, x_2.\ \triangleright\ (S(\sigma_2) * \$_\bullet(x_2) * \text{wp } e_2 \{\varphi\}))). \tag{7}$$

We replace the recursive occurrence of wp $e_2 \{\varphi\}$ with the expected cost of (e_2, σ_2) by applying the induction hypothesis (8) under the cost modality in (7).

$$\forall e_2, \sigma_2, x_2.\ S(\sigma_2) * \$_\bullet(x_2) * \text{wp } e_2 \{\varphi\} \twoheadrightarrow \triangleright^m \text{EC}_m(e_2, \sigma_2) \leq x_2 \tag{8}$$

We are thus left to show the following entailment:

$$\text{ECM}(e, \sigma, x, (\lambda e_2, \sigma_2, x_2.\ \triangleright(\triangleright^m \text{EC}_m(e_2, \sigma_2) \leq x_2))) \vdash \triangleright^{m+1} \text{EC}_{m+1}(e, \sigma) \leq x$$

This fact in turn is proven by a careful rearrangement of the weighted sums arising from the definitions of ECM and EC. □

We remark here that we have also proven that a program with finite expected cost for the $cost_{\text{all}}$ model terminates almost-surely (*i.e.*, $\text{exec}_\Downarrow(e, \sigma) = 1$). This is of course no surprise, but rather serves as a check that our definitions relating to costs have the usual expected properties.

7 Related and Future Work

7.1 Related Work

Logics for expected runtime. Formal methods to reason about the expected cost of first-order programs have seen a surge in recent years. For instance, the line of work by Kaminski et al. [2016] on expected runtime transformers generalize the weakest pre-expectation transformer by McIver and Morgan [2005] to reason about the expected running time of programs. This work has multiple extensions including a recent one to reason about amortized expected cost [Batz et al. 2023], which makes use of separation logic assertions by adapting the approach of quantitative separation logic (QSL) [Batz et al. 2019]. In these logics, assertions are interpreted as functions of type State $\to \mathbb{R}_{\geq 0}^\infty$. Program reasoning is done using an expected-runtime transformer, ert, which is defined in such a way that $\text{ert}(c)\{\mathbf{0}\}$ is a function that maps a state σ to the expected running time of c starting from σ. The analogue of Tachis' HT-RAND-EXP arises from the definition of ert for the command $c_1[p]c_2$, which executes c_1 with probability p and c_2 with probability

$(1-p)$. This command's ert is defined by $\text{ert}(c_1[p]c_2)\{f\} = p \cdot \text{ert}(c_1)\{f\} + (1-p) \cdot \text{ert}(c_2)\{f\}$, taking the weighted average of the ert for each command. In contrast to these logics, in Tachis, assertions are not given a quantitative interpretation and have the standard resource algebra semantics found in Iris. A benefit of our approach is that for reasoning about non-cost-related parts of program correctness, Tachis inherits all of Iris' standard, expressive reasoning rules. The eHL logic [Avanzini et al. 2024] supports proving bounds on expected values of program states for first-order programs. Under the assumption that these programs terminate, a cost analysis can be encoded by instrumenting programs with a step counter. By integrating with a relational logic, eHL supports replacing a program with a simplified model if they compute the same costs.

Amortized reasoning about expected values has also been developed in several probabilistic extensions of Automatic Amortized Resource Analysis (AARA) [Hoffmann and Jost 2022; Hofmann and Jost 2003]. In the AARA approach, the type system is extended with *potential functions* that track the resources associated with an expression. Ngo et al. [2018] observed that this idea could be adapted to reasoning about expected resource consumption in randomized programs by taking weighted averages of potential functions when randomized choices are made, much as HT-RAND-EXP in Tachis takes a weighted average of credits. They develop a static analysis tool called Absynth that automatically derives expected resource bounds for first-order randomized programs. The soundness of Absynth is justified in terms of derivations in a Hoare-like logic with judgements of the form $\{\Gamma_1; Q_1\}\ e\ \{\Gamma_2; Q_2\}$, where the pre- and post-conditions are divided up into a logical predicate Γ_i, and a quantitative potential function Q_i. In contrast, Tachis represents resources as credit assertions, which can appear anywhere within specifications, allowing for stored credit assertions that depend on data structure invariants, such as in the hash table. Wang et al. [2020] subsequently developed pRaML, an AARA-based type system for expected costs of higher-order probabilistic programs. However, pRaML does not support mutable references, so examples combining higher-order functions and references like the meldable heaps in §5.6 are beyond its scope.

