Raising Expectations: Automating Expected Cost Analysis with Types

DI WANG, Carnegie Mellon University, USA
DAVID M. KAHN, Carnegie Mellon University, USA
JAN HOFFMANN, Carnegie Mellon University, USA

This article presents a type-based analysis for deriving upper bounds on the expected execution cost of probabilistic programs. The analysis is naturally compositional, parametric in the cost model, and supports higher-order functions and inductive data types. The derived bounds are multivariate polynomials that are functions of data structures. Bound inference is enabled by local type rules that reduce type inference to linear constraint solving. The type system is based on the potential method of amortized analysis and extends automatic amortized resource analysis (AARA) for deterministic programs. A main innovation is that bounds can contain symbolic probabilities, which may appear in data structures and function arguments. Another contribution is a novel soundness proof that establishes the correctness of the derived bounds with respect to a distribution-based operational cost semantics that also includes nontrivial diverging behavior. For cost models like time, derived bounds imply termination with probability one. To highlight the novel ideas, the presentation focuses on linear potential and a core language. However, the analysis is implemented as an extension of Resource Aware ML and supports polynomial bounds and user defined data structures. The effectiveness of the technique is evaluated by analyzing the sample complexity of discrete distributions and with a novel average-case estimation for deterministic programs that combines expected cost analysis with statistical methods.

CCS Concepts: • **Theory of computation** → **Probabilistic computation**; *Automated reasoning*; **Type theory**; *Operational semantics.*

Additional Key Words and Phrases: analysis of probabilistic programs, expected execution cost, resource-aware type system

ACM Reference Format:
Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types. *Proc. ACM Program. Lang.* 4, ICFP, Article 110 (August 2020), 31 pages. https://doi.org/10.1145/3408992

1 INTRODUCTION

Probabilistic programming [Kozen 1981; McIver and Morgan 2005] is an effective tool for customizing probabilistic inference [Carpenter et al. 2017; Goodman and Stuhlmüller 2014; Mansinghka et al. 2018] as well as for modeling and analyzing randomized algorithms [Tassarotti and Harper 2019], cryptographic protocols [Barthe et al. 2009], and privacy mechanisms [Barthe et al. 2012].

Authors' addresses: Di Wang, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, USA, diw3@andrew.cmu.edu; David M. Kahn, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, USA, davidkah@andrew.cmu.edu; Jan Hoffmann, Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, USA, jhoffmann@cmu.edu.

2475-1421/2020/8-ART110
https://doi.org/10.1145/3408992

In this paper, we study probabilistic programs as models of the execution cost (or resource use) of programs. Execution cost can be defined by a cost semantics or a programmer-defined metric. For such a cost model, a probabilistic program defines a distribution of cost that depends on the distribution of the inputs as well as the probabilistic choices that are made in the code.

The problem of statically analyzing the cost distribution of probabilistic programs has attracted growing attention in recent years. Kaminski et al. [Kaminski et al. 2016; Olmedo et al. 2016] have built on the work of Kozen [Kozen 1981], studying weakest-precondition calculi for deriving upper bounds on the expected worst-case cost of imperative programs, as well as reasoning about lower bounds [Hark et al. 2020]. It has been shown that this calculus can be specialized to automatically infer constant bounds on the sampling cost of non-recursive Bayesian networks [Batz et al. 2018] and polynomial bounds on the worst-case expected cost of arithmetic programs [Chatterjee et al. 2016a,b; Ngo et al. 2018]. The key innovation that enables the inference of symbolic bounds is a template-based approach that reduces bound inference to efficient *linear-program* (LP) solving, a reduction which has been previously applied non-probabilistic programs [Carbonneaux et al. 2017, 2015]. This technique has been extended to best-case bounds and non-monotone cost [Wang et al. 2019] as well as to incorporate higher-moment reasoning for deriving tail bounds using linear [Wang et al. 2020a] and non-linear [Kura et al. 2019] constraint solving.

The only existing technique for analyzing the expected cost of probabilistic (higher-order) functional programs, is the recent work of Avanzini et al. [Avanzini et al. 2019]. It applies an affine refinement type system, called ℓRPCF, to derive bounds on the expected worst-case cost for an affine version of PCF [Plotkin 1977]. ℓRPCF can be seen as a probabilistic version of dℓPCF [Dal Lago and Gaboardi 2011]. While the refinement types of ℓRPCF are expressive and flexible, a disadvantage is that the complexity of the corresponding refinement constraints hampers type inference. It seems unclear if type checking ℓRPCF is decidable.

This article presents the first automatic analysis of worst-case bounds on the expected cost of probabilistic functional programs. It is based on *automatic amortized resource analysis* (AARA) [Hoffmann et al. 2011; Hofmann and Jost 2003], a type system for inferring worst-case bounds. The expressivity of AARA's type-based approach for probabilistic programs goes beyond existing techniques for imperative integer programs in the following ways:

(1) The analysis infers expected cost bounds for higher-order programs.
(2) Bounds can be functions of the sizes of values of (potentially nested) inductive types
(3) Bounds can be functions of symbolic probabilities.

In addition, AARA for probabilistic programs preserves many advantageous features of classical AARA for deterministic programs, which include

- efficient type checking (linear in the size of the type derivation),
- reduction of type inference for *polynomial bounds* to *linear programming*,
- use of the potential method to amortize operations with varying expected cost, and
- natural compositionality, as types summarize the cost behavior of functions.

Nonetheless, while AARA for deterministic programs naturally derives bounds on the high-water mark of non-monotone resources that can become available during evaluation (like memory), this is not the case for AARA for probabilistic programs. Reasoning about high-watermark resource usage of probabilistic programs is in fact an open problem even for manual reasoning systems for first-order languages. This problem is out of the scope of this article and we limit the development to monotone resources like time. The technical difficulties with non-monotone resources are discussed in more detail in §3.

To focus on the novel ideas, we present the analysis for a simple probabilistic functional language with probabilistic branching and lists (§3) with linear potential functions (§4). However, the results carry over to multivariate polynomial potential functions and user-defined inductive data structures. We implemented the analysis as an extension of Resource Aware ML (RaML) [Hoffmann et al. 2017] that we call pRaML (§6).

The main technical innovations are the introduction of a type rule for probabilistic branching, and a new type for symbolic probabilities (§2 and §4). While these new features are fairly intuitive, proving their soundness with respect to a cost semantics is not. The existing proof method for deterministic AARA does not directly generalize to the probabilistic setting because of the complexities introduced by a probabilistic cost semantics. To address the challenges of the probabilistic setting, we present a novel soundness proof with respect to a probabilistic operational cost semantics based on Borgström et al.'s trace-based and step-indexed-distribution-based semantics [Borgström et al. 2016] (§5). The details are discussed in §3.

We evaluate the effectiveness of pRaML by analyzing textbook examples (§6) and by exploring novel problem domains (§7). The first domain (§7.1) is the implementation and analysis of discrete probability distributions. Specifically, we use pRaML to analyze the *sample complexity* of the distributions, i.e., on average, how many steps a program needs to produce a sample from the target distribution. Low sample complexity has recently become an important criterion for efficient sampler implementations, as many probabilistic inference methods require billions of random samples [Djuric 2019]. We also verify some more complex fractional bounds in pRaML using a scaled model. The second domain (§7.2) is the estimation of average-case cost of functional programs on a specific input distribution as a three step process. First, we gather statistics on the branching behavior of conditional branches by evaluating the program on small inputs that are representative for the input distribution. Second, the conditionals are replaced with probabilistic branches that mirror the observed branching behavior on the small inputs. Third, the resulting program is analyzed with pRaML to determine a symbolic bound on the expected cost of the resulting probabilistic program for all input sizes.

In summary, we make the following *contributions*:

(1) Design of a novel type-based AARA for probabilistic programs
(2) Type soundness proof with respect to a probabilistic operational cost semantics
(3) Implementation as an extension of RaML
(4) Application of RaML to automatically analyze sample complexity
(5) Automatic average-case analysis that combines the use of RaML with empirical statistics

2 TOPIC OVERVIEW

AARA. The type system of *automatic amortized resource analysis* (AARA) is a pre-existing framework for inferring cost bounds for deterministic functional programs [Hoffmann et al. 2017; Hofmann and Jost 2003; Jost et al. 2010]. It imbues its types with potential energy so as to perform the *physicist's method* (or *potential method*) of amortized analysis [Tarjan 1985]. When performing type inference, the system generates linear constraints on this potential that, when solved, provide the coefficients of polynomials or other functions. These functions express concrete (non-asymptotic) bounds on worst- or best-case [Ngo et al. 2017] execution costs, parameterized by input size.

In more detail, the potential method works as follows. We say that $\Phi : \text{State} \to \mathbb{Q}_{\geq 0}$ is a valid potential function if, for all states $S \in \text{State}$ and operations $o : S \to S$, the following holds.

$$\Phi(S) \geq 0 \qquad \text{and} \qquad \Phi(S) \geq \mathit{cost}(S, o(S)) + \Phi(o(S)).$$

The second inequality states that the potential of the current state is sufficient to pay for the cost of the transition from S to $o(S)$ and potential of the next state. It then follows that the potential of the initial state establishes an *upper* bound on the *worst-case* cost of a sequence of operations.

```
let rec exists pred lst =
  match lst with
  | [] → false
  | hd::tl →
    let _ = tick 1 in
    if pred hd
    then true
    else exists pred tl
```

(a)

```
let rec bernoulli lst =
  match lst with
  | [] → false
  | hd::tl →
    let _ = tick 1 in
    match flip 0.5 with
    | H → true
    | T → bernoulli tl
```

(b)

```
let rec rdwalk lst =
  match lst with
  | [] → ()
  | p::ps →
    let _ = tick 1 in
    match flip p with
    | H → rdwalk (0.2::0.4::ps)
    | T → rdwalk ps
```

(c)

Fig. 1. Implementations of probabilistic programs in pRaML.

The AARA type system is designed to automatically assign such potential functions to functional programs, where we view evaluation steps as operations on machine states of an abstract machine. Automation is enabled by fixing the format potential functions to linear combinations of base functions, and then incorporating them into the types of values. Consider for example the function *exists* from the OCaml List module in Fig. 9a. We model its cost behaviour using explicit *tick(q)* expressions that consume $q \geq 0 \in \mathbb{Q}$ when evaluated. The function *exists pred lst* has a cost of 1 for in every recursive call, and therefore the worst-case cost is equal to the length of *lst* in addition to the cost of the calls to the function *pred*.

To automatically derive this bound in linear AARA we assign the following type template where q_0, q_1, q, p, r and r' are yet unknown non-negative coefficients.

$$exists : \langle\langle\tau, r\rangle \to \langle\text{bool}, r'\rangle, q_0\rangle \to \langle L^p(\tau), q_1\rangle \to \langle\text{bool}, q\rangle$$

A valid instantiation of the potential annotation would for instance be the following type.

$$exists : \langle\langle\tau, 0\rangle \to \langle\text{bool}, 0\rangle, 0\rangle \to \langle L^1(\tau), 0\rangle \to \langle\text{bool}, 0\rangle$$

If we ignore the potential annotations in τ and the cost of evaluating the function *pred*, then this type expresses that the cost of evaluating *exists pred lst* is $1 \cdot |lst|$, as marked by requiring a list argument with 1 unit of potential per element. Another valid typing is

$$exists : \langle\langle\tau, 2\rangle \to \langle\text{bool}, 0\rangle, 0\rangle \to \langle L^3(\tau), 0\rangle \to \langle\text{bool}, 0\rangle\ .$$

It now expresses that the cost of evaluating *exists pred lst* is $3 \cdot |lst|$ if the cost of evaluating *pred* is raised to 2. The *pred* function here is typed to take 2 units of potential to run, but is balanced by each element of the list argument being paired with 3 units of potential, 2 more than previously.

In general, type inference constrains this type's annotation variables with $p \geq r + 1$ and $q_1 \geq q$, and leaves the other annotations unconstrained. This aids in the compositionality of the approach, as the specific constants chosen can be adapted to the arguments, including arguments that are themselves functions like *pred* here.

To exemplify such compositionality, consider some function *f* that merely iterates over a list, consumes 1 resource every iteration, and then returns the list. It can be typed $\langle L^p(\tau), 0\rangle \to \langle L^q(\tau), 0\rangle$ where $p \geq q + 1$. If we chain its application to some list *lst* as *f (f lst)*, then we might instantiate the type of the inner application with $p = 2, q = 1$, and the outer with $p = 1, q = 0$, composing the costs naturally. In this case, we would also type *lst* as $L^2(\tau)$.

Of course, AARA cannot do the impossible of successfully analyzing all programs. AARA uses structural reasoning methods that cannot pick up on semantical properties that the program may depend on, like Peano arithmetic. Further, not all resource usage can be accurately expressed in a given class of resource functions. For instance, polynomials will over-approximate logarithms, and simply cannot express exponentials. The resource functions we present in this paper are linear, but we make use of polynomial resource functions in our implementation.

Probabilistic programming. In this paper, we extend AARA to deriving bounds on the expected cost of probabilistic programs. In contrast to a deterministic program, a probabilistic program may not always evaluate to the same value (if any), but rather to a distribution over values and divergence. Similarly, the evaluation cost of a probabilistic program is given by a distribution.

Consider for example the function *bernoulli* in Fig. 1b. It is similar to the function *exists*, but the conditional is replaced with the probabilistic construct *match flip 0.5*. Intuitively, this construct means that we flip a coin and evaluate the heads or tails branch based on the outcome. In probabilistic programming, we assume that such flips are truly random (as opposed to an implementation that may rely on a pseudorandom number generator). As a result, function *bernoulli* describes a Bernoulli process across the elements of an input list. It terminates with probability 1 and has the same linear worst-case cost as *exists*, namely $1 \cdot |lst|$. However, the expected cost of *bernoulli* is only 1.

For an example with a more interesting expected cost, consider the function *rdwalk* in Fig. 1c. Its argument is a list of probabilities that are used, one after another, to determine the odds in a probabilistic branch that either pops the head off the list (in the tails case) or adds two new probabilities to the list (in the heads case). The random walk consumes 1 *tick* in each iteration and terminates if the argument list is empty. One can show that the function *rdwalk* terminates with probability 1 and the expected cost is a function of the argument $[p_1, \ldots, p_n]$ as

$$n + \sum_{1 \le i \le n} 5p_i \ .$$

This is an example of a program with non-terminating execution that may nonetheless have expected costs that can be bounded. If only finite cost is accrued on non-terminating execution, nontermination may even occur with positive probability and still yield a finite bound. Conversely, programs that terminate with probability 1 may still have unbounded expected cost, e.g., a symmetric random walk over natural numbers that stops at 0 [McIver and Morgan 2005].

AARA for Expected Cost. Now reconsider the potential method in the presence of probabilistic operations, that is, the cost and the next state of an operation are given by distributions. Let $o(S)$ denote the probability distribution over possible next states induced by o operating on S. One can derive bounds on the worst-case *expected* cost by requiring that the following inequality for the potential function holds over all states S and operations o. We use the notation $\mathbb{E}_{S' \sim o(S)}$ (defined in §3) to weight expected cost over states S' by the probability given by $o(S)(S')$.

$$\Phi(S) \ge \mathbb{E}_{S' \sim o(S)}(cost(S, S') + \Phi(S')) = \mathbb{E}_{S' \sim o(S)}(cost(S, S')) + \mathbb{E}_{S' \sim o(S)}(\Phi(S')),$$

The intuitive meaning of the inequality is that the potential $\Phi(S) \ge 0$ is sufficient to pay for the expected cost of the operation o from the state S, and the expected potential of the next state S' with respect to the probability distribution $o(S)$.

