Bounded Expectations: Resource Analysis for Probabilistic Programs

Van Chan Ngo
Carnegie Mellon University, USA

Quentin Carbonneaux
Yale University, USA

Jan Hoffmann
Carnegie Mellon University, USA

Abstract

This paper presents a new static analysis for deriving upper bounds on the expected resource consumption of probabilistic programs. The analysis is fully automatic and derives symbolic bounds that are multivariate polynomials in the inputs. The new technique combines manual state-of-the-art reasoning techniques for probabilistic programs with an effective method for automatic resource-bound analysis of deterministic programs. It can be seen as both, an extension of automatic amortized resource analysis (AARA) to probabilistic programs and an automation of manual reasoning for probabilistic programs that is based on weakest preconditions. An advantage of the technique is that it combines the clarity and compositionality of a weakest-precondition calculus with the efficient automation of AARA. As a result, bound inference can be reduced to off-the-shelf LP solving in many cases and automatically-derived bounds can be interactively extended with standard program logics if the automation fails. Building on existing work, the soundness of the analysis is proved with respect to an operational semantics that is based on Markov decision processes. The effectiveness of the technique is demonstrated with a prototype implementation that is used to automatically analyze 39 challenging probabilistic programs and randomized algorithms. Experiments indicate that the derived constant factors in the bounds are very precise and even optimal for some programs.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords probabilistic programming, resource bound analysis, static analysis

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192394

ACM Reference Format:
Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3192366.3192394

1 Introduction

Probabilistic programming [67, 84] is an increasingly popular technique for implementing and analyzing Bayesian Networks and Markov Chains [43], randomized algorithms [9], cryptographic constructions [11], and machine-learning algorithms [42]. Compared with deterministic programs, reasoning about probabilistic programs adds additional complexity and challenges. As a result, there is a renewed interest in developing automatic and manual analysis and verification techniques that help programmers to reason about their probabilistic code. Examples of such developments are probabilistic program logics [21, 63, 67, 72, 83], automatic probabilistic invariant generation [23, 26], abstract interpretation for probabilistic programs [30, 73, 74], symbolic inference [40], and probabilistic model checking [64].

One important property that is often part of the formal and informal analysis of programs is *resource bound analysis*: What is the amount of resources such as time, memory or energy that is required to execute a program? Over the past decade, the programming language community has developed numerous tools that can be used to (semi-)automatically derive non-trivial symbolic resource bounds for imperative [17, 47, 66, 87] and functional programs [7, 33, 55, 92].

Existing techniques for resource bound analysis can be applied to derive bounds on the *worst-case* resource consumption of probabilistic programs. However, if the control-flow is influenced by probabilistic choices then a worst-case analysis is often not applicable because there is no such upper bound. Consider the example *trader*(s_{min},*s*) in Figure 1(a) that implements a 1-dimensional random walk to model the fluctuations of a stock price *s*. With probability $\frac{1}{4}$ the price increases by 1 point and with probability $\frac{3}{4}$ the price decreases by 1 point. After every price change, a trader performs an action *trade(s)*—like buying 10 shares—that can depend on the current price until the stock price falls to s_{min}. In the worst case, *s* will be incremented in every loop iteration and the loop will not terminate. However, the loop terminates with probability 1 or *almost surely* [36].

```
void trader(int s_min, int s) {
  assume (s_min >=0);
  while (s > s_min) {
    s=s+1 ⊕_{1/4} s=s-1;
    trade(s);
  }
}
```
(a)

```
void trade(int s) {
  int nShares;
  nShares=unif(0,10);
  while (nShares>0) {
    nShares=nShares-1;
    cost=cost+s;
  }
}
```
(b)

Figure 1. (a) A 1-dimensional random walk that models the progression of a stock price that is incremented with probability $\frac{1}{4}$ and decremented with probability $\frac{3}{4}$ while it is greater than $s_{\min}$. (b) A trader that decides to buy between 0 and 10 shares by sampling from a uniform distribution. The program cost is modeled by the global variable *cost*.