In work done concurrently with ours, Lohse and Garg [2024] developed *ExpIris*, a variant of Iris that supports establishing bounds on the expected cost of higher-order programs with mutable state. ExpIris adds additional parameters to the standard weakest precondition of Iris [Jung et al. 2018] that record an initial potential $p : \mathbb{R}$ and a final potential $\mathfrak{P} : Val \rightarrow \mathbb{R}$. The cost of a program is defined via manually inserted tick operations, where each tick(c) consumes c from the initial potential. Because ExpIris tracks potentials through weakest precondition parameters, there are some restrictions on the use of potentials that are similar to those in the Hoare logic of Ngo et al. [2018] described above. In particular, the final potential can only depend on the return value of a program but not on its final state. Tachis can handle such cases by breaking the tight coupling between the weakest precondition and costs through the use of the credit resource.[6] As an orthogonal consideration, ExpIris also supports reasoning about the total *work* of concurrent programs (but not the *span*, in the sense of complexity of parallel algorithms).

Analyses and logics based on Ranking Supermartingales (RSMs) have been developed for proving almost-sure termination, deriving high-probability bounds on random variables, bounding expected values and higher moments of running time, *e.g.*, its variance [Agrawal et al. 2018; Chatterjee et al. 2024; Fioriti and Hermanns 2015; Fu and Chatterjee 2019; Kura et al. 2019; McIver et al. 2018]. An RSM can be seen as a function assinging an expected termination time to every line in a program, depending on the state. These works focus on supporting automation for first-order programs; RSMs are inferred using template-based synthesis and proven using arithmetic solvers. To enable this, assignments and conditional guards are often required to be written in a particularly

---

[6]We prove the example discussed as "out of scope" in Lohse and Garg [2024, §7] in Appendix C.

well-behaved subset of expressions, *e.g.*, polynomials. It would be interesting to investigate the idea that a Tachis proof of a cost bound implicitly ensures the existence of a certain RSM.

Some other approaches provide new formalisms to reason about expected runtime of higher-order programs. Avanzini et al. [2021b] introduce a probabilistic extension of sized types. This is a rich type system, which includes refinement types as well as distributions types which can be thought of as probabilistic counterparts of sum types. Other type-based approaches to reason about almost-sure termination of probabilistic programs include intersection types [Dal Lago et al. 2021]. Another approach by Avanzini et al. [2021a] instead proposes a continuation-passing style translation of a program into a runtime transformer, *i.e.*, a translated program that computes the runtime of the original program. Then, one can use a standard higher-order program logic to reason about the translated program and obtain runtime bounds on the original program.

Separation logics for probabilistic programs. In addition to QSL and its variants, several other separation logics have been developed for reasoning about probabilistic programs. Starting with PSL [Barthe et al. 2020], several separation logics have been developed in which the notion of separation encoded in the separating conjunction corresponds to probabilistic independence [Bao et al. 2021, 2022; Li et al. 2023]. Others re-use a standard interpretation of separating conjunction, but encode a probabilistic property in the interpretation of Hoare triples or weakest preconditions. For example, Polaris [Tassarotti and Harper 2019] and Clutch [Gregersen et al. 2024b] are Iris-based logics for doing probabilistic relational reasoning using the technique of probabilistic couplings [Barthe et al. 2015; Lindvall 2002]. Caliper [Gregersen et al. 2024a] exploits probabilistic couplings to show almost-sure termination through termination-preserving refinement of a Markov chain model. Eris [Aguirre et al. 2024] is a variant of Iris that adapts the ideas of approximate Hoare Logic [Barthe et al. 2016] to enable proving that a specification holds with high probability. Eris uses assertions called *error credits*, where an error credit of value ε can be "spent" to exclude reasoning about a randomized outcome that occurs with probability ε. These error credits can be averaged in an expectation-preserving way with a rule similar to HT-RAND-EXP.