Further, if for some operation o' we have $\Phi(S') \ge \mathbb{E}_{S'' \sim o'(S')}(cost(S', S'')) + \mathbb{E}_{S'' \sim o'(S')}(\Phi(S''))$ for each state S' the could succeed S under o, then we can *compose* the reasoning for o and o' as follows.

$$\begin{aligned}
\Phi(S) &\ge \mathbb{E}_{S' \sim o(S)}(cost(S, S')) + \mathbb{E}_{S' \sim o(S)}(\Phi(S')) \\
&\ge \mathbb{E}_{S' \sim o(S)}(cost(S, S')) + \mathbb{E}_{S' \sim o(S)}\left[\mathbb{E}_{S'' \sim o'(S')}(cost(S', S'')) + \mathbb{E}_{S'' \sim o'(S')}(\Phi(S''))\right] \\
&= \mathbb{E}_{S' \sim o(S), S'' \sim o'(S')}(cost(S, S') + cost(S', S'')) + \mathbb{E}_{S' \sim o(S), S'' \sim o'(S')}(\Phi(S'')).
\end{aligned}$$

Thus, the potential $\Phi(S)$ is sufficient to cover the expected cost of operations o and o', as well as the expected potential of the final state. This can be sequenced indefinitely to cover all operations of an entire program. A valid potential assignment for the initial state of the program then provides an *upper* bound on the *expected* total cost of running the program.

In §4, we extend the AARA type system to support this kind of potential-method reasoning while preserving the benefits of AARA such as compositionality and reduction of type inference to LP solving. For example, our probabilistic extension to AARA can type the code of the function *bernoulli* in Fig. 1b as

$$bernoulli : \langle L^0(\tau), 1\rangle \to \langle \mathsf{bool}, 0\rangle$$

where the input can be typed as a list with 0 units of potential per element (assuming τ does not assign potential). To cover the expected cost, it only needs 1 available potential unit per run, indicated by the 1 paired with the input type. When typing the probabilistic *flip*, this single unit of potential can pay for the expected cost of the two equally-likely branches: The H branch costs 0, the T branch costs 2 (1 each for the recursive call and for *bernoulli* to consume), and they average to 1. As *bernoulli* can be typed to consume 1 unit of potential, the upper bound AARA finds is exact.

The functions *bernoulli* and *exists* form an example of the automatic average-case estimation algorithm that we introduce in §7.2. Assume that you want to run *exists* on a certain distribution of inputs and you want to determine the average cost of *exists* on this distribution. To approximately answer this question, we collapse code like *exists* into code like *bernoulli* and use pRaML to estimate that the average cost is 1. In this case, such a collapse would be justified by finding empirically that *pred* holds with probability 0.5.

The technical innovation that makes possible the typing of *bernoulli* is a new typing rule for probabilistic branching. Another innovation is the introduction of the type $\mathbb{P}$ for probabilities. The introduction form for values of type $\mathbb{P}$ simply takes a rational number $0 \le p \le 1$ and the elimination form is a probabilistic branch. We can assign potential

$$\Phi(p : \mathbb{P}^{q_H}_{q_T}) \overset{\text{def}}{=} q_H \cdot p + q_T \cdot (1 - p)$$

to a value p of type $\mathbb{P}$. The potential q_H and q_T then becomes available in the head and tails cases, respectively, of the probabilistic branching.

Consider for example the function *rdwalk* in Fig. 1c again. Our probabilistic analysis can automatically derive the typing

$$rdwalk : \langle L^1(\mathbb{P}^5_0), 0\rangle \to \langle \mathsf{unit}, 0\rangle\ .$$

The potential of the argument

$$\Phi([p_1, \dots, p_n] : \langle L^1(\mathbb{P}^5_0), 0\rangle) = n + \sum_{1\le i\le n} 5p_i\ ,$$

corresponds to the exact bounds on the expected cost.

Here we present these novel ideas for a simple functional language with lists and linear potential functions. However, the results carry over to user-defined inductive types and multivariate polynomial potential functions of RaML [Hoffmann et al. 2017] that we use in the implementation. The main theorem of this paper (see §5) states that the expected cost bounds are sound, with respect to a step-indexed distribution-based operational semantics inspired by Borgström et al.'s semantics for the probabilistic lambda calculus [Borgström et al. 2016]. We then extend the semantics with *partial evaluations* to capture the resource behavior of non-terminating executions of a probabilistic program. This novel extension enables an improved soundness result, which implies that expected bounds on run-times ensure termination with probability 1.

3 LANGUAGE AND SEMANTICS

In this section, we introduce a subset of pRaML as a functional ML-like language that includes units, lists, recursion, pattern match, and a new *flip* expression for probabilistic branching. We then present an initial form of our operational cost semantics for probabilistic programs, which keeps track of both the probability and the cost of executions. We will use this language and semantics to formalize and justify our type-based expected cost analysis in §4 and §5.

		Abstract	**Concrete**	
e	$::=$	x	x	variable
		triv	$\langle\rangle$	null tuple
		nil	$[]$	empty list
		$\mathrm{cons}(x_1; x_2)$	$x_1 :: x_2$	cons list
		$\mathrm{mat_L}\{e_0; x_1, x_2.e_2\}(x)$	$\mathsf{case}\ x\ \{\ \mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\ (x_1, x_2) \hookrightarrow e_2\}$	pattern match
		$\mathrm{fun}(f, x.e)$	$\mathsf{fun}\ f x = e$	function
		$\mathrm{app}(x_1; x_2)$	$x_1(x_2)$	application
		$\mathrm{tick}\{q\}$	$\mathsf{tick}\ q$	cost
		$\mathrm{let}(e_1; x.e_2)$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	definition
		$\mathrm{share}(x; x_1, x_2.e)$	$\mathsf{share}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ e$	sharing
		$\mathrm{flip}\{e_1; e_2\}(p)$	$\mathsf{flip}\ p\ \{\mathsf{H} \hookrightarrow e_1 \mid \mathsf{T} \hookrightarrow e_2\}$	coin flip
		$\mathrm{prob}\{p\}$	p	probability
		$\mathrm{flip}_{\mathbb{S}}(x; e_1, e_2)$	$\mathsf{flip}_{\mathbb{S}}\ x\ \{\mathsf{H} \hookrightarrow e_1 \mid \mathsf{T} \hookrightarrow e_2\}$	symbolic flip

Fig. 2. Syntax of the language

Syntax. We only consider expressions in *share-let-normal-form* [Hoffmann et al. 2011]. This is a syntactic form that uses variables instead of arbitrary terms whenever possible, without loss of expressivity. This is done through maximizing the use of let-expressions. The syntax also must use $\mathrm{share}(x; x_1, x_2.e)$ to allow multiple uses of a variable x in an expression e, due to linear properties of the type system. The abstract and concrete syntax of our probabilistic programming language is given by the grammar in Fig. 2. Abstract syntax is given via abstract binding trees [Harper 2016]. While the concrete syntax matches the intuitive meaning of each expression, the abstract syntax conveys the same information and compacts some overly large expressions, allowing them to be written down more succinctly.

The syntactic form $\mathrm{flip}\{e_1; e_2\}(p)$ is introduced to execute e_1 or e_2 at random. The intuitive meaning of the flip expression is to flip a biased coin, which shows heads with probability p and tails with probability $(1 - p)$, then execute e_1 if the coin shows heads, or execute e_2 if the coin shows tails. Additionally, the introduction form $\mathrm{prob}\{p\}$ and the elimination form $\mathrm{flip}_{\mathbb{S}}(x; e_1, e_2)$ are provided for the new *probability* type: $\mathrm{prob}\{p\}$ encapsulates a rational number $0 \le p \le 1$ for probability, and $\mathrm{flip}_{\mathbb{S}}(x; e_1, e_2)$ is essentially the same as flip expressions except that the branching probability is specified by a variable x of probability type. The syntactic form $\mathrm{share}(x; x_1, x_2.e)$ has to be used to allow multiple uses of a variable x in an expression e.

Elementary probability theory. We recount some essential concepts from elementary probability theory. You can find more serious mathematical development of probabilities in textbooks on measure theory [Billingsley 2012; Williams 1991].

Consider a random experiment. Let Ω denote the set of all the possible outcomes, called the *sample space*. A discrete *probability space* is a pair $(\Omega, \mathbb{P})$, where $\mathbb{P} : \Omega \to [0, 1]$ is a *probability distribution* on Ω, i.e., $\sum_{\omega \in \Omega} \mathbb{P}(\omega) = 1$. The probability of an *event* $E \subseteq \Omega$, written $\mathbb{P}(E)$, is defined as $\sum_{\omega \in E} \mathbb{P}(\omega)$. We often write $\mathbb{P}(\theta)$ for the probability of a statement θ, i.e., $\mathbb{P}(\{\omega \mid \theta(\omega) \text{ is true}\})$. A *random variable* $X : \Omega \to \mathbb{R} \cup \{-\infty, +\infty\}$ is a function from a probability space to the extended real numbers. The *expected value* of a random variable X is the weighted average $\mathbb{E}_{\omega \sim (\Omega, \mathbb{P})}(X) \stackrel{\text{def}}{=} \sum_{\omega \in \Omega} X(\omega) \cdot \mathbb{P}(\omega)$. We often write $\mathbb{E}(X)$ if there is no ambiguity in the choice of the probability space. An important property of expected value is *linearity*: If X and Y are random variables and $a, b \in \mathbb{R}$, then $(aX + bY)$ is a random variable and $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$.

Obstacles for probabilistic semantics. To define the expected resource usage of probabilistic programs, we formulate a cost semantics based on an evaluation dynamics. This turns out to be challenging. Previous work on AARA cost semantics for non-probabilistic programs lack the infrastructure to reason about certain effects of probabilistic phenomena. One such example is the poor behaviour of high-water marks: the well-known probabilistic Martingale betting strategies have an unbounded expected high-water mark, even while having finite expected net gain. In this section we describe the sorts of problems faced from the perspective of the cost semantics.

The notion of values in the cost semantics can proceed unchanged: *value* $v \in \mathsf{Val}$ is either a null tuple $\langle\rangle$, an empty list $[]$, a cons list $v_1 :: v_2$, or a *function closure* $\mathsf{clo}(V; f, x.e)$ that consists of an *environment* $V : \mathsf{Var} \to \mathsf{Val}$ and a function definition $\mathsf{fun}\ f x = e$. However, the evaluation cost dynamics surrounding such values must be altered to deal with probability. In prior work on AARA [Hoffmann and Hofmann 2010a], the cost semantics is defined by a judgment of the form

$$V \vdash e \Downarrow v \mid (q, q')$$

This judgment means that, under an evaluation environment V, the expression e evaluates to the value v using a high-water mark of $q \in \mathbb{Q}_{\geq 0}$ resources and leaving $q' \in \mathbb{Q}_{\geq 0}$ resources leftover. By tracking both the high-water mark and leftover resources, non-probabilistic AARA was able to reason about resources that might be returned after use, like space. This tracking is performed by the *resource monoid* [Hoffmann and Hofmann 2010a], which algebraically composes the high-water mark/leftover pairs (q, q').

Unfortunately, this operational judgment does not adapt to the probabilistic domain. Firstly, it distinguishes a particular value v for evaluation, rather than a distribution. Further, the resource monoid does not compose under probability. Both points must be remedied to soundly model cost.

To illustrate the resource monoid problem, we first define it. The following accounts for how the high-water mark and leftover resource constraints change under non-probabilistic composition.

$$(a, b) \cdot (c, d) \overset{\text{def}}{=} (a + \max(c - d, 0), d + \max(d - c, 0))$$

Now consider the following two expressions. Letting e_1 have the associated resource monoid term $(0, 0)$, e_2 have $(4, 2)$ and e_3 have $(2, 1)$, we see both have an expected high-water mark resource usage of 2 and expected leftover of 1.

$$\mathsf{flip}\ \frac{1}{2}\ \{\mathsf{H} \hookrightarrow e_1 \mid \mathsf{T} \hookrightarrow e_2\} \quad vs \quad e_3$$

However, we cannot represent the resource constraints of both expressions uniformly with the term $(2, 1)$. This becomes apparent if we precede both expressions by a copy of e_3, as then the expected high-water mark of each differs. The latter can be correctly calculated to be 3 with $(2, 1) \cdot (2, 1) = (3, 1)$. However, the high-water mark of the former would be 3.5 since half the time it would be 2 and half the time 5. There is no way to get two different results as a function of the same input $(2, 1)$, so two-place resource monoid terms cannot be salvaged for probabilistic use.

To avoid this problem, we forgo the high-water mark/leftover resource distinction, and reason only about resources that *monotonically* decrease, like time. It then suffices to track only the net cost with a more well-behaved one-place term. As a result, the AARA system described here only consumes resources, and never provides them.

This restriction to monotonically consumed resources solves an additional problem for the cost semantics concerning the well-definedness of expected cost in the presence of nontermination. Even programs with finite expected cost may have nonterminating executions. However, if the execution can be non-terminating, there can be an infinite number of execution traces, and thus the expected value of their cost is defined over an infinite sum. Such a sum must converge absolutely to represent an expected value, and if the costs for operations can have different signs this is not clearly the case. Recent work [Wang et al. 2019] has proposed techniques to reason about non-monotone resources for imperative programs; adapting these techniques to analyze functional programs is beyond the scope of this paper, but is an interesting future research direction.

Besides the cost, a probabilistic semantics must also account for probabilistic execution resulting in a distribution of values, rather than 1 particular value. To solve this problem, one might first think to reason about individual executions separately by adding a component that tracks the probability of a particular value resulting. By collecting such judgments with probabilities adding to 1, one could then recover the desired value distributions. For this approach, one might create the judgment $V \vdash e \Downarrow^p v \mid q$ which would mean that there exists *an* execution where the expression e evaluates to the value v with net cost q and probability p. However, this approach has a subtle problem: There might be multiple different executions with the same evaluation result, cost, and probability. For example, consider the following program

$$e \equiv \mathsf{flip}\ \frac{1}{2}\ \{\mathsf{H} \hookrightarrow \mathsf{tick}\ 2 \mid \mathsf{T} \hookrightarrow \mathsf{let}\ \_ = \mathsf{tick}\ 1\ \mathsf{in}\ \mathsf{tick}\ 1\}.$$

Although the program has two possible syntactically-distinct executions, there is only one valid evaluation relation derivable from the given rules, which is

$$\cdot \vdash e \Downarrow^{1/2} \langle\rangle \mid 2.$$

This thwarts the idea of collecting relations with probabilities summing to 1, as some relations would need to be counted multiple times, and the present components to the judgment leave no way to determine the multiplicity. To solve these problems, we present the following cost semantics.

Trace-based cost semantics. We deal with obstacles surrounding cost semantics by adapting Borgström et al.'s trace-based semantics for lambda calculus [Borgström et al. 2016] to our setting. The key observation is that an execution is uniquely determined by the *trace* of outcomes of the coin flips in the execution. We augment the evaluation relation with a component for traces, i.e., a finite sequence of elements in $\{\mathsf{H}, \mathsf{T}\}$. The trace-based evaluation judgment then has the form

$$V; \sigma \vdash e \Downarrow^p v \mid q,$$

The intuitive meaning is that under the environment V, with a sequence σ of coin-flip outcomes, the expression e evaluates to a value v with cost q and probability p.