While almost sure termination is a useful property, we might be also interested in the distribution of the *number of loop iterations* or, considering a different resource, in *the spending of our trader*. The distribution of the spending depends of course on the implementation of the auxiliary function *trade()*. In general, it is not straightforward to derive such distributions, even for relatively simple programs. Consider for example the implementation of *trade()* in Figure 1(b). It models a trader that randomly buys between 0 and 10 shares according to a uniform distribution. It is not immediately clear what the distribution of the cost is.

In this article, we are introducing a new method for automatically deriving bounds on the expected resource consumption of probabilistic programs. Such bounds can be directly used to predict the amount of resources required to sample from a probabilistic program. Moreover, bounds on the expected resource usage can also be used to derive tail bounds on the resource usage using Markov's inequality or Chebyshev's inequality (see Section 7).

For example, given the function *trade*(s_{min},*s*), our technique automatically derives the bound $2{\cdot}\max(0, s{-}s_{min})$ on the expected number of loop iterations. For the total spending of the trader, we automatically derive the bound

$$5{\cdot}|[s_{min}, s]| + 10{\cdot}|[s_{min}, s]|{\cdot}|[0, s_{min}]| + 5{\cdot}|[s_{min}, s]|^2$$

on the expected value of the variable *cost* in less than 4.6 seconds. Here, we write $|[a, b]|$ for $\max(0, b{-}a)$. Both bounds are tight in the sense that they precisely describe the expected resource consumption.

Our technique derives symbolic polynomial bounds and generates certificates that are derivations in a quantitative program logic [63, 83]. To the best of our knowledge, we present the first fully automatic analysis for deriving symbolic bounds on the expected resource consumption of probabilistic programs with probabilistic branchings and sampling from discrete distributions. It is also one of the few techniques that can automatically derive polynomial properties. Different resource metrics can be defined either by using a resource-counter variable or by using *tick* commands. The analysis is compositional, automatically tracks size changes, and derives whole program bounds. Note that derived time bounds also imply *positive termination* [36], that is, termination with bounded expected runtime. Compositionality is particularly tricky for probabilistic programs since the composition of two positively terminating programs is not a positively terminating in general. While we focus on bounds on the expected cost, the analysis can also be used to derive worst-case bounds. Moreover, we can adapt the analysis (following [76]) to also derive lower bounds on the expected resource usage.

Resource bound analysis and static analysis of probabilistic programs have developed largely independently. The key insight of our work is that there are close connections between (manual) quantitative reasoning methods for probabilistic programs and automatic resource analyses for deterministic programs. Our novel analysis combines probabilistic quantitative reasoning using a weakest precondition (WP) calculus [21, 63, 72, 83] with an automatic resource analysis method that is known as automatic amortized resource analysis (AARA) [18, 20, 51, 55], while preserving the best properties of both worlds. On the one hand, we have the strength and conceptual simplicity of the WP calculus. On the other hand, we get template-based bound inference that can be efficiently reduced to off-the-shelf LP solving.

We implemented our analysis in the tool Absynth. We currently support imperative integer programs that features procedures, recursion, and loops. We have performed an evaluation with 39 probabilistic programs and randomized algorithms that include examples from the literature and new challenging benchmarks. To determine the precision of the analysis, we compared the statically-derived bounds with the experimentally-measured resource usage for different inputs derived by sampling. Our experiments show that we often derive bounds with very precise constant factors and that bound inference usually only takes seconds. In summary, we make the following contributions.

- We describe the first automatic analysis that derives symbolic bounds on the expected resource usage of probabilistic programs.
- We prove the soundness of the method by showing that a successful bound analysis produces derivations in a probabilistic WP calculus [83].
- We show the effectiveness of the technique with a prototype implementation and by successfully analyzing 39 examples from a new benchmark set and from previous work on probabilistic programs and randomized algorithms.