Separation logics for non-probabilistic cost analysis. The idea of using a separation logic resource to account for the runtime of a program is due to Atkey [2011]. Time credits were introduced to Iris by Mével et al. [2019] in a logic called Iris$^\$$. Besides time credits, which represent upper bounds on the time before a good event happens, they also introduce a dual notion of time receipt, which allows lower bounding the time before a bad event happens. In later work, Pottier et al. [2024] show how to encode Okasaki's debit-based reasoning [Okasaki 1999] about thunks in Iris$^\$$. When reasoning about non-probabilistic parts of a program, the time credits in Tachis have equivalent reasoning rules as cost credits in Iris$^\$$. Thus, the proofs of sophisticated examples developed in these previous works could also be carried out in Tachis.

However, it would not have been possible to extend the elegant techniques of Iris$^\$$ to support probabilistic reasoning: In Iris$^\$$ deterministic time credits are realized by (1) translating the program to an instrumented program that has a distinguished counter that decreases at every step, (2) making the program crash if this counter reaches 0, and (3) making the time credits track the value of this counter. At this point, the standard Iris adequacy theorem [Jung et al. 2018] that ensures crash-freedom also enforces that this counter can never reach 0. The program thus cannot take more steps than the initial amount of credits. But this method cannot be applied to establish *expected* cost bounds because one needs to increase the amount of credits in some execution branches and decrease in others. This would require reasoning about the expected value of a program location, and this is not supported in any existing logic for higher-order randomized stateful programs.

We can summarize our solution as follows. First, we define a new resource algebra representing expected cost as well as a new bespoke weakest precondition that tracks this resource algebra and connects it to the probabilistic operational semantics of RandML. This weakest precondition supports distributing cost credits across execution branches via the expected cost modality. We then prove that this weakest precondition interacts with step indexing and other Iris features such as invariants. Support for step indexing and handling this interaction is necessary to be able to recover all of the expressive features of Iris, such as reasoning about higher-order state. Next, we prove all of the "standard" program logic rules which one may expect from, *e.g.*, Jung et al. [2018], as well as the novel cost-averaging rules. Finally, we prove an adequacy theorem that connects a Hoare triple in Tachis to the expected cost of a program. The proof of the adequacy theorem intricate since it requires approximating the expected cost inductively, and connecting to the expected cost of full-program execution via a limiting argument.

7.2 Future Work

We see several interesting research directions for extending Tachis.

Continuous distributions. Adding support for sampling from continuous distributions to RandML will bring significant challenges in terms of the mechanization of the measure theory needed to construct the operational semantics of our language and its associated metatheory. However, these challenges are orthogonal to cost analysis, and we believe a similar rule as the expectation-preserving composition for cost credits should hold in the continuous setting.

Cost lower bounds. Some program logics for cost analysis support reasoning about lower bounds on costs. In the non-probabilistic setting, this can be achieved through *time receipts* [Mével et al. 2019], but for reasons analogous to the discussion of time credits in §7.1, the technique used by Mével et al. of deriving cost bounds via a program translation cannot directly be transposed into the probabilistic setting. We expect that such an extension would require adding native support for time receipts at the level of the logic and hence the model of Tachis. However, extending Tachis with support for lower bounds would be a challenge since, in general, proving lower bounds via an inductive argument is not sound and one has to impose additional conditions [Hark et al. 2019]. Alternatively, a *linear* separation logic could allow dualizing the credit-style representation of upper bounds as a *prerequisite* for taking steps into an *obligation* to take steps.

8 Conclusion

We presented Tachis, the first program logic for proving upper bounds of the expected cost of higher-order probabilistic programs with local state. We developed the notion of probabilistic cost credits, derived from time credits that were originally used to prove the runtime cost of deterministic programs, which enables us to reason about the cost of *probabilistic* programs. Our definition of cost credits can depend flexibly on user-defined cost models, enabling reasoning about a much richer class of costs specific for different applications. We demonstrated the strength of Tachis on various examples, including ones whose runtime bounds are non-trivial or are amortized, which to our knowledge are outside the scope of prior techniques.

Acknowledgments

those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Data-Availability Statement

A full Coq formalization of Tachis including the case studies is available [Haselwarter et al. 2024].