Fig. 3 presents the rules for this trace-based evaluation dynamics. We write $[]$ for empty traces, $\sigma_1 @ \sigma_2$ for trace concatenation, and $\mathsf{H} :: \sigma$ or $\mathsf{T} :: \sigma$ to observe a new coin flip and prepend the outcome to σ. In the rule E:Let, we multiply the probabilities of an execution of e_1 and an execution of e_2, as well as concatenate their traces of coin flips.

Recall that in order to reason about expected resource usage, we need a notion of *probability distributions* over executions, and found that accounting for the multiplicity of operational judgments made this difficult. With the trace-based dynamics, we can now capture all the terminating executions uniquely. This is because the result value v, the net cost q, and the probability p, are determined uniquely by the environment V, the expressions e, and the trace of coin flips σ.

By induction on the structure of expression e, we prove the lemma below.

Lemma 3.1. *For all V, e and σ, there is at most one combination of v, q and p s.t. $V; \sigma \vdash e \Downarrow^p v \mid q$.*

Therefore, for fixed V and e, the set of all finite traces induces a "distribution" over terminating executions. We can extract a "distribution" $[\![e]\!]^V_\Downarrow$ on values v and costs q as follows:

$$[\![e]\!]^V_\Downarrow(v, q) \stackrel{\text{def}}{=} \sum_\sigma p_\sigma \quad \text{where } \sigma\text{'s are finite traces satisfying } V; \sigma \vdash e \Downarrow^{p_\sigma} v \mid q.$$

Note that if there are non-terminating executions with non-zero probabilities, the map defined above is a subprobability distribution in the sense that the probabilities do *not* sum up to one. In other words, the probability that e diverges under environment V is $(1 - \sum_{(v,q)} [\![e]\!]^V_\Downarrow(v, q))$.

With this trace-based cost semantics in hand, we can finally define the expected cost of evaluating some terminating expression e with variable bindings given the values of V. The expected cost is

$\boxed{V; \sigma \vdash e \Downarrow^p v \mid q \quad \text{“in environment } V\text{, with trace } \sigma\text{, expression } e \text{ evaluates to value } v \text{ with cost } q \text{ and probability } p\text{”}}$

$$\frac{}{V; [] \vdash x \Downarrow^1 V(x) \mid 0}\ (\text{E:Var}) \qquad \frac{}{V; [] \vdash \mathsf{triv} \Downarrow^1 \langle\rangle \mid 0}\ (\text{E:Triv}) \qquad \frac{}{V; [] \vdash \mathsf{nil} \Downarrow^1 [] \mid 0}\ (\text{E:Nil})$$

$$\frac{V(x_1) = v_1 \qquad V(x_2) = v_2}{V; [] \vdash \mathsf{cons}(x_1; x_2) \Downarrow^1 v_1 :: v_2 \mid 0}\ (\text{E:Cons}) \qquad \frac{V(x) = [] \qquad V; \sigma \vdash e_0 \Downarrow^p v \mid q}{V; \sigma \vdash \mathsf{mat_L}\{x; e_0, x_1.x_2\}(e_1) \Downarrow^p v \mid q}\ (\text{E:MatL-1})$$

$$\frac{V(x) = v_1 :: v_2 \qquad V, x_1 \mapsto v_1, x_2 \mapsto v_2; \sigma \vdash e_1 \Downarrow^p v \mid q}{V; \sigma \vdash \mathsf{mat_L}\{x; e_0, x_1.x_2\}(e_1) \Downarrow^p v \mid q}\ (\text{E:MatL-2}) \qquad \frac{}{V; [] \vdash \mathsf{tick}\{q\} \Downarrow^1 \langle\rangle \mid q}\ (\text{E:Tick})$$

$$\frac{V; \sigma_1 \vdash e_1 \Downarrow^{p_1} v_1 \mid q_1 \qquad V, x \mapsto v_1; \sigma_2 \vdash e_2 \Downarrow^{p_2} v_2 \mid q_2}{V; \sigma_1 \mathbin{@} \sigma_2 \vdash \mathsf{let}(e_1; x.e_2) \Downarrow^{p_1 \cdot p_2} v_2 \mid q_1 + q_2}\ (\text{E:Let}) \qquad \frac{}{V; [] \vdash \mathsf{fun}(f, x.e) \Downarrow^1 \mathsf{clo}(V; f, x.e) \mid 0}\ (\text{E:Fun})$$

$$\frac{V(x_1) = \mathsf{clo}(V'; f, x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \mathsf{clo}(V'; f, x.e), x \mapsto v_2; \sigma \vdash e \Downarrow^p v \mid q}{V; \sigma \vdash \mathsf{app}(x_1; x_2) \Downarrow^p v \mid q}\ (\text{E:App})$$

$$\frac{V; \sigma \vdash e_1 \Downarrow^{p_1} v_1 \mid q_1}{V; \mathsf{H} :: \sigma \vdash \mathsf{flip}\{e_1; e_2\}(p) \Downarrow^{p \cdot p_1} v_1 \mid q_1}\ (\text{E:Flip-1}) \qquad \frac{V; \sigma \vdash e_2 \Downarrow^{p_2} v_2 \mid q_2}{V; \mathsf{T} :: \sigma \vdash \mathsf{flip}\{e_1; e_2\}(p) \Downarrow^{(1-p) \cdot p_2} v_2 \mid q_2}\ (\text{E:Flip-2})$$

$$\frac{V(x) = v \qquad V, x_1 \mapsto v, x_2 \mapsto v; \sigma \vdash e \Downarrow^p v' \mid q}{V; \sigma \vdash \mathsf{share}(x; x_1, x_2.e) \Downarrow^p v' \mid q}\ (\text{E:Share}) \qquad \frac{}{V; [] \vdash \mathsf{prob}\{p\} \Downarrow^1 \mathsf{prob}(p) \mid 0}\ (\text{E:Prob})$$

$$\frac{V(x) = \mathsf{prob}(p) \qquad V; \sigma \vdash e_1 \Downarrow^{p_1} v_1 \mid q_1}{V; \mathsf{H} :: \sigma \vdash \mathsf{flip}_{\mathbb{S}}(x; e_1, e_2) \Downarrow^{p \cdot p_1} v_1 \mid q_1}\ (\text{E:FlipS-1}) \qquad \frac{V(x) = \mathsf{prob}(p) \qquad V; \sigma \vdash e_2 \Downarrow^{p_2} v_2 \mid q_2}{V; \mathsf{T} :: \sigma \vdash \mathsf{flip}_{\mathbb{S}}(x; e_1, e_2) \Downarrow^{(1-p) \cdot p_2} v_2 \mid q_2}\ (\text{E:FlipS-2})$$

Fig. 3. Evaluation rules of the trace-based cost semantics

just the sum of costs q weighted by probability p over all execution traces σ.

$$\sum_{\sigma : V; \sigma \vdash e \Downarrow^p v \mid q} p \cdot q = \sum_{V, v, q} [\![e]\!]^V_{\Downarrow}(v, q) \cdot q$$

However, generalizing this definition for non-termination would be nontrivial. As probability is only countably additive, and the set of infinite traces of a non-terminating execution may be uncountable, the above sum could no longer be used. It would appear that a more complicated summation mechanism like integration over a cost density function would be required to deal with such divergence, and we do not deign to develop that here. Instead, to deal with this concern and others, the cost semantics will be revisited in §5. There we will do like [Borgström et al. 2016] and convert from trace-based to distribution-based semantics.

4 TYPE SYSTEM

In this section, we develop an AARA type system to carry out expected cost analysis for probabilistic programs. To focus on the changes that probabilistic choice induces on the type system, we describe its action here in *linear* AARA, where all potential functions are linear in terms of list sizes. In other work, potential functions have been expanded to cover polynomials [Hoffmann et al. 2011; Hoffmann and Hofmann 2010b] and exponentials [Kahn and Hoffmann 2020], but this extension to AARA is orthogonal to probabilistic choice. Indeed, we have carried over the implementation

		Abstract	**Concrete**	
τ	::=	unit	$\mathbb{1}$	nullary product
		list(A)	$L^q(\tau)$	list
		arr(A; B)	$A \multimap B$	arrow
		prob$\{q_H; q_T\}$	$\mathbb{P}^{q_H}_{q_T}$	probability
A, B	::=	pot(τ; q)	$\langle \tau, q \rangle$	potential

Fig. 4. Syntax of the type system

and soundness of probabilistic AARA to support multivariate-polynomial potential functions and user-defined datatypes without problem, which we use to perform analyses in §6 and beyond.

Types and potentials. Fig. 4 presents the types that are supported in linear AARA. Aside from usual types like the nullary $\mathbb{1}$ and binary product $\tau_1 \times \tau_2$, there are three special types that have potential-related components. The first is the potential pairing $\langle \tau, q \rangle$, which represents storing a constant $q \in \mathbb{Q}_{\geq 0}$ units of potential alongside a value of type τ. The second is the list type $L^q(\tau)$—a compact representation of $\text{list}(\langle \tau, q \rangle)$—which represents a list with $q \in \mathbb{Q}_{\geq 0}$ units of potential per element. The combination is sufficient to express potential functions that are linear combinations of input list lengths and constants. The last is the probability type $\text{prob}\{q_H; q_T\}$. As introduced in §2, it represents q_H units of potential for head cases and q_T units for tail cases after a coin flip.

Formally, the *potential function* $\Phi(\cdot : \tau)$ or $\Phi(\cdot : A)$, which maps values of type τ or A to non-negative rational numbers, is defined as follows.

$$\Phi(\langle\rangle : \text{unit}) \overset{\text{def}}{=} 0, \qquad \Phi(v : \text{pot}(\tau; q)) \overset{\text{def}}{=} \Phi(v : \tau) + q,$$

$$\Phi([\,] : \text{list}(A)) \overset{\text{def}}{=} 0, \qquad \Phi(v_1 :: v_2 : \text{list}(A)) \overset{\text{def}}{=} \Phi(v_1 : A) + \Phi(v_2 : \text{list}(A)),$$

$$\Phi(\text{clo}(V; f, x.e) : \text{arr}(A; B)) \overset{\text{def}}{=} 0, \quad \Phi(\text{prob}(p) : \text{prob}\{q_H; q_T\}) \overset{\text{def}}{=} q_H \cdot p + q_T \cdot (1 - p).$$

From the inductive definition above, we can derive the following closed form for the potential of a list $\ell = [v_1, \cdots, v_n]$ with respect to a type $L^q(\tau)$, which is linear in the length of the list ℓ.

$$\Phi(\ell : L^q(\tau)) = q \cdot n + \sum_{i=1}^{n} \Phi(v_i : \tau).$$

Note that these definitions leave potential as a function of both type and value. Different values of the same type may differ in their total potential. For instance, in the case of lists, one term in the above closed form for potential depends on the length of the list, so lists of differing lengths but the same type may differ in total potential.

Static semantics. The typing judgment for linear AARA the form $\Gamma; q \vdash e : A$, the intuitive meaning of which is that the potential given by Γ and q is sufficient to cover the *expected* evaluation cost of e and the *expected* potential of the evaluation result with respect to A.

As existing AARA type systems, our typing rules form an *affine* linear type system, which ensures that every variable is used *at most* once [Walker 2002]. Fig. 5 lists the typing rules. It turns out that most of the rules coincide with those of non-probabilistic linear AARA systems. This fact indicates that our type system is a conservative extension of non-probabilistic AARA for monotonic resources, and our type system is able to derive worst-case cost bounds for deterministic programs.

To understand the new rule L:Flip for probabilistic branching, consider the expression $\text{flip}\{e_1; e_2\}(p)$, where e_1 requires Φ_1 units of potential and e_2 requires Φ_2. The evaluation of the flip expression should expect to require a weighted average of Φ_1 and Φ_2, specifically $p \cdot \Phi_1 + (1-p) \cdot \Phi_2$. This should be paid out of the typing context Γ and constant potential q, both of which are shared between branches. The distribution of this sharing is expressed using a *sharing relation* $\tau \curlyvee (\tau_1, \tau_2)$, which apportions the potential indicated by τ into two parts to be associated with τ_1 and τ_2, alongside a *potential-scaling* operation. We formally define these relations and prove they capture the correct intuition with Lemmas 4.1 and 4.2, but first we explain why the rule does *not* also perform expected value calculations for the type A.

One might think that a similar weighted average could be used to combine the types of e_1 and e_2 to get *expected* type A, rather than require both expressions have type A exactly. Perhaps equally likely types $L^3(\tau)$ and $L^1(\tau)$ could convert to expected type $L^2(\tau)$. However, the value produced in each branch might differ, and for lists of type $L^q(\tau)$ total potential is a scalar q of length; taking the expected value of the scalars without accounting for length does not succeed in finding the expected potential. Thus, the rule L:Flip cannot be made more permissive in that manner.

Nonetheless, note that the same return type for both branches in L:Flip still can leave differing potential after each branch, which is necessary for expected cost reasoning. For example, consider the following program where the function *append* requires 1 unit of potential per element in its first argument.

```
append (flip 0.5 | H →[1;2;3] | T → [0]) [5;6]
```

The return types of the two branches of the flip expression are the same ($L^2(\text{int})$), but the actual potential in the results is different: the heads branch returns a list with 3 units of potential, and the tails branch returns 1 unit. This shows that the analysis properly composes and correctly reasons that expected cost of the program is 2. This also works for symbolic lists and can derive the bound $|x| + |y|$ for the function

```
fun x y → append (flip 0.5 | H →y | T →(append x y)) []
```

Now we formalize the sharing and scaling relations. The sharing relation for types is defined as follows. Note that the sharing relation is also used in L:Share to make "copies" of a variable, while ensuring that the total potential over copies is preserved.

$$\frac{}{\text{unit} \curlyvee (\text{unit}, \text{unit})}\ (\textsc{Sh:Unit}) \qquad \frac{A \curlyvee (A_1, A_2)}{\text{list}(A) \curlyvee (\text{list}(A_1), \text{list}(A_2))}\ (\textsc{Sh:List})$$

$$\frac{}{\text{arr}(A; B) \curlyvee (\text{arr}(A; B), \text{arr}(A; B))}\ (\textsc{Sh:Arrow}) \qquad \frac{q_H = q_H^{(1)} + q_H^{(2)} \qquad q_T = q_T^{(1)} + q_T^{(2)}}{\text{prob}\{q_H; q_T\} \curlyvee (\text{prob}\{q_H^{(1)}; q_T^{(1)}\}, \text{prob}\{q_H^{(2)}; q_T^{(2)}\})}\ (\textsc{Sh:Prob})$$

$$\frac{q = q_1 + q_2 \qquad \tau \curlyvee (\tau_1, \tau_2)}{\text{pot}(\tau; q) \curlyvee (\text{pot}(\tau_1; q_1), \text{pot}(\tau_2; q_2))}\ (\textsc{Sh:Pot})$$

We extend the sharing relation to typing contexts, as it has previously only been used on a per-type basis. This splits the potential across all types in Γ across 2 new contexts of the same base types.

$$\frac{}{\cdot \curlyvee (\cdot, \cdot)}\ (\textsc{Sh:Empty}) \qquad \frac{\Gamma \curlyvee (\Gamma_1, \Gamma_2) \qquad \tau \curlyvee (\tau_1, \tau_2)}{\Gamma, x : \tau \curlyvee (\Gamma_1, x : \tau_1, \Gamma_2, x : \tau_2)}\ (\textsc{Sh:Extend})$$

Potential-scaling can be defined syntactically as follows. Intuitively, $p \times \tau$ (resp., $p \times A$) produces a type with as much potential as that of the original type τ (resp., A) scaled by the factor p.

$$p \times \text{unit} \overset{\text{def}}{=} \text{unit}, \qquad p \times \text{pot}(\tau; q) \overset{\text{def}}{=} \text{pot}(p \times \tau; p \cdot q),$$

$$p \times \text{list}(A) \overset{\text{def}}{=} \text{list}(p \times A), \qquad p \times \text{arr}(A; B) = \text{arr}(A; B),$$

$$p \times \text{prob}\{q_H; q_T\} \overset{\text{def}}{=} \text{prob}\{p \cdot q_H; p \cdot q_T\}.$$

Also, we extend the scaling operation to typing contexts.

$$p \times (\cdot) \overset{\text{def}}{=} \cdot, \qquad p \times (\Gamma, x : \tau) \overset{\text{def}}{=} p \times \Gamma, x : (p \times \tau).$$

By induction on the structure of value v, we prove the following lemmas that ensure the sharing and scaling relations are consistent with their intuitive meaning.