The advantages of our technique are compositionality, efficient reduction of bound inference to LP solving, and compatibility with manual bound derivation in the WP calculus.

2 Probabilistic Programs

In this section, we first recall some essential concepts and notations from probability theory that are used in this paper. We then present the syntax of our imperative probabilistic programming language.

2.1 Essential Notions and Concepts

The interested readers can find more detailed descriptions in standard textbooks [5].

Probability Space. Consider a random experiment. The set Ω of all possible outcomes is called the *sample space*. A *probability space* is a triple $(\Omega, \mathscr{F}, \mathbb{P})$, where $\mathscr{F}$ is a σ-algebra of Ω and $\mathbb{P}$ is a probability measure for $\mathscr{F}$, that is, a function from $\mathscr{F}$ to the closed interval $[0,1]$ such that $\mathbb{P}(\Omega) = 1$ and $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$ for all disjoint sets $A, B \in \mathscr{F}$. The elements of $\mathscr{F}$ are called *events*. A function $f : \Omega \to \Omega'$ is *measurable* w.r.t $\mathscr{F}$ and $\mathscr{F}'$ if $f^{-1}(B) \in \mathscr{F}$ for all $B \in \mathscr{F}'$.

Random Variable. A *random variable* X is a measurable function from a probability space $(\Omega, \mathscr{F}, \mathbb{P})$ to the real numbers, e.g., it is a function $X : \Omega \to \mathbb{R} \cup \{-\infty, +\infty\}$ such that for every Borel set $B \in \mathscr{B}$, $X^{-1}(B) := \{\omega \in \Omega | X(\omega) \in B\} \in \mathscr{F}$. Then the function $\mu_X(B) = \mathbb{P}(X^{-1}(B))$, called *probability distribution*, is a probability measure for $\mathscr{B}$ and $(\mathbb{R}, \mathscr{B}, \mu_X)$ is a probability space. If μ_X measures on a countable set of reals, or the range of X is countable, then X is called a *discrete random variable*. If μ_X gives zero measure to every singleton set, then X is called a *continuous random variable*. The distribution μ_X is often characterized by the *cumulative distribution function* defined by $F_X(x) = \mathbb{P}(X \leq x) = \mu_X((-\infty, x])$.

Expectation. The *expected value* of a discrete random variable X is the weighted average $\mathbb{E}(X) := \sum_{x_i \in R_X} x_i \mathbb{P}(X = x_i)$, where R_X is the range of X. To emphasize the distribution μ_X, we often write $\mathbb{E}_{\mu_X}(X)$ instead of $\mathbb{E}(X)$. An important property of the expectation is *linearity*. If X and X' are random variables and $\lambda, \mu \in \mathbb{R}$ then $Y = \lambda X + \mu X'$ is a random variable and $\mathbb{E}(Y) = \lambda \mathbb{E}(X) + \mu \mathbb{E}(X')$.

2.2 Syntax of Probabilistic Programs

The probabilistic programming language we use is a simple imperative integer language structured into expressions and commands. The abstract syntax is given by the grammar in Figure 2. The command $\mathsf{id} = e \ \mathsf{bop} \ R$, where R is a (discrete) random variable whose probability distribution is μ_R (written as $R \sim \mu_R$), is a *random sampling* assignment. It first samples according to the distribution μ_R to obtain a sample value and then evaluates the expression in which R is replaced by the sample value. Finally, the evaluated value is assigned to the variable id. The command $c_1 \oplus_p c_2$ is a *probabilistic branching*. It executes the command c_1 with probability p, or the command c_2 with probability $(1-p)$.