References

Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 34:1–34:32. https://doi.org/10.1145/3158122

Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. 2024. Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 246 (aug 2024), 33 pages. https://doi.org/10.1145/3674635

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021a. On Continuation-Passing Transformations and Expected Cost Analysis. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 87:1–87:30. https://doi.org/10.1145/3473592

Martin Avanzini, Gilles Barthe, Benjamin Grégoire, Georg Moser, and Gabriele Vanoni. 2024. Hopping Proofs of Expectation-Based Properties: Applications to Skiplists and Security Proofs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 122 (apr 2024), 26 pages. https://doi.org/10.1145/3649839

Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2021b. Type-based complexity analysis of probabilistic functional programs. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science* (Vancouver, Canada) *(LICS '19)*. IEEE Press, Article 41, 13 pages.

Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 172 (nov 2020), 30 pages. https://doi.org/10.1145/3428240

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A Bunched Logic for Conditional Independence. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470712

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A separation logic for negative dependence. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498719

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. 387–401. https://doi.org/10.1007/978-3-662-48899-7_27

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs, Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107:1–107:15. https://doi.org/10.4230/LIPICS.ICALP.2016.107

Gilles Barthe, Justin Hsu, and Kevin Liao. 2020. A probabilistic separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 55:1–55:30. https://doi.org/10.1145/3371123

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. https://doi.org/10.1145/3290347

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986. https://doi.org/10.1145/3571260

Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '93)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/168588.168596

Jon Louis Bentley. 1986. *Programming pearls*. Addison-Wesley.

Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Math. Comput. Sci.* 9, 1 (2015), 41–62. https://doi.org/10.1007/S11786-014-0181-1

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Đorđe Žikelić. 2024. Quantitative Bounds on Resource Usage of Probabilistic Programs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 107 (apr 2024), 30 pages. https://doi.org/10.1145/3649824

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

T.M. Cover and J.A. Thomas. 2006. *Elements of Information Theory*. Wiley.

Ugo Dal Lago, Claudia Faggian, and Simona Ronchi Della Rocca. 2021. Intersection types and (positive) almost-sure termination. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 1–32. https://doi.org/10.1145/3434313

Richard Durstenfeld. 1964. Algorithm 235: Random permutation. *Commun. ACM* 7, 7 (jul 1964), 420. https://doi.org/10.1145/364520.364540

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 489–501. https://doi.org/10.1145/2676726.2677001

Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

Anna Gambin and Adam Malinowski. 1998. Randomized Meldable Priority Queues. In *SOFSEM '98: Theory and Practice of Informatics, 25th Conference on Current Trends in Theory and Practice of Informatics, Jasná, Slovakia, November 21-27, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1521)*, Branislav Rovan (Ed.). Springer, 344–349. https://doi.org/10.1007/3-540-49477-4_26

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024a. Almost-Sure Termination by Guarded Refinement. *Proc. ACM Program. Lang.* 8, ICFP, Article 243 (aug 2024), 31 pages. https://doi.org/10.1145/3674632

Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024b. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 26 (2024). https://doi.org/10.1145/3632868

Te Sun Han and Mamoru Hoshi. 1997. Interval algorithm for random number generation. *IEEE Trans. Inf. Theory* 43, 2 (1997), 599–611. https://doi.org/10.1109/18.556116

Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2019. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL, Article 37 (dec 2019), 28 pages. https://doi.org/10.1145/3371105

Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. 2024. *Tachis: Higher-Order Separation Logic with Credits for Expected Costs - Coq Artifact*. https://doi.org/10.5281/zenodo.12659527

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/604131.604148

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. *SIGPLAN Not.* 51, 9 (sep 2016), 256–269. https://doi.org/10.1145/3022670.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *ACM-SIGACT Symposium on Principles of Programming Languages*.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer, Berlin, Heidelberg, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.* 52, 1 (jan 2017), 205–217. https://doi.org/10.1145/3093333.3009855

Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Momentshsu. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Vol. 11428. Springer International Publishing, Cham, 135–153. https://doi.org/10.1007/978-3-030-17465-1_8

Daniel Le Métayer. 1988. ACE: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.* 10, 2 (apr 1988), 248–266. https://doi.org/10.1145/42190.42347

Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13372)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 70–91. https://doi.org/10.1007/978-3-031-13188-2_4

John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proc. ACM Program. Lang.* 7, PLDI (2023), 148–171. https://doi.org/10.1145/3591226

T. Lindvall. 2002. *Lectures on the Coupling Method.* Dover Publications, Incorporated.

Janine Lohse and Deepak Garg. 2024. An Iris for Expected Cost Analysis. arXiv:2406.00884 [cs.PL]