Lemma 4.1. *For any value v of type τ (resp., A), if $\tau \curlyvee (\tau_1, \tau_2)$ (resp., $A \curlyvee (A_1, A_2)$), then $\Phi(v : \tau) = \Phi(v : \tau_1) + \Phi(v : \tau_2)$ (resp., $\Phi(v : A) = \Phi(v : A_1) + \Phi(v : A_2)$).*

Lemma 4.2. *For any probability p and value v of type τ (resp., A), $\Phi(v : p \times \tau) = p \cdot \Phi(v : \tau)$ (resp., $\Phi(v : p \times A) = p \cdot \Phi(v : A)$).*

We now discuss the rule L:Prob and L:FlipS for the new probability type $\mathbb{P}^{q_H}_{q_T}$. To type a probability encapsulation prob$\{p\}$, we need $\Phi(\text{prob}(p) : \mathbb{P}^{q_H}_{q_T}) = p \cdot q_H + (1 - p) \cdot q_T$ units of potential in the context to cover its expected value. Then to type an expression $\text{flip}_{\mathbb{S}}(x; e_1, e_2)$ that flips a variable x with type $\mathbb{P}^{q_H}_{q_T}$, one might want to use the potential-scaling operation as the rule L:FLIP does. However, the probability x here is *symbolic*, thus we cannot define the scaling operation in linear AARA.[1] The rule L:FLIPS avoids the problem by forcing e_1 and e_2 to be typed under the same context, appealing to the equality $\Phi = x \cdot \Phi + (1 - x) \cdot \Phi$ for any x and Φ. Note that it assigns q_H units of potential to type e_1, and q_T units to type e_2; this assignment is sound because we pay $x \cdot q_H + (1 - x) \cdot q_T$ to create $x : \mathbb{P}^{q_H}_{q_T}$.

Finally, we briefly explain other typing rules. In the rule L:CONS, we have to provide potential p to account for the potential of the new list element. Conversely, the potential of the head x_1 of the list $x : L^p(\tau)$ becomes available in the cons branch of the pattern match in the rule L:MATL. As a result, we have constant potential $p + q$ available when typing e_1. In the rule L:APP, we require that we have the exact potential annotations ($x_2 : \tau$ and q) that are required by the argument. The resulting potential is given by the result type B. In the rule L:FUN for (recursive) function abstraction, we require that the potential of the variables captured in the context Γ is zero. We write $|\Gamma|$ for the context Γ in which every potential annotation q is replaced by 0. This is formally defined below. The reason for this requirement is that we allow functions to be used an arbitrary number of times (recall the definition of sharing). If Γ would carry potential then we could use this potential multiple times to account for cost, which is not sound. Since functions do not carry potential, we do not have to restrict the type of the recursively defined function f in a similar way. An alternative would be to remove the premise $\Gamma = |\Gamma|$ and to treat functions in an affine way.

$$|\text{unit}| \overset{\text{def}}{=} \text{unit}, \qquad |\text{pot}(\tau; q)| \overset{\text{def}}{=} \text{pot}(|\tau|; 0),$$
$$|\text{list}(A)| \overset{\text{def}}{=} \text{list}(|A|), \qquad |\text{arr}(A; B)| \overset{\text{def}}{=} \text{arr}(A; B),$$
$$|\text{prob}\{q_H; q_T\}| \overset{\text{def}}{=} \text{prob}\{0; 0\}.$$

Note that for function types, we do not have to recursively eliminate potential with $|\cdot|$ since the potential of a function is already 0. The definition is then lifted point-wise to annotated contexts Γ.

$$|\cdot| \overset{\text{def}}{=} \cdot, \qquad |\Gamma, x : \tau| \overset{\text{def}}{=} |\Gamma|, x : |\tau|.$$

The structural rules L:SUB, L:SUP, L:WEAK, and L:RELAX can be applied to every expression. The weakening rule L:WEAK is standard. However, there is another form of weakening: The rule L:RELAX, states that, given a judgment $\Gamma; p \vdash e : \langle \tau, p' \rangle$, we can also have more potential q in the context and give up some of the potential p'. Additionally, the rule also covers the case in which we pass through additional potential $c \geq 0$ yielding the judgment $\Gamma; p + c \vdash e : \langle \tau, p' + c \rangle$. The subtyping rules L:SUB and L:SUP enable us to relax the potential requirements for potential in data structures in the same way as T:RELAX does for constant potential. The subtyping relation for types is defined by the following rules.

$$\frac{}{\text{unit} <: \text{unit}} \qquad \frac{A <: B}{\text{list}(A) <: \text{list}(B)}$$

$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{\text{arr}(A_1; B_1) <: \text{arr}(A_2; B_2)} \qquad \frac{q_H^{(1)} \geq q_H^{(2)} \qquad q_T^{(1)} \geq q_T^{(2)}}{\text{prob}\{q_H^{(1)}; q_T^{(1)}\} <: \text{prob}\{q_H^{(2)}; q_T^{(2)}\}} \qquad \frac{q_1 \geq q_2 \qquad \tau_1 <: \tau_2}{\text{pot}(\tau_1; q_1) <: \text{pot}(\tau_2; q_2)}$$

By induction on the structure of value v followed by inversion on the subtyping judgment, we prove the following lemma.

LEMMA 4.3. *If $\tau <: \tau'$ then $\Phi(v : \tau') \leq \Phi(v : \tau)$ for any value v of type τ.*

[1]In our implementation of pRaML, we use multivariate polynomial AARA to support symbolic scaling, which unifies the two distinct flip operations presented here. We also incorporate the ability to multiply and complement symbolic probabilities.

$$\boxed{\Gamma; q \vdash e : A \quad \text{“in context } \Gamma \text{ with constant potential } q \text{, expression } e \text{ has potential-annotated type } A\text{”}}$$

$$\frac{}{x : \tau; 0 \vdash x : \langle \tau, 0 \rangle}\ (\text{L:Var}) \qquad \frac{}{\cdot; 0 \vdash \text{triv} : \langle \text{unit}, 0 \rangle}\ (\text{L:Unit}) \qquad \frac{}{\cdot; 0 \vdash \text{nil} : \langle \text{list}(A), 0 \rangle}\ (\text{L:Nil})$$

$$\frac{A = \langle \tau, p \rangle}{x_1 : \tau, x_2 : \text{list}(A); p \vdash \text{cons}(x_1; x_2) : \langle \text{list}(A), 0 \rangle}\ (\text{L:Cons})$$

$$\frac{A = \langle \tau, p \rangle \qquad \Gamma; q \vdash e_0 : B \qquad \Gamma, x_1 : \tau, x_2 : \text{list}(A); q + p \vdash e_1 : B}{\Gamma, x : \text{list}(A); q \vdash \text{mat}_{\text{L}}\{e_0; x_1, x_2.e_1\}(x) : B}\ (\text{L:MatL}) \qquad \frac{}{\cdot; q \vdash \text{tick}\{q\} : \langle \text{unit}, 0 \rangle}\ (\text{L:Tick})$$

$$\frac{\Gamma_1; q \vdash e_1 : \langle \tau, p \rangle \qquad \Gamma_2, x : \tau; p \vdash e_2 : B}{\Gamma_1, \Gamma_2; q \vdash \text{let}(e_1; x.e_2) : B}\ (\text{L:Let}) \qquad \frac{A = \langle \tau, q \rangle}{x_1 : \text{arr}(A; B), x_2 : \tau; q \vdash \text{app}(x_1; x_2) : B}\ (\text{L:App})$$

$$\frac{A = \langle \tau, q \rangle \qquad \Gamma = |\Gamma| \qquad \Gamma, f : \text{arr}(A; B), x : \tau; q \vdash e : B}{\Gamma; 0 \vdash \text{fun}(f, x.e) : \langle \text{arr}(A; B), 0 \rangle}\ (\text{L:Fun}) \qquad \frac{\tau \curlyvee (\tau_1, \tau_2) \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : B}{\Gamma, x : \tau; q \vdash \text{share}(x; x_1, x_2.e) : B}\ (\text{L:Share})$$

$$\frac{\Gamma \curlyvee (p \times \Gamma_1, (1-p) \times \Gamma_2) \qquad q = p \cdot q_1 + (1-p) \cdot q_2 \qquad \Gamma_1; q_1 \vdash e_1 : A \qquad \Gamma_2; q_2 \vdash e_2 : A}{\Gamma; q \vdash \text{flip}\{e_1; e_2\}(p) : A}\ (\text{L:Flip})$$

$$\frac{q = p \cdot q_H + (1-p) \cdot q_T}{\cdot; q \vdash \text{prob}\{p\} : \langle \text{prob}\{q_H; q_T\}, 0 \rangle}\ (\text{L:Prob}) \qquad \frac{\Gamma; q + q_H \vdash e_1 : A \qquad \Gamma; q + q_T \vdash e_2 : A}{\Gamma, x : \text{prob}\{q_H; q_T\}; q \vdash \text{flip}_{\mathbb{S}}(x; e_1, e_2) : A}\ (\text{L:FlipS})$$

$$\frac{\Gamma; q \vdash e : \langle \tau', q' \rangle \qquad \tau' <: \tau}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ (\text{L:Sub}) \qquad \frac{\Gamma, x : \tau; q \vdash e : B \qquad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B}\ (\text{L:Sup})$$

$$\frac{\Gamma; q \vdash e : B}{\Gamma, x : \tau; q \vdash e : B}\ (\text{L:Weak}) \qquad \frac{\Gamma; p \vdash e : \langle \tau, p' \rangle \qquad q \geq p \qquad q - q' \geq p - p'}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ (\text{L:Relax})$$

Fig. 5. Typing rules

Example. To illustrate the type system in action, we apply it to a random-walk program in concrete syntax below. Consider the function *brdwalk* which performs a biased random walk over the length of its input list, stopping whenever the list is empty. With $\frac{3}{4}$ probability of shrinking the list and $\frac{1}{4}$ of growing it, we expect the list length to shrink by $\frac{1}{2}$ per iteration of the walk. Thus, we expect a stopping time of twice the input list's length.

```
brdwalk  ≡  fun f ℓ =
              case ℓ {[] ↪ ⟨⟩
                      _ :: x₂ ↪ let _ = tick 1 in flip 3/4 { H ↪ f(x₂) | T ↪ f(⟨⟩ :: ⟨⟩ :: x₂)}}
```

We now derive the type $\langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle$ for *brdwalk*, which indicates using twice the input list's length for initial potential. This amount of potential provides an upper bound on expected stopping time which happens to be exact.

In the *tails* case, we need four units of extra constant potential to construct the argument list.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{f : \langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle, x_2 : L^2(\mathbb{1}); 0 \vdash x_2 : \langle L^2(\mathbb{1}), 0 \rangle}\ (\text{L:Var})}{f : \langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle, x_2 : L^2(\mathbb{1}); 2 \vdash \langle\rangle :: x_2 : \langle L^2(\mathbb{1}), 0 \rangle}\ (\text{L:Cons})}{f : \langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle, x_2 : L^2(\mathbb{1}); 4 \vdash \langle\rangle :: \langle\rangle :: x_2 : \langle L^2(\mathbb{1}), 0 \rangle}\ (\text{L:Cons})}{f : \langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle, x_2 : L^2(\mathbb{1}); 4 \vdash f(\langle\rangle :: \langle\rangle :: x_2) : \langle \mathbb{1}, 0 \rangle}\ (\text{L:App})$$

Otherwise, if the coin flip shows *heads*, the type derivation goes as follows:

$$\frac{}{f : \langle L^2(\mathbb{1}), 0 \rangle \to \langle \mathbb{1}, 0 \rangle, x_2 : L^2(\mathbb{1}); 0 \vdash f(x_2) : \langle \mathbb{1}, 0 \rangle}\ (\text{L:App})$$

$$\boxed{v : \tau \text{ (or } v : A) \quad \text{“value } v \text{ has type } \tau \text{ or } A\text{”}}$$

$$\frac{}{\langle\rangle : \mathsf{unit}}\ (\text{V:Unit}) \qquad \frac{}{\mathsf{prob}(p) : \mathsf{prob}\{q_H; q_T\}}\ (\text{V:Prob}) \qquad \frac{}{[\,] : \mathsf{list}(A)}\ (\text{V:Nil})$$

$$\frac{v_1 : A \qquad v_2 : \mathsf{list}(A)}{v_1 :: v_2 : \mathsf{list}(A)}\ (\text{V:Cons}) \qquad \frac{v : \tau}{v : \mathsf{pot}(\tau; q)}\ (\text{V:Anno}) \qquad \frac{\begin{array}{c} A = \mathsf{pot}(\tau; q) \qquad V : \Gamma \\ |\Gamma|, f : \mathsf{arr}(A; B), x : \tau; q \vdash e : B \end{array}}{\mathsf{clo}(V; f, x.e) : \mathsf{arr}(A; B)}\ (\text{V:Fun})$$

Fig. 6. Typing rules for values

Via the definition of potential scaling, we find that

$$L^2(\mathbb{1}) \curlyvee (3/4 \times L^2(\mathbb{1}), (1 - 3/4) \times L^2(\mathbb{1})) \quad \text{and} \quad 1 = 3/4 \cdot 0 + 1/4 \cdot 4.$$

Then we apply the rule L:Flip, deriving the desired type $\langle L^2(\mathbb{1}), 0\rangle \to \langle \mathbb{1}, 0\rangle$ for *brdwalk*.

$$\frac{\begin{array}{c} f : \langle L^2(\mathbb{1}), 0\rangle \to \langle \mathbb{1}, 0\rangle, x_2 : L^2(\mathbb{1}); 0 \vdash f(x_2) : \langle \mathbb{1}, 0\rangle \\ f : \langle L^2(\mathbb{1}), 0\rangle \to \langle \mathbb{1}, 0\rangle, x_2 : L^2(\mathbb{1}); 4 \vdash f(\langle\rangle :: \langle\rangle :: x_2) : \langle \mathbb{1}, 0\rangle \end{array}}{f : \langle L^2(\mathbb{1}), 0\rangle \to \langle \mathbb{1}, 0\rangle, x_2 : L^2(\mathbb{1}); 1 \vdash \mathsf{flip}\ 3/4\ \{\mathsf{H} \hookrightarrow f(x_2) \mid \mathsf{T} \hookrightarrow f(\langle\rangle :: \langle\rangle :: x_2)\} : \langle \mathbb{1}, 0\rangle}\ (\text{L:Flip})$$

5 SOUNDNESS

In this section, we formalize our intuition that our type system derives expected cost bounds and sketch a soundness proof (Thm. 5.3). We also study nontrivial non-termination behavior of probabilistic programs, and prove a stronger result (Thm. 5.7) which implies that derived expected bounds on resources like time imply that the analyzed program terminates with probability one (Cor. 5.8). Proofs can be found in [Wang et al. 2020b].