Jérémie O. Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *CoRR* abs/1304.1916 (2013). arXiv:1304.1916 http://arxiv.org/abs/1304.1916

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer. https://doi.org/10.1007/B138392

Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A new proof rule for almost-sure termination. *Proc. ACM Program. Lang.* 2, POPL (2018), 33:1–33:28. https://doi.org/10.1145/3158121

Andrea Mennucci. 2010. Bit recycling for scaling random number generators. *CoRR* abs/1012.4290 (2010). arXiv:1012.4290 http://arxiv.org/abs/1012.4290

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018.* ACM Press, Philadelphia, PA, USA, 496–512. https://doi.org/10.1145/3192366.3192394

Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 1482–1508. https://doi.org/10.1145/3632892

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings.* 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318. https://doi.org/10.1137/0606031

Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 64:1–64:30. https://doi.org/10.1145/3290377

H. Toutenburg. 1971. Fisher, R. A., and F. Yates: Statistical Tables for Biological, Agricultural and Medical Research. 6th Ed. Oliver & Boyd, Edinburgh and London 1963. X, 146 P. Preis 42 s net. *Biometrische Zeitschrift* 13, 4 (1971), 285–285. https://doi.org/10.1002/bimj.19710130413 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/bimj.19710130413

Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 110:1–110:31. https://doi.org/10.1145/3408992

A Full Definition of the Weakest Precondition

The weakest precondition that Tachis is based on is defined by the following guarded recursive fixed point.

$$\text{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \triangleq (e_1 \in \mathit{Val} \wedge \models_{\mathcal{E}} \Phi(e_1))$$
$$\vee\, (e_1 \notin \mathit{Val} \wedge \forall \sigma_1, x_1.\, S(\sigma_1) * \$_{\bullet}(x_1) \ast \!\!-\!\! {}_{\mathcal{E}}\!\!\models_{\emptyset}$$
$$\text{ECM}(e_1, \sigma_1, x_1,\, (\lambda\, e_2, \sigma_2, x_2 \, . \, \triangleright\, {}_{\emptyset}\!\!\models_{\mathcal{E}} (S(\sigma_2) * \$_{\bullet}(x_2) * \text{wp}_{\mathcal{E}}\, e_2\, \{\Phi\}))))$$

It differs from the presentation in §6.1 in the use of the Iris update modality $\models$, which we earlier omitted for the sake of readability. The purpose of the update modality is to allow the user of the logic to perform updates to the resources they own; see, *e.g.*, Jung et al. [2018] for details.

B Hash Map

In this example, we establish bounds on inserting and looking-up elements in a hash map. We use the $cost_{\text{tick}}$ model, where we incur $\$(1)$ whenever we perform an access to memory by manually adding a tick 1 operation after each dereference.

Recall from §5.4 that a hash map supports two operations: insertion and lookup. The hash map contains an array, where each array index contains a pointer to a linked list (called a bucket), which stores a list of elements. To insert an element, the hash map uses a hash function to hash the element, and append it to the bucket with the array index of the hash. On the other hand, to perform a lookup on an element, the hash map hashes the element, and traverse the bucket with the array index of the hash to determine whether it is present in the bucket.

We first implement a model of the idealized hash function in Figure 11 under the uniform hash assumption [Bellare and Rogaway 1993], which assumes that the hash function hf from a set of keys K to values V behaves as if, for each key k, the hash $hf(k)$ is randomly sampled from a uniform distribution over V, independently of all other keys. We implement this model by first initializing an empty mutable map hf. When we want to hash a key k, we first check the map to determine whether there is an element under that key. If so, we return the hash value stored in $hf(k)$. Otherwise, we sample a value uniformly from $V = \{0, \ldots, n\}$, store the value in hf with key k, and return it.

$$
\begin{aligned}
\text{compute\_hash} \triangleq \lambda\, hf\, k.\ &\text{match get } hf\, k \text{ with} \\
&\quad \text{Some}(v) \Rightarrow v \\
&\quad |\ \text{None} \quad \Rightarrow \text{let } v = \text{rand } n \text{ in} \\
&\qquad\qquad\qquad\quad \text{set } hf\, k\, v; \\
&\qquad\qquad\qquad\quad v \\
&\text{end}
\end{aligned}
$$

Fig. 11. An idealized hash function implementation.