Values. Before we can state the theorem, we need to properly extend the definition of potential to typing contexts and evaluation environments. We introduce a type judgment $v : \tau$ (or $v : A$) for values, which is defined in Fig. 6. This relation ignores potential annotations and checks only the values are well-typed. An evaluation environment V is said to have type context Γ, written $V : \Gamma$, if for all x bound in Γ, we have $V(x) : \Gamma(x)$. The most interesting rule is the rule V:Fun for function closures. It uses the type rule L:Fun for expressions and existentially quantifies over the context Γ. This rule ensures that we only consider functions that are well-formed with respect to the type system, which is necessary to prove the soundness of the analysis.

Let $V : \Gamma$. We define the potential of V with respect to Γ as follows.

$$\Phi(V : \Gamma) \stackrel{\text{def}}{=} \sum_{x \in \mathrm{dom}(\Gamma)} \Phi(V(x) : \Gamma(x)).$$

A first attempt. With the trace-based evaluation dynamics, we might state the soundness theorem for probabilistic programs as follows. Intuitively, it says that the initial potential is sufficient to pay for the expected evaluation cost and the typing of the result.

Let $\Gamma; q \vdash e : A$ and $V : \Gamma$. Then

$$\Phi(V : \Gamma) + q \geq \sum_{\sigma_0 : V; \sigma_0 \vdash e \Downarrow^{p_0} v_0 \mid q_0} p_0 \cdot (\Phi(v_0 : A) + q_0).$$

Note that the summation is taken over traces σ_0, and by Lem. 3.1, the tuple (p_0, v_0, q_0) is uniquely determined by V, e, and σ_0. However, it is unclear how to prove the theorem by induction on the evaluation judgment. The reason is that we now have to deal with a collection of evaluation judgments, instead of one. Intuitively, the trace-based evaluation dynamics talks about *individual* executions, while the goal of our resource analysis for probabilistic programs is to reason about *aggregated* information over all possible executions. We therefore develop another evaluation dynamics that deals with *distributions* of executions more directly, and show that it agrees with our previous semantics.

First we illustrate why a naive approach here will not work. One might start with a new judgment $V \vdash e \Rightarrow \mu$ where μ is a distribution over pairs (v, q), v is the evaluation result, and q is the net cost. Then one might use the following rule for composition under probabilistic branching.

$$\frac{V \vdash e_1 \Rightarrow \mu_1 \qquad V \vdash e_2 \Rightarrow \mu_2}{V \vdash \text{flip}\{e_1; e_2\}(p) \Rightarrow p \cdot \mu_1 + (1-p) \cdot \mu_2} \text{(Bad:Flip)}$$

Here, we denote the weighted sum of two distributions μ_1 and μ_2 by $p \cdot \mu_1 + (1-p) \cdot \mu_2$, defined as $\lambda\omega.p \cdot \mu_1(\omega) + (1-p) \cdot \mu_2(\omega)$.

For the leaf cases, such as unit values, one might then introduce the rule where $\delta(\omega) = \lambda\omega'.[\omega = \omega']$ denotes the *point distribution* on ω, and where the *Iverson brackets* $[\cdot]$ are defined by $[\varphi] = 1$ if φ is true and otherwise $[\varphi] = 0$.

$$\frac{}{V \vdash \text{triv} \Rightarrow \delta(\langle\rangle, 0)} \text{(Bad:Triv)}$$

However, the attempt does not work well for *almost-sure* termination, i.e., terminating with probability 1. The issue is that the inductive definition of such a distribution dynamics will fail if there is a non-terminating execution. Consider the following program

$$f \equiv \text{fun } f\_ = \text{flip } \frac{1}{2} \; \{ \mathsf{H} \hookrightarrow \langle\rangle \mid \mathsf{T} \hookrightarrow f(\langle\rangle) \}$$

and suppose that we want to derive an evaluation judgment for $f(\langle\rangle)$. There does not exist a distribution μ such that $V \vdash f(\langle\rangle) \Rightarrow \mu$, because if we try to apply the rules inductively, we will end up with a derivation tree with an infinite depth.

$$\cfrac{\cfrac{\cfrac{}{V \vdash \langle\rangle \Rightarrow \delta(\langle\rangle, 0)} \text{(Bad:Triv)} \qquad \cfrac{\vdots}{V \vdash f(\langle\rangle) \Rightarrow ???} \text{(Bad:App)}}{V \vdash \text{flip } {}^1\!/\!_2 \; \{ \mathsf{H} \hookrightarrow \langle\rangle \mid \mathsf{T} \hookrightarrow f(\langle\rangle) \} \Rightarrow ???} \text{(Bad:Flip)}}{V \vdash f(\langle\rangle) \Rightarrow ???} \text{(Bad:App)}$$

Distribution-based semantics. To cope with possible non-terminating executions, we develop a partial-evaluation-like dynamics equivalent to our trace-based one. Unlike partial evaluation dynamics used in AARA literature to deal with non-termination [Hoffmann and Hofmann 2010a], we *do* care about the evaluation results. For our new dynamics, we adapt the distribution-based semantics of [Borgström et al. 2016], which index judgments by their derivation depth to be able to construct a "complete" semantics from the "partial" ones. To this end, we need only modify the subprobability distributions to be over value-cost pairs, resulting in judgments of the following form:

$$V \vdash e \Rightarrow^n \mu.$$

The meaning is that the expression e reduces to a subprobability distribution with an at-most-n derivation depth. We use *sub*probability distributions, whose probabilities sum to possibly less than one, because there could be terminating executions with a derivation tree whose depth is more than n. Fig. 7 presents the rules for this distribution-based semantics. In addition to the syntax-directed rules, we introduce a special base case where $n = 0$ and μ is set to a zero distribution $\mathbf{0} \overset{\text{def}}{=} \lambda\omega.0$.

We can now approximate the distribution over terminating executions using the depth-indexed distributions by making use of the following lemma.

Lemma 5.1. *If $V \vdash e \Rightarrow^n \mu_1$, $V \vdash e \Rightarrow^m \mu_2$ and $n \leq m$, then $\mu_1 \leq \mu_2$ pointwise. As a consequence, we can define $[\![e]\!]^V_{\Rightarrow} \overset{\text{def}}{=} \sup\{\mu_n : V \vdash e \Rightarrow^n \mu_n\} = \lim_{n\to\infty} \mu_n$ as the subprobability distribution of all possible terminating executions of a probabilistic program e under environment V.*

$$\boxed{V \vdash e \Rightarrow^n \mu \qquad \text{“in environment } V\text{, expression } e \text{ reduces to result distribution } \mu \text{ within } n \text{ steps}}$$

$$\frac{}{V \vdash e \Rightarrow^0 \mathbf{0}}\ (\text{DE:Base}) \qquad \frac{n > 0}{V \vdash x \Rightarrow^n \delta(V(x), 0)}\ (\text{DE:Var}) \qquad \frac{n > 0}{V \vdash \text{triv} \Rightarrow^n \delta(\langle\rangle, 0)}\ (\text{DE:Triv})$$

$$\frac{n > 0}{V \vdash \text{nil} \Rightarrow^n \delta([\,], 0)}\ (\text{DE:Nil}) \qquad \frac{n > 0 \qquad V(x_1) = v_1 \qquad V(x_2) = v_2}{V \vdash \text{cons}(x_1; x_2) \Rightarrow^n \delta(v_1 :: v_2, 0)}\ (\text{DE:Cons})$$

$$\frac{V(x) = [\,] \qquad V \vdash e_0 \Rightarrow^n \mu}{V \vdash \text{mat}_{\text{L}}\{e_0; x_1, x_2.e_1\}(x) \Rightarrow^{n+1} \mu}\ (\text{DE:MatL-1}) \qquad \frac{\begin{array}{c} V(x) = v_1 :: v_2 \\ V, x_1 \mapsto v_1, x_2 \mapsto v_2 \vdash e_1 \Rightarrow^n \mu \end{array}}{V \vdash \text{mat}_{\text{L}}\{e_0; x_1, x_2.e_1\}(x) \Rightarrow^{n+1} \mu}\ (\text{DE:MatL-2})$$

$$\frac{n > 0}{V \vdash \text{tick}\{q\} \Rightarrow^n \delta(\langle\rangle, q)}\ (\text{DE:Tick}) \qquad \frac{n > 0}{V \vdash \text{fun}(f, x.e) \Rightarrow^n \delta(\text{clo}(V; f, x.e), 0)}\ (\text{DE:Fun})$$

$$\frac{V(x_1) = \text{clo}(V'; f, x.e) \qquad V(x_2) = v_2 \qquad V', f \mapsto \text{clo}(V'; f, x.e), x \mapsto v_2 \vdash e \Rightarrow^n \mu}{V \vdash \text{app}(x_1; x_2) \Rightarrow^{n+1} \mu}\ (\text{DE:App})$$

$$\frac{V \vdash e_1 \Rightarrow^n \mu \qquad \forall (v_1, q_1) \in \text{supp}(\mu) : V, x \mapsto v_1 \vdash e_2 \Rightarrow^n \mu_{(v_1, q_1)}}{V \vdash \text{let}(e_1; x.e_2) \Rightarrow^{n+1} \sum_{(v_1, q_1)} \sum_{(v_2, q_2)} \mu(v_1, q_1) \cdot \mu_{(v_1, q_1)}(v_2, q_2) \cdot \delta(v_2, q_1 + q_2)}\ (\text{DE:Let})$$

$$\frac{\begin{array}{c} V(x) = v \\ V, x_1 \mapsto v, x_2 \mapsto v \vdash e \Rightarrow^n \mu \end{array}}{V \vdash \text{share}(x; x_1, x_2.e) \Rightarrow^{n+1} \mu}\ (\text{DE:Share}) \qquad \frac{V \vdash e_1 \Rightarrow^n \mu_1 \qquad V \vdash e_2 \Rightarrow^n \mu_2}{V \vdash \text{flip}\{e_1; e_2\}(p) \Rightarrow^{n+1} p \cdot \mu_1 + (1 - p) \cdot \mu_2}\ (\text{DE:Flip})$$

$$\frac{n > 0}{V \vdash \text{prob}\{p\} \Rightarrow^n \delta(\text{prob}(p), 0)}\ (\text{DE:Prob}) \qquad \frac{V(x) = \text{prob}(p) \qquad V \vdash e_1 \Rightarrow^n \mu_1 \qquad V \vdash e_2 \Rightarrow^n \mu_2}{V \vdash \text{flip}_{\mathbb{S}}(x; e_1, e_2) \Rightarrow^{n+1} p \cdot \mu_1 + (1 - p) \cdot \mu_2}\ (\text{DE:FlipS})$$

Fig. 7. Evaluation rules of the distribution-based cost semantics

Proof. By induction on the derivation of $V \vdash e \Rightarrow^m \mu_2$, followed by inversion on $V \vdash e \Rightarrow^n \mu_1$. The existence of the sequence appeals to the Monotone Convergence Theorem. □

Recall the problem case from attempting a non-indexed distribution-based operational semantics:

$$f \equiv \mathsf{fun}\ f\_ = \mathsf{flip}\ \frac{1}{2}\ \{\mathsf{H} \hookrightarrow \langle\rangle \mid \mathsf{T} \hookrightarrow f(\langle\rangle)\}$$

With the depth-indexed distribution-based dynamics, we can now derive the following judgments:

$$V \vdash f(\langle\rangle) \Rightarrow^0 \mathbf{0}, \qquad V \vdash f(\langle\rangle) \Rightarrow^3 {}^1\!/\!_2 \cdot \delta(\langle\rangle, 0),$$

$$V \vdash f(\langle\rangle) \Rightarrow^5 {}^1\!/\!_2 \cdot \delta(\langle\rangle, 0) + {}^1\!/\!_4 \cdot \delta(\langle\rangle, 0), \quad \cdots, \quad V \vdash f(\langle\rangle) \Rightarrow^{2k+1} \textstyle\sum_{i=1}^{k} ({}^1\!/\!_2)^i \cdot \delta(\langle\rangle, 0).$$

Letting k approach infinity, we derive that $\llbracket f(\langle\rangle) \rrbracket^V_{\Rightarrow} = \delta(\langle\rangle, 0)$, i.e., the program terminates with probability one. Further, the evaluation result is always unit, and the net cost is always zero.

Finally, we show that the distribution-based dynamics is equivalent to the trace-based one, so we can proceed to prove soundness with respect to the distribution-based semantics.

Proposition 5.2. *Let V be an environment and e be an expression. Then $\llbracket e \rrbracket^V_{\Rightarrow} = \llbracket e \rrbracket^V_{\Downarrow}$.*

Proof. We proceed by proving both $\llbracket e \rrbracket^V_{\Rightarrow} \le \llbracket e \rrbracket^V_{\Downarrow}$ and $\llbracket e \rrbracket^V_{\Downarrow} \le \llbracket e \rrbracket^V_{\Rightarrow}$. For the first inequality, it is sufficient to show that $\mu_n \le \llbracket e \rrbracket^V_{\Downarrow}$ for all $n \in \mathbb{N}$ where $V \vdash e \Rightarrow^n \mu_n$. For the second one, it suffices to show that $\nu_n \le \llbracket e \rrbracket^V_{\Rightarrow}$ for all $n \in \mathbb{N}$ where ν_n is a sub-distribution of executions in $\llbracket e \rrbracket^V_{\Downarrow}$ whose trace has length at most n. Both cases are done by induction on n. □

Soundness. We now restate and prove the soundness theorem using the distribution-based semantics. Again, it states that the initial potential can pay for the expected evaluation cost and the typing of the result.

Theorem 5.3 (Soundness of AARA). *Let $\Gamma; q \vdash e : A$ and $V : \Gamma$. Then*

$$\Phi(V : \Gamma) + q \geq \sum_{(v_0,q_0)} [\![e]\!]^{V}_{\Rightarrow}(v_0, q_0) \cdot (\Phi(v_0 : A) + q_0).$$

Proof. It suffices to prove for every $n \in \mathbb{N}$, if $V \vdash e \Rightarrow^n \mu$, then

$$\Phi(V : \Gamma) + q \geq \sum_{(v_0,q_0)} \mu(v_0, q_0) \cdot (\Phi(v_0 : A) + q_0).$$

Proceed by induction on n with inversion on $V \vdash e \Rightarrow^n \mu$ then inner induction on $\Gamma; q \vdash e : A$. □

Non-termination. So far we have only considered terminating executions in the evaluation dynamics, dealing with non-termination indirectly. Recall that the distribution over e's evaluations in environment V is defined as

$$[\![e]\!]^{V}_{\Downarrow}(v, q) \overset{\text{def}}{=} \sum_{\sigma} p_\sigma \quad \text{where } \sigma\text{'s are \textit{finite} traces satisfying } V; \sigma \vdash e \Downarrow^{p_\sigma} v \mid q,$$

thus *infinite* traces (e.g., non-terminating executions) are totally ignored. Hence, the soundness theorem (Thm. 5.3) does *not* imply that the typing judgment $\Gamma; q \vdash e : A$ (where e is instrumented with ticks to count evaluation steps) entails that the expected termination time of e is finite. We therefore now extend the dynamics to account for non-terminating behavior directly.

To deal with non-termination, we first introduce a dummy value $\circ$ to represent some partial evaluation. We can then enrich the distribution-based dynamics with partial evaluation by forcing the result distribution μ in the judgment $V \vdash e \Rightarrow^n \mu$ to be a *full* probability distribution instead of a subprobability one. To achieve this, we extend μ's distributions to be over $(\mathsf{Val} \cup \{\circ\}) \times (\mathbb{Q}_{\geq 0} \cup \{\infty\})$, including this new dummy value. Most of the rules stay unchanged, except the following two:

$$\frac{}{V \vdash e \Rightarrow^0 \delta(\circ, 0)} \text{(PE:Base)}$$

$$\frac{V \vdash e_1 \Rightarrow^n \mu \qquad \forall (v_1, q_1) \in \mathrm{supp}(\mu) \colon (v_1 \neq \circ) \implies V, x \mapsto v_1 \vdash e_2 \Rightarrow^n \mu_{(v_1,q_1)}}{V \vdash \mathrm{let}(e_1; x.e_2) \Rightarrow^{n+1} \sum_{q_1} \mu(\circ, q_1) \cdot \delta(\circ, q_1) + \sum_{(v_1,q_1): v_1 \neq \circ} \sum_{(v_2,q_2)} \mu(v_1, q_1) \cdot \mu_{(v_1,q_1)}(v_2, q_2) \cdot \delta(v_2, q_1 + q_2)} \text{(PE:Let)}$$

However, we can no longer take the previous approach of defining $[\![e]\!]^{V}_{\Rightarrow}$ by the limit of $\{\mu_n\}_{n \in \mathbb{N}}$ where $V \vdash e \Rightarrow^n \mu_n$, because it no longer holds that, if $n \leq m$,then $\mu_n \leq \mu_m$ pointwise. To get around this, we define a new ordering on complete distributions, extending it to cover the dummy value differently. We define $\mu_1 \sqsubseteq \mu_2$ as

- $\forall v, q \colon (v \neq \circ) \implies \mu_1(v, q) \leq \mu_2(v, q)$, and
- $\forall q \colon \mu_1((\mathsf{Val} \cup \{\circ\}) \times [0, q]) \geq \mu_2((\mathsf{Val} \cup \{\circ\}) \times [0, q])$.

For concrete values, the order above is the same as the pointwise order on subprobability distributions, but for divergence, we take the other direction—the property above implies that $\mu_1(\{\circ\} \times [0, q]) \geq \mu_2(\{\circ\} \times [0, q])$ for all $q \in \mathbb{Q}_{\geq 0} \cup \{\infty\}$. Since we assume non-negative ticks, the probability that the cost is smaller than any q with respect to μ_1 should be greater than or equal to that with respect to μ_2. Formally, we prove that $\sqsubseteq$ defines an *ω-complete partial order* on distributions.

Lemma 5.4. *The relation $\sqsubseteq$ defines a partial order on the distributions. Further, let $\{\mu_n\}_{n \in \mathbb{N}}$ be a sequence such that $\mu_1 \sqsubseteq \mu_2 \sqsubseteq \cdots \sqsubseteq \mu_n \sqsubseteq \cdots$. Then there exists a least distribution μ such that for all $n \in \mathbb{N}$, $\mu_n \sqsubseteq \mu$. Further, we denote μ by $\bigsqcup_{n \in \mathbb{N}} \mu_n$.*

We now restate Lem. 5.1 in terms of the partial order $\sqsubseteq$ over distributions.

Lemma 5.5. *If $V \vdash e \Rightarrow^n \mu_1$, $V \vdash e \Rightarrow^m \mu_2$ and $n \leq m$, then $\mu_1 \sqsubseteq \mu_2$ pointwise. As a consequence, we can define $[\![e]\!]^V_{\Rightarrow} \stackrel{\text{def}}{=} \bigsqcup_{n\in\mathbb{N}} \mu_n$ as the distribution of all possible terminating and non-terminating executions of a probabilistic program e under environment V.*

Proof. By induction on the derivation of $V \vdash e \Rightarrow^m \mu_2$, followed by inversion on $V \vdash e \Rightarrow^n \mu_1$. The existence of the sequence appeals to Lem. 5.4. □

Recall that in the soundness proof, we induct on the index n of $V \vdash e \Rightarrow^n \mu$. The reason why this approach works is that the expected cost with respect to μ is *ω-continuous*, i.e., monotone and interchangeable with a limit operator. Although it is unclear whether the continuity still holds for $\sqsubseteq$ or not, we can prove the following weaker result that is sufficient for our soundness proof.

Lemma 5.6. *Let $h(\mu) \stackrel{\text{def}}{=} \sum_q \mu(\circ, q)\cdot q + \sum_{(v,q):v\neq\circ} \mu(v,q)\cdot(\Phi(v:A)+q)$. Let $\{\mu_n\}_{n\in\mathbb{N}}$ be a sequence such that $\mu_1 \sqsubseteq \mu_2 \sqsubseteq \cdots \sqsubseteq \mu_n \sqsubseteq \cdots$. Let $M \in \mathbb{R}_{\geq 0}$. If $h(\mu_n) \leq M$ for all $n \in \mathbb{N}$, then $h(\bigsqcup_{n\in\mathbb{N}} \mu_n) \leq M$.*

Now we can strengthen the soundness theorem to capture both termination and non-termination.

Theorem 5.7 (Soundness of AARA, improved). *Let $\Gamma; q \vdash e : A$ and $V : \Gamma$. Then*

$$\Phi(V:\Gamma) + q \geq \sum_{q_0} [\![e]\!]^V_{\Rightarrow}(\circ, q_0)\cdot q_0 + \sum_{(v_0,q_0):v_0\neq\circ} [\![e]\!]^V_{\Rightarrow}(v_0, q_0)\cdot(\Phi(v_0:A)+q_0).$$

Proof. By Lemmas 5.5 and 5.6 it suffices to prove for every $n \in \mathbb{N}$, if $V \vdash e \Rightarrow^n \mu$, then

$$\Phi(V:\Gamma) + q \geq \sum_{q_0} \mu(\circ, q_0)\cdot q_0 + \sum_{(v_0,q_0)} \mu(v_0, q_0)\cdot(\Phi(v_0:A)+q_0).$$

Again proved by induction on n with inversion on $V \vdash e \Rightarrow^n \mu$, then $\Gamma; q \vdash e : A$ inner induction. □

Corollary 5.8. *Let $\Gamma; q \vdash e : A$ and $V : \Gamma$. If a program e is instrumented with ticks that account for evaluation steps, then e terminates with probability one, i.e., $[\![e]\!]^V_{\Rightarrow}(\circ, q_0) = 0$ for all $q_0 \in \mathbb{Q}_{\geq 0} \cup \{\infty\}$.*

Proof. For all $q_0 \in \mathbb{Q}_{\geq 0}$, the probability $[\![e]\!]^V_{\Rightarrow}(\circ, q_0)$ is zero because if an execution does not terminate, the cost will keep increasing. For the case where $q_0 = \infty$, by Thm. 5.7, $[\![e]\!]^V_{\Rightarrow}(\circ, \infty)\cdot\infty$ is bounded by $\Phi(V:\Gamma) + q < \infty$, thus the probability $[\![e]\!]^V_{\Rightarrow}(\circ, \infty)$ must be zero. □

6 IMPLEMENTATION AND EXAMPLES

In this section we present some non-trivial probabilistic models which our implementation pRaML can handle in the same manner as described in previous sections. We follow up with a collection of experimental benchmarks from typing variants of our examples, and other examples from literature.

For these complex examples, we use our implementation pRaML of the probabilistic AARA type system extended to *multivariate polynomial* potential functions with user-defined data types. While the potential functions supported in linear AARA are already multivariate, as each addend can depend on a different input size, the term *multivariate* in the setting of potential functions refers to each addend depending on *products* of input sizes - and in this case, also products of symbolic probabilities. With user-defined data types, those sizes can also measure the number of particular constructor types. We also include additional support for symbolic probabilities by allowing complementation (i.e., subtraction from 1). Extending the probabilistic type system laid out here to these domains does not involve significant conceptual changes; the potential function extensions - described in [Hoffmann et al. 2011] and [Hoffmann et al. 2017] - are orthogonal to the new probabilistic operation.

Tab. 1 shows some analysis data given by pRaML on models described below and some examples from literature. It displays the number of linear constraints generated by typing the program using resource polynomials at a fixed degree for all programs of the same class, as well as how fast pRaML

```
let rec gr Alice Bob =
  match Alice with
  | [] → ()
  | ha::ta →
    match Bob with
    | [] → ()
    | hb::tb →
      let _ = tick 1 in
      match flip 0.5 with
      | H → gr ta (ha::Bob)
      | T → gr (hb::Alice) tb
```

(a) Gambler's ruin

```
let rec goat below at above =
  let _ = tick 1 in
  match at with
  | Lichen → match flip 0.75 with
    | H → match below with
      | [] → ()
      | hd::tl → goat tl hd (at::above)
    | T → match above with
      | [] → ()
      | hd::tl → goat (at::below) hd tl
  | Grass → match flip 0.5 with
    | H → match below with
      | [] → ()
      | hd::tl → goat tl hd (at::above)
    | T →  match above with
      | [] → ()
      | hd::tl → goat (at::below) hd tl
```

(b) The life expectancy of a goat

Fig. 8. Implementations probabilistic programs in pRaML.

can complete type inference on consumer hardware. The literature examples include some example probabilistic loop code and conditional sampling model [Gordon et al. 2014], the simulation of a fair die with a fair coin using a Markov chain [Knuth and Yao 1976], a probabilistic variant of example code demonstrating quadratic resource usage [Carbonneaux et al. 2017], and the program *miner* [Ngo et al. 2018]. The final example, *fill* and *consume*, fills a list with probability values of $1/2$ or $1/3$ randomly according to a symbolic probability p, then iterates over the list, flipping a coin biased by each probability, and paying cost 1 for each heads flip.

Random walks form the core of stochastic algorithms and simulations. The Internet is so large that the tractability of measuring its contents is real concern, and it can be solved by random walks [Bar-Yossef and Gurevich 2008]. Modeling problems from various fields also use random walks, ranging from economics [Meese and Rogoff 1983], to biology [Codling et al. 2008], to ecology [Visser 1997], to astrophysics [MacLeod et al. 2010], and beyond. However, many random walks are non-trivial to analyze, which obscures properties like code efficiency from a non-expert programmer, and obscures stochastic model properties from their users. Even knowing the bounds of complex random walk first, the bounds can be nontrivial to verify by hand. Nonetheless, AARA can find them quickly, giving non-experts automatic access to expert bounds.

Example 6.1 (Gambler's Ruin). There is an old problem in probability called the *Gambler's Ruin.* Fig. 8a shows an implementation. It is set up so that Alice and Bob continually bet one dollar against each other on the results of a coin-flip until one player runs out of money. This is essentially a 2-sided random walk. If the coin is fair, Alice starts with A dollars and Bob starts with B dollars, then this series of bets is expected to take AB time. Our multivariate implementation finds this bound exactly.

Example 6.2 (The Life Expectancy of a Goat). Consider modeling the following scenario: A mountain goat lives high up in the Rocky Mountains, eating grasses and lichens from the rocks. Depending on the food it find abundant, it either moves up or down the mountain. When it finds only lichens, it moves down with probability 75% in an attempt to find better food sources. When it finds grasses, it moves with equal probability in either direction. However, if the goat moves too far down the mountain, it passes the treeline and gets hunted by wolves. On the other hand, if the goat tries to go up the mountain when at the very top, it falls off a cliff. Given some distribution of

Table 1. Experimental data of typing with pRaML.

Program description	Bound	#Constraints	Time (in sec.)
goat with $\frac{1}{2}, \frac{3}{4}$	$(B+1)(2(G+1)-G_B)$	2084	0.15
goat with $\frac{2}{3}, \frac{3}{4}$	$3B+3$	2084	0.14
goat with $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}$	$(B+1)(2(G+1.5)-G_B)$	5336	0.25
goat with $\frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}$	$(B+1)(2(G+2.5)-G_B)$	10996	1.95
trade with $\frac{3}{5}, \frac{1}{3}$	$\frac{1}{15}T^2 + \frac{1}{3}TP + \frac{4}{15}T$	157	0.04
trade with $\frac{3}{5}, 1$	$\frac{1}{5}T^2 + TP + \frac{4}{5}T$	157	0.03
trade with $\frac{2}{5}, 1$	$\frac{3}{10}T^2 + TP + \frac{7}{10}T$	157	0.03
trade with $\frac{2}{5}, \frac{1}{3}$	$\frac{1}{10}T^2 + \frac{1}{3}TP + \frac{7}{30}T$	157	0.04
probabilistic loop Ex 3 [Gordon et al. 2014]	$^4/_3$ probability	61	0.01
bayes sampling Ex 6 [Gordon et al. 2014]	$^3/_5$ probability	112	0.01
die simulation from coin [Knuth and Yao 1976]	$^1/_6$ per die face	5731	0.33
random no-op *nested* variant [Carbonneaux et al. 2017]	$M^2 + M$	205	0.03
miner from [Ngo et al. 2018]	$^{15}/_2 M$	31	0.01
fill and *consume*	$(\frac{1}{3} + \frac{p}{6})M$	633	0.11

grasses and lichens on the mountain, and where the goat starts, what is the expected lifetime of the goat?

This is nontrivial to analyze by hand, but easy to code with the function *goat* in Fig. 8b. Then pRaML can find a cost bound. Letting B be the distance from the goat to the treeline below, G_A be the number of grassy areas above the goat, and G be the total number of grassy areas, the expected lifetime is bounded above by $(B+1)(2(G+1)-G_B)$. This bound is rather complex, but its generality reveals some interesting cost dependencies. For instance, the derived bound is independent of the actual distance to the top of the mountain. It also makes it easy to get a sense of cost behaviour for particular cases: If the whole mountain is covered in lichen, then the expected lifespan is $2(B+1)$, in line with the goat's expected movement of half-a-space down the mountain per iteration. On the other hand, if the mountain is all grassy, then the lifetime more like the stopping time of the Gambler's Ruin experiment.

Tab. 1 lists the analysis data for many different movement probabilities for varying amounts of plants. There we also use A to represent the distance to the top of the mountain.

Example 6.3 (Stock Buying). Stock prices may behave like a random walk. In Fig. 9c we simulate a buyer occasionally buying some stock over time, similarly to [Ngo et al. 2018]. Analysis with pRaML finds that the expected expenditure is $\frac{1}{15}T^2 + \frac{1}{3}TP + \frac{4}{15}T$, where T is the time span and P is the starting stock price. Results for other parameters for the price's walk and buy rate, respectively, may be found under *trade* in Tab. 1.

```
let reprice price =
  match flip 0.6 with
  | H →
    match price with
    | [] → []
    | _::t → t
  | T → ()::price
```

(a)

```
let rec buy price =
  match price with
  | [] → ()
  | _::t →
    let _ = tick 1 in
    buy t
```

(b)

```
let rec trade price time =
  match time with
  | [] → ()
  | _::t →
    let () = match flip 1/3 with
      | H → buy price
      | T → ()
    in
    trade (reprice price) t
```

(c)

Fig. 9. Stock buying

7 APPLICATIONS

In this section, we discuss two application domains of pRaML: analysis of discrete distributions (§7.1) and estimation of average-case cost (§7.2).