Note that in this example, although the implementation of maps involve various location accesses, we do not include tick operations in our hash implementation. This is because our implementation only serves to simulate the effects of a realistic hash function, which in real life, a hash function usually incurs constant cost for each hash operation.

We then implement the hash map as a client of the hash function. We represent our hash map h as a tuple (l, hf). The former is an array whose elements are buckets of the hash map while the latter is the hash function. We represent the list of elements which we store in the bucket with a linked list.

insert ≜ λ h v.   lookup ≜ λ h v.
  let (l, hf) = h in   let (l, hf) = h in
  let idx = compute_hash hf v in   let idx = compute_hash hf v in
  let b = ! l[idx] in   let b = ! l[idx] in
  tick 1;   tick 1;
  (l +ₗ idx) ← LinkedList.insert b v   LinkedList.lookup b v

Fig. 12. A hash map implementation.

The hash map supports two main operations shown in Figure 12: insert which inserts an element into the hash table, and lookup which determines whether an element is stored in it. The function insert hashes an element with the hash function hf to get the index of the bucket it should go to. Then we access that bucket with a dereference, incurring one tick. Finally we insert the element into the end of the linked list representing the bucket. Here, when we traverse to the end of the linked list, we also perform a check to determine whether the element already exists in the bucket, in which case, we do not add it at the end of the linked list to avoid duplication. The function lookup works similarly by first hashing the element with hf to get the index of the bucket, accessing the bucket with a load from memory and hence incurring one tick cost, and finally performing a lookup by traversing the linked list. Since both linked list functions LinkedList.insert and LinkedList.lookup traverses the entire linked list, and memory dereferencing costs 1 in our model, the functions incur $\$(l)$ credits for each invocation where l is the length of the linked list.

Let us consider the costs of insert and lookup in the case where an element has never been hashed by a hash function hf before. For insert, when we first hash the element, since it has not been hashed by hf before, we randomly pick an index idx from $V = \{0, \ldots, n\}$. Let the cost of insert in the case that the element is hashed to idx by hf be $C(idx)$. By stepping through the program, we can show that $C(idx) = 1 + \text{hmLength}(h, idx)$, where $\text{hmLength}(h, idx)$ is notation for the length of the linked list representing the idx-th bucket. The first 1 cost comes from accessing the idx-th bucket, and hmLength comes from inserting the element at the end of the linked list. Hence the expected cost of insert is $\mathbb{E}_{\mathbb{U}(0,n)}[C]$, where $\mathbb{U}(0, n)$ is the discrete uniform distribution from 0 to n. By linearity of expectations, we have $\mathbb{E}_{\mathbb{U}(0,n)}[C] = \mathbb{E}_{\mathbb{U}(0,n)}[1] + \mathbb{E}_{\mathbb{U}(0,n)}[\text{hmLength}(h, idx)] = 1 + \frac{\text{size}(h)}{n+1}$ where size(h) is shorthand for the total number of elements stored in the hash map (also the sum of lengths of all the linked list buckets). The time complexity of lookup is exactly the same, as instead of adding an element to the end of a linked list in insert, we still traverse the linked list comparing our element with each value stored by the linked list.

To prove this claim, we use Tachis to prove the following two specifications.

$$\left\{ v \notin \text{dom}(m_1) * \text{isHashMap}(h, m_1, m_2) * \$\left(1 + \frac{\text{size}(h)}{n+1}\right) \right\}$$
$$\quad \text{insert } h\, v$$
$$\left\{ \_.\, \exists i.\, \text{isHashMap}(h, m_1[v \leftarrow i], m_2[i \leftarrow m_2[i] \mathbin{+\!\!+} [v]]) \right\}$$
$$\left\{ v \notin \text{dom}(m_1) * \text{isHashMap}(h, m_1, m_2) * \$\left(1 + \frac{\text{size}(h)}{n+1}\right) \right\}$$
$$\quad \text{lookup } h\, v$$
$$\left\{ b.\, b = \text{false} * \exists i.\, \text{isHashMap}(h, m_1[v \leftarrow i], m_2) \right\}$$

Intuitively, the predicate isHashMap h m_1 m_2 asserts that the hash map h is valid according the two mathematical maps m_1 and m_2. For the rest of the subsection, it suffices to understand that the abstract predicate contains two main components, that the abstract map m_1 tracks the concrete map implementing the hash function, which we represent with the predicate isHashFunction, and the abstract map m_2 tracks the linked lists of each bucket of the concrete hash map. Consequently, the proposition $v \notin \mathrm{dom}(m_1)$ encodes the idea that the hash function has not hashed the element v before.