7.1 Analysis of Discrete Distributions

Although the only probabilistic fragment introduced by our programming language is probabilistic branching, we are able to implement a broad suite of discrete probability distributions and analyze their properties in our system. In this section, we demonstrate how our tool can be used to not only verify that a program implements the desired distribution, but also analyze *sample complexity* of the program, i.e., the expected number of flips consumed by the program to obtain a sample. Sampling from probability distributions is a fundamental activity in many fields, e.g., Bayesian inference on probabilistic programs [Goodman and Stuhlmüller 2014; Wingate and Weber 2013], and the efficiency of sampling algorithms becomes increasingly important because Monte Carlo methods for probabilistic inference have a trend of requiring billions of random samples per second [Djuric 2019]. Our work provides an approach for understanding sample complexity of discrete distributions.

Case study: Discrete distribution generating (DDG) trees. Recent work provides a universal representation of sampling algorithms for finite supports as *discrete distribution generating* (DDG) binary trees [Saad et al. 2020]. The idea is to implement discrete distributions by only *fair* coin flips. Given a DDG binary tree T, the sample algorithm starts at the root of T, then repeatedly flips a fair coin, takes the left (resp., right) branch if the coin shows heads (resp., tails) until it reaches a leaf node labeled with an outcome from the support of the distribution. Note the tree T may contain back edges, i.e., the algorithm goes back to an ancestor after taking a branch of the current non-leaf node. Back-edges are crucial for implementing non-dyadic probabilities, and they make the running time of the sampling algorithm nontrivial because the algorithm can have non-terminating executions.

Fig. 10 presents two sample algorithms modified from an example in prior work [Saad et al. 2020]. Both programs are supposed to implement a distribution over $\{Red, Black\}$, and return *Red* with probability 0.3, otherwise return *Black*. First, we verify that both programs correctly implement the target distribution. We achieve this by inserting ticks such that the program has one unit of cost when returning *Red*. Our tool then derives that the expected cost for both programs is bounded by 0.3 from above. Meanwhile, we insert ticks in original programs where the program returns *Black* instead of *Red*, and our tool infers that the expected cost for both programs is at most 0.7. Because the expectation of an indicator function for an event E equals to the probability of E, i.e.,

$$\mathbb{E}(\lambda\omega.[E(\omega) \text{ is true}]) = \sum_{\omega} \mathbb{P}(\omega) \cdot [E(\omega) \text{ is true}] = \sum_{\omega \in E} \mathbb{P}(\omega) = \mathbb{P}(E),$$

we conclude that $\mathbb{P}(\text{result is } Red) \le 0.3$ and $\mathbb{P}(\text{result is } Black) \le 0.7$, thus the programs implement the desired distribution, by the fact that probabilities sum up to one.

Then, we study the expected performance of the two sample algorithms in Figs. 10a and 10b. We instrument the two programs with ticks to count the number of probabilistic choices made during the execution. Our expected cost analysis successfully derives an upper bound for both programs: 2.0 for Fig. 10a, and 4.2 for Fig. 10b. By a manual analysis, we also verify that these bounds are tight. The result suggests that Fig. 10a is better than Fig. 10b. We leave automatic tightness checking (e.g., by integrating a lower-bound analysis [Wang et al. 2020a, 2019]) for future work.

Case study: Negative binomial distributions. Beyond distributions with fixed, finite supports, our system is also capable of analyzing discrete distributions with infinite supports and symbolic probabilities. Fig. 10c gives an implementation of negative binomial distributions; it returns a unit list whose length is the number of heads in a series of independent coin flips with probability p

```
let sample_fast () =
  let rec aux () =
    let _ = tick 1 in
    match flip 0.5 with
    | H →let _ = tick 1 in
      match flip 0.5 with
      | H →let _ = tick 1 in
        match flip 0.5 with
        | H →let _ = tick 1 in
          match flip 0.5 with
          | H → aux ()
          | T → Red
        | T → Black
      | T → Black
    | T → Red
  in
  let _ = tick 1 in
  match flip 0.5 with
  | H → aux ()
  | T → Black
```

(a)

```
let rec sample_slow () =
  let _ = tick 1 in
  match flip 0.5 with
  | H →let _ = tick 1 in
    match flip 0.5 with
    | H →let _ = tick 1 in
      sample_slow ()
    | T →let _ = tick 1 in
      match flip 0.5 with
      | H →
        sample_slow ()
      | T →let _ = tick 1 in
        match flip 0.5 with
        | H → Red
        | T → Black
  | T →let _ = tick 1 in
    match flip 0.5 with
    | H →let _ = tick 1 in
      match flip 0.5 with
      | H → Black
      | T → Red
    | T → Black
```

(b)

```
let rec negative_binomial p l =
  match l with
  | [] → []
  | _::l' →
    let _ =
      consume p : prob{0}{1}
    in
    match flip p with
    | H →
      ()::(negative_binomial p l)
    | T →
      negative_binomial p l'
```

(c)

Fig. 10. (a) and (b) are samplers that return *Red* with probability 0.3 or *Black* with probability 0.7. (c) is a program for negative binomial distributions.

before $|\ell|$ number of tails occurs. The *consume* expression is used to specify value-*dependent* costs, which we explain later.

In this example, we want to study the program's sample complexity with respect to p and ℓ. At first glance, the task seems impossible for our system, because while our AARA-based approach is able to derive multivariate-polynomial bound, the expected number of flips for negative binomial distributions involves fractions like $\frac{1}{1-p}$, which is not expressible in our system. Nevertheless, we come up with a workaround that scales all the costs in program by a factor of $(1-p)$ to get rid of the resource bound's denominator. This is achieved by the *consume* expression. Intuitively, consume $x : \tau$ specifies a cost that equals to the potential of the value of x with respect to τ. Recall that $\Phi(p : \text{prob}\{q_H; q_T\}) \overset{\text{def}}{=} p \cdot q_H + (1-p) \cdot q_T$; thus, the *consume* expression in the program introduces a cost of $(1-p)$. Our type system succeeds in finding a linear bound $|\ell|$ on the expected number of flips. Taking the scale factor into account, we conclude that the expected sample complexity for negative binomial distributions is at most $\frac{|\ell|}{1-p}$.

More examples. A summary of all the case studies in the analysis of discrete distributions carried out in our system can be found in Tab. 2 . All the analyses were processed in around one second. The fractional bounds are derived using the scaling technique mentioned above. For distributions *dist* with integer supports, we also create a variant $dist_{\mathbb{E}}$ that specifies the value of the output sample as the cost. For such a case, our tool essentially performs a *first-moment* analysis that computes the *mean* value of the distributions.

7.2 Estimation of Average Case Cost

Understanding resource requirements of computer programs is important for software engineering. Much of the research has been focused on analyzing *worst-case* resource usage and generating an input that exhibits the *worst-case* performance, e.g. [Noller et al. 2018; Wang and Hoffmann 2019]. However, in practice, software performance can be sensitive to the *distribution* of the actual inputs.

Table 2. Examples for sample-complexity or first-moment analysis of discrete distributions. In the bounds, p is the value of the first probability argument, n is the length of the first list argument, and p_i's are the probability-valued elements in the first list argument.

Function	Description	Inferred Bound
$\mathit{sample\_fast} : \mathbb{1} \to \text{red\_or\_black}$	Fig. 10a	2.00
$\mathit{sample\_slow} : \mathbb{1} \to \text{red\_or\_black}$	Fig. 10b	4.20
$\mathit{dice} : \mathbb{1} \to \text{dice}$	A fair dice	3.67
$\mathit{von\_neumann} : \mathbb{P} \to \text{bool}$	Make a fair coin from a biased one	$\frac{1}{p(1-p)}$
$\mathit{binomial} : \mathbb{P} \to L(\mathbb{1}) \to L(\mathbb{1})$	Binomial distribution	n
$\mathit{binomial}_{\mathbb{E}} : \mathbb{P} \to L(\mathbb{1}) \to L(\mathbb{1})$	Binomial distribution; output as cost	$p \cdot n$
$\mathit{geometric} : \mathbb{P} \to L(\mathbb{1})$	Geometric distribution	$\frac{1}{p}$
$\mathit{geometric}_{\mathbb{E}} : \mathbb{P} \to L(\mathbb{1})$	Geometric distribution; output as cost	$\frac{1-p}{p}$
$\mathit{poisson\_binomial} : L(\mathbb{P}) \to L(\mathbb{1})$	Poisson binomial distribution	n
$\mathit{poisson\_binomial}_{\mathbb{E}} : L(\mathbb{P}) \to L(\mathbb{1})$	Poisson binomial distribution; output as cost	$\sum_{1 \le i \le n} p_i$
$\mathit{negative\_binomial} : \mathbb{P} \to L(\mathbb{1}) \to L(\mathbb{1})$	Negative binomial distribution	$\frac{n}{1-p}$
$\mathit{negative\_binomial}_{\mathbb{E}} : \mathbb{P} \to L(\mathbb{1}) \to L(\mathbb{1})$	Negative binomial distribution; output as cost	$\frac{p \cdot n}{1-p}$

For example, although quicksort has a worst-case quadratic time complexity, it usually outperforms many other sorting algorithms (e.g., insertion sort) on randomly generated inputs. Understanding the *performance distribution* induced by the real-world *input distribution* can then help carry out important tasks in software development such as performance evaluation and algorithm selection. In this section, we illustrate how our tool can be used to characterize performance distributions of deterministic programs by their *average-case* resource usage, through a combination with *profiling* techniques.

Program tranformation. Profiling techniques, such as *edge* profiling and *path* profiling, have been used for speculative optimization (especially of branch conditions) [Da Silva and Steffan 2006; Ramalingam 1996], symbolic execution [Filieri et al. 2013, 2014], and performance analysis [Chen et al. 2016]. The idea is to approximate a deterministic branch condition as a probabilistic choice, whose probability is determined by counting *frequencies* of the two branches executed by a program on a collection of real-world inputs. For example, if the then-branch e_1 of the expression if x then e_1 else e_2 is executed 90% of the time, then we transform the conditional with a probabilistic choice flip $0.9 \{ \mathsf{H} \hookrightarrow e_1 \mid \mathsf{T} \hookrightarrow e_2 \}$. Benefits of such profiling-based program transformation are: (i) it does not require complicated analyses to account for the conditional probability of branches, (ii) it provides insights how the input distribution influences the control-flow of a program via an empirical probabilistic model, and (iii) it can accrue profiling information from samples with small sizes but still generalize its average-case cost bounds to inputs with large sizes.

We have implemented an interpreter for the deterministic fragment of our programming language, which executes programs with concrete inputs and collects profiling information including frequencies of control-flow transitions. We then use the profiling information to transform branch conditions to proper probabilistic choices. Note that we have also implemented a statistical independence test to ensure that branch probabilities are constants, rather than dependent on structural features (e.g., lengths of lists) of the input. Then we pass the transformed program to our type-based expected cost analysis to obtain a symbolic bound as the average-case estimation for the cost of the original program.

Case study: Sorting nearly-sorted lists. It is known that comparison-based sorting algorithms cannot beat the $\Theta(n \log n)$ time complexity for input lists of length n. However, if the sorting

```
let rec insert x l =
  match l with
  | [] → [x]
  | y::ys →
    let _ = tick 1.0 in
    match (x > y) with
    | true → y::(insert x ys)
    | false → x::y::ys

let rec isort l =
  match l with
  | [] → []
  | x::xs →
    insert x (isort xs)
```

(a) Original

```
let rec insert' x l =
  match l with
  | [] → [x]
  | y::ys →
    let _ = tick 1.0 in
    match flip 0.9 with
    | H → y::(insert' x ys)
    | T → x::y::ys

let rec isort' l =
  match l with
  | [] → []
  | x::xs →
    insert' x (isort' xs)
```

(b) Transformed

Fig. 11. Average-case cost estimation for insertion sort on nearly-sorted inputs

function is intended to process *nearly-sorted* data—where every element may *on average* be misplaced by at most some constant number k of positions from the correct sorted order—then some sorting algorithms, e.g., *insertion sort*, can achieve linear time complexity. Fig. 11a presents an implementation for insertion sort that uses ticks to count the number of comparisons. Our tool derives that the worst-case cost bound for *isort*(ℓ) is $\binom{|\ell|}{2}$, which is quadratic in the length $|\ell|$ of the list ℓ. The only conditional expression occurs when the *insert* function compares the inserted element x and the head y of the sorted list ℓ, and it recurses on the tail of ℓ if $x > y$. Since an element may be misplaced by k positions on average, intuitively, there should on average be k recursions when inserting an element, which means that the condition $x > y$ evaluates to true with a constant probability $\frac{k-1}{k}$.

In our experiments, our tool managed to detect from a set of nearly-sorted lists that the conditional expression in *insert* can be approximated by a probabilistic choice with a constant probability. Fig. 11b illustrates one case where the branching probability is about 0.9. Our tool derives that the expected cost bound for *isort'*(ℓ) is $10 \cdot |\ell|$, which is linear in the length $|\ell|$ of the list ℓ. The linear bound also reflects that the list ℓ should be nearly sorted, in the sense that every element in ℓ, on average, is misplaced by 10 positions from the correct sorted order.

Case study: Short-circuit Boolean interpretation. When implementing a compiler, one usually must decide how to interpret Boolean expressions. Most commonly, the decision is made to *short-circuit* the *and* and *or* connectives. That is, if the first term determines the whole expression - *false* for *and* or *true* for *or* - then one skips evaluation of the second. Programmatically, this can be implemented with conditionals as in the following code for *interpret* in Fig. 12a.

In the worst case, the code in Fig. 12a must iterate over every node of its input Boolean expression tree, which is exactly the non-probabilistic bound given by our tool. Specifically, letting C be the number of constants, B the number of binary connectives, and N the number of negations, the bound is $C + B + N$. This is the same cost bound as naively evaluating every sub-expression, so it is unclear what value short-circuiting provides. However, if the Boolean constants used are uniformly random, one finds that the branching probability at each conditional can be approximated by a constant: about 0.5 probability for each branch. Converting the code into *interpret'*, our tool now finds a better expected cost bound of $1.5B + N$. Because C is always equal to $B + 1$, this is a strictly better cost bound.

Case study: Sequential insertions in a hash table. We implement a program in our language to model the hash table function from prior work on worst-case analyisis [Noller et al. 2018; Wang

```
let rec interpret exp =
  let _ = tick 1 in
  match exp with
  | True → true
  | False → false
  | Or (a,b) →
      match interpret a with
      | true → true
      | false → interpret b
  | And (a,b) →
      match interpret a with
      | true → interpret b
      | false → false
  | Neg a → not (interpret a)
```

(a) Original

```
let rec interpret' exp =
  let _ = tick 1 in
  match exp with
  | True → true
  | False → false
  | Or (a,b) →
      let _ = interpret' a in
      match flip 0.5 with
      | H → true
      | T → interpret' b
  | And (a,b) →
      let _ = interpet' a in
      match flip 0.5 with
      | H → interpret' b
      | T → false
  | Neg a → not (interpret' a)
```

(b) Transformed

Fig. 12. Average-case cost estimation for short-circuiting Boolean interpretation across uniform inputs

and Hoffmann 2019]. This is a complicated program where each key in the hash table is a string of length 8 and the hash function is DJBX33A from a PHP implementation. The resource model is defined as the number of hash collisions. In the worst case, our system derives that the number of collisions is bounded by $\binom{n}{2}$ where n is the number of insertions. The worst-case quadratic bound makes sense because one may construct a list of different strings with the same hash key. However, if the hash table is used in a setting where security vulnerabilities like Denial-of-Service are not crucial and the inputs are sufficiently random, then the quadratic bound is not meaningful because it is usually assumed that an insertion into a hash table takes constant time.