The proof of the two above specifications are fairly standard, except for the part where we hash the element v. Note that we have to split our cost credits $1 + \frac{\mathrm{size}(h)}{n+1}$ somewhere but at first glance there does not seem to be any rand expressions for us to apply the expectation rule. The reason is that the rand expression is encapsulated in the hash function and is not exposed for the client. This subtle detail means that we have to provide a strong specification of the hash function that allows us to split our credits, just as we could for a normal rand expression.

As an example, the following specification for the hash function is not strong enough for verifying the hash table albeit being valid:

$$\{v \notin \mathrm{dom}(m_1) * \mathsf{isHashFunction}\ hf\ m_1\}\ hf\ v\ \{idx.\, \mathsf{isHashFunction}\ hf\ m_1[v \leftarrow idx]\}$$

The reason is that the specification does not allow us to split credits in any meaningful way. To prove the specification of the hash map functions, one have to use the following strictly stronger specification for the hash function which allows the client to divide the credits depending on the hash value, similarly to the expectation rule for rand:

$$\left\{v \notin \mathrm{dom}(m_1) * \mathsf{isHashFunction}\ hf\ m_1 * \Sigma_{i=0}^{n}(\frac{x_2(i)}{n+1}) = x_1 * \forall i, 0 \le x_2(i) * \$(x_1)\right\}$$
$$hf\ v$$
$$\left\{idx.\, \mathsf{isHashFunction}\ hf\ m_1[v \leftarrow idx]\ *\ \$(x_2(idx))\right\}$$

One limitation of the specification of insert is that required credits for each insertion is propositional to the size of the hash table, which leads to worse modularity as a client of the hash table needs to know the exact size of the hash table. We can alleviate this problem by providing an amortized specification of insert where the cost credits required for each insertion is constant. The tradeoff is that one has to fix the total maximum size of the hash table *a priori*. If the total maximum size of the hash table is MAX, the total number of cost is hence $\$\left(\Sigma_{i=0}^{\mathsf{MAX}-1}(1 + \frac{\mathrm{size}(h)}{n+1})\right) = \mathsf{MAX} + \frac{(\mathsf{MAX}-1)\cdot\mathsf{MAX}}{2(n+1)}$. If we average out the credits, the average cost is simply $\$\left(\frac{\mathsf{MAX}}{\mathsf{MAX}} + \frac{(\mathsf{MAX}-1)\cdot\mathsf{MAX}}{2(n+1)\cdot\mathsf{MAX}}\right) = \$\left(1 + \frac{\mathsf{MAX}-1}{2(n+1)}\right)$, which we simply denote as $\$(x)$. Therefore we can write the following amortized spec:

$$\{v \notin \mathrm{dom}(m_1) * \mathrm{size}(h) < \mathsf{MAX} * \mathsf{isAmortizedHashMap}(h, m_1, m_2) * \$(x)\}$$
$$\mathsf{insert}\ h\ v$$
$$\{\_.\, \exists i.\, \mathsf{isAmortizedHashMap}(h, m_1[v \leftarrow i], m_2[\leftarrow m_2[i] + [v]])\}$$

Here we make three main changes to the original specification of insert. First we require that the total size of the hashmap is smaller than MAX. We also fix the credit in the precondition to a constant. Lastly, we change the abstract predicate isHashMap to isAmortizedHashMap. Intuitively, isAmortizedHashMap not only contains the isHashMap abstract predicate but also the extra time credits one pays in excess for the first half of the insertions. For the last half of the insertions, the amortized cost is insufficient to pay for the expensive operation so we take just enough from this reserve to pay for the difference. To highlight the modularity aspect of Tachis, we actually derive this amortized specification from the non-amortized one of the insert function.

We note that we are unable to provide a meaningful amortized specification for the lookup function as each invocation of the function does not change the size of the hash table, and hence we are unable to fix the number of times we can call lookup.