In our experiments, from a set of randomly generated strings, our tool found out that both the probability that two input strings have the same hash key—and the probability that two input strings with the same hash key are different—are small constants. Our tool then derives $0.11 \cdot n$ as an expected cost bound for the transformed hash-table program with n insertions, which indicates that the number of hash collisions should be linear in the number of insertions in practice.

8 RELATED WORK

We discuss the most-closely related work on expected cost analysis of probabilistic programs in §1. Other related work includes cost analysis of deterministic programs and other (type-based) analyses of probabilistic programs.

Cost analysis for deterministic programs. Automatic and semiautomatic resource bound analysis for deterministic programs has been extensively studied. Our work is based on AARA, which was initially introduced [Hofmann and Jost 2003] to automatically derive linear heap-space bounds for first-order functional programs. AARA has been extended to polynomial bounds [Hoffmann et al. 2011; Hoffmann and Hofmann 2010b; Hofmann and Moser 2015], exponential bounds [Kahn and Hoffmann 2020], logarithmic bounds [Hofmann and Moser 2018], higher-order functions [Hoffmann et al. 2017; Jost et al. 2010], user-defined datatypes [Hoffmann et al. 2017; Jost et al. 2009], and separation logic [Atkey 2010]. The technique has also been generalized to imperative arithmetic programs [Carbonneaux et al. 2017, 2015], as well as integrated into formal proof assistants [Charguéraud and Pottier 2015; Nipkow 2015].

Beyond AARA, there have been many other approaches to formal resource analysis of deterministic programs. Some approaches, similarly to AARA, do so via a type system, including sized types [Vasconcelos 2008] refinement types [Çiçek et al. 2017, 2015; Knoth et al. 2019; Radicek et al.

2018; Wang et al. 2017; Xi 2002], linear dependent types [Dal Lago and Gaboardi 2011; Dal Lago and Petit 2013], and annotated type systems [Crary and Weirich 2000; Danielsson 2008]. Such a type-based approach usually involves constraint solving, some notion of linearity, and high composability, like AARA. However, there is often a tradeoff in programmer burden, like requiring more user annotation for better results. There are also non-typed-based approaches, recurrence solving [Albert et al. 2009, 2015; Danner et al. 2015; Flores-Montoya and Hähnle 2014; Kavvos et al. 2020; Kincaid et al. 2017], abstract interpretation [Blanc et al. 2010; Gulwani 2009; Sinn et al. 2014; Zuleger et al. 2011], term-rewriting techniques [Avanzini and Moser 2013; Brockschmidt et al. 2014; Frohn et al. 2016; Noschinski et al. 2013], defunctionalization [Avanzini et al. 2015], and symbolic execution [Burnim et al. 2009; Noller et al. 2018]. These approaches vary more wildly from the system used in this work.

Despite the number of such approaches to resource analysis, exceedingly few have been adapted to the probabilistic domain, and even less automated. The work in this article represents the first such automated system for probabilistic functional programs. Imperative probabilistic programs have already enjoyed such automated resource analysis in prior work, first established through imperative AARA techniques [Ngo et al. 2018].

Type-base analysis for probabilistic programs. Other properties of probabilistic programs, aside from expected cost, can be analyzed by type-based approaches. Almost-sure termination of functional probabilistic programs can be reasoned about through the dependent type systems of of Dal Lago et al. [Dal Lago and Ghyselen 2018; Dal Lago and Grellois 2019]. Bhat et al. [Bhat et al. 2012, 2013] develop a type system to check absolute continuity of probabilistic first-order let-programs and derive corresponding density functions for the distributions specified by the programs. Fuzz [Reed and Pierce 2010] uses linear types augmented with a probability monad to reason about differential privacy of randomized computation, and DFuzz [Gaboardi et al. 2013] later generalizes it with indexed types and lightweight dependent types to certify differential privacy for a broader class of benchmarks. Recently, Lew et al. [Lew et al. 2020] have developed a type system for programmable probabilistic inference with trace types, where well-typed inference programs soundly derive posterior distributions by construction. In this paper, we focus on expected cost bound analysis for probabilistic programs.

9 CONCLUSION

By combining a carefully developed probabilistic semantics with the AARA type system, we have shown that probabilistic programs in a functional language can be effectively analyzed in an automated manner. Our implementation pRaML infers worst-case expected bounds on resource usage for a variety of probabilistic models and algorithms, and parameterizes the bounds by both input sizes and symbolic probabilities. We make use of these parameterized bounds to analyze new and interesting application domains, like sample-complexity and a generalized average-case analysis. In the future, we hope to overcome the semantic soundness obstacles that bar non-monotone resource usage, and in doing so provide a fully-conservative extension of non-probabilistic AARA.

ACKNOWLEDGMENTS

This article is based on research supported by DARPA under AA Contract FA8750-18-C-0092 and by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514, and SHF Awards 1812876 and 2007784. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

REFERENCES

E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. 2009. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comp. Sci.* 258 (December 2009). Issue 1.

E. Albert, J. C. Fernández, and G. Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'15).*

R. Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *European Symp. on Programming (ESOP'10).*

M. Avanzini, U. Dal Lago, and A. Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Logic in Computer Science (LICS'19).*

M. Avanzini, U. Dal Lago, and G. Moser. 2015. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Int. Conf. on Functional Programming (ICFP'15).*

M. Avanzini and G. Moser. 2013. A Combination Framework for Complexity. In *Int. Conf. on Rewriting Techniques and Applications (RTA'13).*

Ziv Bar-Yossef and Maxim Gurevich. 2008. Random sampling from a search engine's index. *Journal of the ACM (JACM)* 55, 5 (2008), 1–74.

G. Barthe, B. Grégoire, and S. Zanella Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Princ. of Prog. Lang. (POPL'09).*

G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Princ. of Prog. Lang. (POPL'12).*

Kevin Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2018. How long, O Bayesian network, will I sample thee?. In *European Symp. on Programming (ESOP'18).*

S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. 2012. A Type Theory for Probability Density Functions. In *Princ. of Prog. Lang. (POPL'12).*

S. Bhat, J. Borgström, A. D. Gordon, and C. Russo. 2013. Deriving probability density functions from probabilistic functional programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'13).*

P. Billingsley. 2012. *Probability and Measure.* John Wiley & Sons, Inc.

R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning (LPAR'10).*

J. Borgström, U. Dal Lago, A. D. Gordon, and M. Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Int. Conf. on Functional Programming (ICFP'16).*

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'14).*

J. Burnim, S. Juvekar, and K. Sen. 2009. WISE: Automated Test Generation for Worst-case Complexity. In *Int. Conf. on Softw. Eng. (ICSE'09).*

Q. Carbonneaux, J. Hoffmann, T. Reps, and Z. Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verif. (CAV'17).*

Q. Carbonneaux, J. Hoffmann, and Z. Shao. 2015. Compositional Certified Resource Bounds. In *Prog. Lang. Design and Impl. (PLDI'15).*

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. 2017. Stan: A Probabilistic Programming Language. *J. Statistical Softw.* 76 (2017). Issue 1.

A. Charguéraud and F. Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP'15).*

K. Chatterjee, H. Fu, and A. K. Goharshady. 2016a. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verif. (CAV'16).*

K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. 2016b. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. In *Princ. of Prog. Lang. (POPL'16).*

B. Chen, Y. Liu, and W. Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *Int. Conf. on Softw. Eng. (ICSE'16).*

E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. 2017. Relational Cost Analysis. In *Princ. of Prog. Lang. (POPL'17).*

E. Çiçek, D. Garg, and U. A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *European Symp. on Programming (ESOP'15).*

Edward A Codling, Michael J Plank, and Simon Benhamou. 2008. Random walk models in biology. *Journal of the Royal Society Interface* 5, 25 (2008), 813–834.

K. Crary and S. Weirich. 2000. Resource Bound Certification. In *Princ. of Prog. Lang. (POPL'00).*

J. Da Silva and J. G. Steffan. 2006. A Probabilistic Pointer Analysis for Speculative Optimizations. In *Architectural Support for Prog. Lang. and Op. Syst. (ASPLOS'06).*

U. Dal Lago and M. Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Logic in Computer Science (LICS'11).*

U. Dal Lago and A. Ghyselen. 2018. On Linear Dependent Types and Probabilistic Termination. In *International Workshop on Developments in Implicit Computational Complexity*.

U. Dal Lago and C. Grellois. 2019. Probabilistic Termination by Monadic Affine Sized Typing. *Trans. on Prog. Lang. and Syst.* 41 (June 2019). Issue 2.

U. Dal Lago and B. Petit. 2013. The Geometry of Types. In *Princ. of Prog. Lang. (POPL'13)*.

N. A. Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Princ. of Prog. Lang. (POPL'08)*.

N. Danner, D. R. Licata, and R. Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Int. Conf. on Functional Programming (ICFP'15)*.

D. Djuric. 2019. Billions of Random Numbers in a Blink of an Eye. Available on https://dragan.rocks/articles/19/Billion-random-numbers-blink-eye-Clojure.

A. Filieri, C. S. Păsăreanu, and W. Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *Int. Conf. on Softw. Eng. (ICSE'13)*.

A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. 2014. Statistical Symbolic Execution with Informed Sampling. In *Found. of Softw. Eng. (FSE'14)*.

A. Flores-Montoya and R. Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Asian Symp. on Prog. Lang. and Systems (APLAS'14)*.

F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. 2016. Lower Runtime Bounds for Integer Programs. In *Int. Joint Conf. on Automated Reasoning (IJCAR'16)*.

M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Princ. of Prog. Lang. (POPL'13)*.

N. D. Goodman and A. Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. Available on http://dippl.org.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. 167–181.

S. Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verif. (CAV'09)*.

M. Hark, B. L. Kaminski, J. Giesl, and J.-P. Katoen. 2020. Aiming Low Is Harder: Induction for Lower Bounds in Probabilistic Program Verification. In *Princ. of Prog. Lang. (POPL'20)*.

R. Harper. 2016. *Practical Foundations for Programming Languages*. Cambridge University Press.

J. Hoffmann, K. Aehlig, and M. Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Princ. of Prog. Lang. (POPL'11)*.

J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Princ. of Prog. Lang. (POPL'17)*.

J. Hoffmann and M. Hofmann. 2010a. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Asian Symp. on Prog. Lang. and Systems (APLAS'10)*.

J. Hoffmann and M. Hofmann. 2010b. Amortized Resource Analysis with Polynomial Potential. In *European Symp. on Programming (ESOP'10)*.

M. Hofmann and S. Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Princ. of Prog. Lang. (POPL'03)*.

M. Hofmann and G. Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *Int. Conf. on Typed Lambda Calculi and Applications (TLCA'15)*.

M. Hofmann and G. Moser. 2018. *Analysis of Logarithmic Amortised Complexity*. Technical Report. Computing Research Repository.

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *Princ. of Prog. Lang. (POPL'10)*.

S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. 2009. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *Symp. on Form. Meth. (FM'09)*.

D. M. Kahn and J. Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'20)*.

B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. 2016. Weakest Precondition Reasoning for Expected Run—Times of Probabilistic Programs. In *European Symp. on Programming (ESOP'16)*.

G. A. Kavvos, E. Morehouse, D. R. Licata, and N. Danner. 2020. Recurrence Extraction for Functional Programs through Call-by-Push-Value. In *Princ. of Prog. Lang. (POPL'20)*.

Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps. 2017. Compositional Recurrence Analysis Revisited. In *Prog. Lang. Design and Impl. (PLDI'17)*.

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *Prog. Lang. Design and Impl. (PLDI'19)*.

Donald Knuth and Andrew Yao. 1976. Algorithms and Complexity: New Directions and Recent Results, chapter The complexity of nonuniform random number generation.

D. Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22 (June 1981). Issue 3.

S. Kura, N. Urabe, and I. Hasuo. 2019. Tail Probability for Randomized Program Runtimes via Martingales for Higher Moments. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'19)*.

A. K. Lew, M. F. Cusumano-Towner, B. Sherman, M. Carbin, and V. K. Mansinghka. 2020. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. In *Princ. of Prog. Lang. (POPL'20)*.

Ch L MacLeod, Ž Ivezić, CS Kochanek, S Kozłowski, B Kelly, E Bullock, A Kimball, B Sesar, D Westman, K Brooks, et al. 2010. Modeling the time variability of SDSS stripe 82 quasars as a damped random walk. *The Astrophysical Journal* 721, 2 (2010), 1014.

V. K. Mansinghka, U. Schaechtle, S. Handa, A. Radul, Y. Chen, and M. C. Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Prog. Lang. Design and Impl. (PLDI'18)*.

A. K. McIver and C. C. Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems.* Springer Science+Business Media, Inc.

Richard A Meese and Kenneth Rogoff. 1983. Empirical exchange rate models of the seventies: Do they fit out of sample? *Journal of international economics* 14, 1-2 (1983), 3–24.

V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Prog. Lang. Design and Impl. (PLDI'18)*.

V. C. Ngo, Mario Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Symp. on Sec. and Privacy (SP'17)*.

T. Nipkow. 2015. Amortized Complexity Verified. In *Interactive Theorem Proving (ITP'15)*.

Y. Noller, R. Kersten, and C. S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Int. Symp. on Softw. Testing and Analysis (ISSTA'18)*.

L. Noschinski, F. Emmes, and J. Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Automated Reasoning* 51 (June 2013). Issue 1.

F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Logic in Computer Science (LICS'16)*.

G. D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5 (1977), 223–255.

I. Radicek, G. Barthe, M. Gaboardi, D. Garg, and F. Zuleger. 2018. Monadic Refinements for Relational Cost Analysis. In *Princ. of Prog. Lang. (POPL'18)*.

G. Ramalingam. 1996. Data Flow Frequency Analysis. In *Prog. Lang. Design and Impl. (PLDI'96)*.

J. Reed and B. C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Int. Conf. on Functional Programming (ICFP'10)*.

F. A. Saad, C. E. Freer, M. C. Rinard, and V. K. Mansinghka. 2020. Optimal Approximate Sampling from Discrete Probability Distributions. In *Princ. of Prog. Lang. (POPL'20)*.

M. Sinn, F. Zuleger, and H. Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verif. (CAV'14)*.

R. E. Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6 (August 1985). Issue 2.

Joseph Tassarotti and Robert Harper. 2019. A Separation Logic for Concurrent Randomized Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 64 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290377

P. B. Vasconcelos. 2008. *Space Cost Analysis Using Sized Types.* Ph.D. Dissertation. School of Computer Science, University of St Andrews.

Andre W Visser. 1997. Using random walk models to simulate the vertical distribution of particles in a turbulent water column. *Marine Ecology Progress Series* 158 (1997), 275–281.

D. Walker. 2002. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages.* MIT Press.

D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *Princ. of Prog. Lang. (POPL'19)*.

Di Wang, Jan Hoffmann, and Thomas Reps. 2020a. Tail Bound Analysis for Probabilistic Programs via Central Moments. arXiv:2001.10150 [cs.PL]

Di Wang, David M Kahn, and Jan Hoffmann. 2020b. Raising Expectations: Automating Expected Cost Analysis with Types. arXiv:2006.14010 [cs.PL]

P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi. 2019. Cost Analysis of Nondeterministic Probabilistic Programs. In *Prog. Lang. Design and Impl. (PLDI'19)*.

P. Wang, D. Wang, and A. Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17)*.

D. Williams. 1991. *Probability with Martingales.* Cambridge University Press.

D. Wingate and T. Weber. 2013. *Automated Variational Inference in Probabilistic Programming.* Technical Report. Computing Research Repository.

H. Xi. 2002. Dependent Types for Program Termination Verification. *J. Higher-Order and Symbolic Comp.* 15 (2002). Issue 1.

F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *Static Analysis Symp. (SAS'11)*.