We also note that in Tachis, we are unable to provide a cost that is proportional to the size of the hash table for the case where a key has been hashed before. In textbook proofs, one can argue that the cost is still proportional to the size of the hash table even if the key has been hashed before (or inserted into the hash table) assuming that the element is chosen uniformly across all elements stored in the hash table. However, our logic cannot describe this assumption and we leave this extension to our logic as future work.

C State-dependent Expected Cost

We prove that the following program, suggested by [Lohse and Garg 2024, §7], has expected cost $\frac{1}{2}$.

$$\text{toss } \mathsf{l} \triangleq \text{if } \text{rand } 1 = 1 \text{ then } ()$$
$$\text{else } \mathsf{l} \leftarrow \, ! \, \mathsf{l} + 1$$

$$\mathsf{T}_{\frac{1}{2}} \triangleq \text{let } \mathsf{l} = \text{ref } 0 \text{ in toss } \mathsf{l} \, ; \, \text{tick}(! \, \mathsf{l})$$

According to *loc. cit.*, the analysis of the expected cost of $\mathsf{T}_{\frac{1}{2}}$ is interesting in that the cost depends on the value stored in the location l.

We can derive the cost bound in Tachis by choosing the $cost_{\text{tick}}$ model. To begin, we allocate a reference ℓ and perform an expectation-preserving splitting of the cost credits $\$\left(\frac{1}{2}\right)$. We give $\$(f(1)) = \$(0)$ to the case where rand 1 returns 1 and ℓ is not incremented, and reserve $\$(f(0)) = \$(1)$ for the else branch.

$$\dfrac{\dfrac{\dfrac{\dfrac{f(n) \triangleq \text{if } n = 1 \text{ then } 0 \text{ else } 1 \qquad 0 + \mathbb{E}[f] \leq \frac{1}{2}}{\{\$\left(\frac{1}{2}\right)\} \text{ rand } 1 \, \{n \, . \, \$(f(n))\}} \text{ \scriptsize RAND-EXP}}{\{\ell \mapsto 0 * \$\left(\frac{1}{2}\right)\} \text{ rand } 1 \, \{n \, . \, \ell \mapsto 0 * \$(f(n))\}} \text{ \scriptsize FRAME} \qquad \dfrac{(1)}{\forall n. \, \{\ell \mapsto 0 * \$(f(n))\} \text{ toss } \ell \, ; \, \text{tick}(! \, \ell) \, \{\text{True}\}}}{\{\ell \mapsto 0 * \$\left(\frac{1}{2}\right)\} \text{ toss } \ell \, ; \, \text{tick}(! \, \ell) \, \{\text{True}\}} \text{ \scriptsize BIND}}{\{\$\left(\frac{1}{2}\right)\} \, \mathsf{T}_{\frac{1}{2}} \, \{\text{True}\}}$$

The result now follows from a straightforward case analysis and the definition of $cost_{\text{tick}}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\$(0)\} \, () \, ; \, \text{tick}(0) \, \{\text{True}\}} \text{ \scriptsize TICK}}{\{\ell \mapsto 0 * \$(f(1))\} \, () \, ; \, \text{tick}(! \, \ell) \, \{\text{True}\}} \text{ \scriptsize LOAD,FRAME} \qquad \dfrac{\dfrac{\dfrac{}{\{\$(f(1))\} \, \text{tick}(1) \, \{\text{True}\}} \text{ \scriptsize TICK}}{\{\ell \mapsto 1 * \$(f(1))\} \, \text{tick}(! \, \ell) \, \{\text{True}\}} \text{ \scriptsize LOAD,FRAME}}{\{\ell \mapsto 0 * \$(f(0))\} \, \ell \leftarrow \, ! \, \ell + 1 \, ; \, \text{tick}(! \, \ell) \, \{\text{True}\}} \text{ \scriptsize LOAD,STORE}}{\forall n. \, \{\ell \mapsto 0 * \$(f(n))\} \, (\text{if } n = 1 \text{ then } () \text{ else } \ell \leftarrow \, ! \, \ell + 1) \, ; \, \text{tick}(! \, \ell) \, \{\text{True}\}} \text{ \scriptsize CASE } n = 1 \, ?}{(1)}$$

The expected cost bound then follows directly by adequacy.