The command $\mathsf{if} \star c_1 \ \mathsf{else} \ c_2$ is a *non-deterministic* choice between c_1 and c_2. The command $\mathsf{call} \ P$ makes a (possibly recursive) call to the procedure with identifier $P \in \mathsf{PID}$. In this article, we assume that procedures only manipulate the global program state. Thus, we avoid to use local variables, arguments, and return commands for passing information across procedure calls. However, we support local variables and return commands in the implementation. Arguments, can be easily simulated by using global variables as registers.

$$
\begin{array}{lll}
e & := & \mathsf{id} \mid n \mid e_1 \ \mathsf{bop} \ e_2 \\
c & := & \mathsf{skip} \mid \mathsf{abort} \mid \mathsf{assert} \ e \mid \mathsf{tick}(q) \mid \mathsf{id} = e \\
 & & \mid \mathsf{id} = e \ \mathsf{bop} \ R \mid \mathsf{if} \ e \ c_1 \ \mathsf{else} \ c_2 \mid \mathsf{if} \star c_1 \ \mathsf{else} \ c_2 \\
 & & \mid c_1 \oplus_p c_2 \mid c_1; c_2 \mid \mathsf{while} \ e \ c \mid \mathsf{call} \ P \\
\mathsf{bop} & := & + \mid - \mid * \mid \mathsf{div} \mid \mathsf{mod} \mid == \mid <> \mid > \mid < \mid <= \mid \ldots \\
R & \sim & \mu_R \ \text{(probability distribution)}
\end{array}
$$

Figure 2. Abstract syntax of the probabilistic language.

We include the built-in primitive assert e that terminates the program if the expression e evaluates to 0 and does nothing otherwise. The primitive $\mathsf{tick(q)}$, where $q \in \mathbb{Q}_{\geq 0}$ is used to model resource usage of commands and thus to define the *cost model*. As we have seen in the introduction, we can also derive bounds on regular variables.

A program is a pair $(c, \mathscr{D})$, where $c \in C$ is the body of the main procedure and $\mathscr{D} : \mathsf{PID} \to C$ is a map from procedure identifiers to their bodies. A command with no procedure calls is called *closed* command.

3 Expected Resource Bound Analysis

In this section, we show informally how automatic amortized resource analysis (AARA) [19, 51, 55] can be generalized to compute upper bounds on the expected resource consumption of probabilistic programs. We first illustrate with a classic analysis of a one-dimensional random walk how developers currently analyze the expected resource usage. We then recap AARA for imperative programs [19, 20]. Finally, we explain the new concept of the *expected potential method* that we develop in this work and apply it to our random walk and more challenging examples.

3.1 Manual Analysis of a Simple Random Walk

Consider the following implementation of a random walk.

```
while (x>0) { x=x-1 ⊕3/4 x=x+1; tick(1); }
```

The traditional way to analyze this program is using recurrence relations. Let $T(n)$ be the expected runtime of the program when x is initially n. By expected runtime we mean the expected number of $\mathsf{tick}(1)$ statements executed. Then, for all $n \geq 1$, $T(n)$ satisfies the following equation

$$T(n) = 1 + \frac{3}{4}T(n-1) + \frac{1}{4}T(n+1)$$

This is a recurrence relation of degree 3, but it can be solved more easily by defining $D(n)$ to be $T(n) - T(n-1)$ and rewriting the equation as $3D(n) = 4 + D(n+1)$. One systematic way

to find solutions for this equation is to use the generating function $G(z) = \sum_{n\geq 1} D(n)z^n$. Writing D for $D(1)$, we get

$$3G(z) = 4\sum_{n\geq 1} z^n + \frac{1}{z}G(z) - D = \frac{4z}{1-z} + \frac{1}{z}G(z) - D$$

And thus, using algebra and generating functions, we have

$$G(z) = \sum_{n\geq 1}(2 - D\cdot 3^{n-1} + 2\cdot 3^{n-1})z^n$$

To finish the reasoning, we have to find the constant D using a boundary condition. It is clear that $T(0) = 0$, because the loop is never entered when the program is started with $x = 0$. To find $T(1)$, we observe that the program has to first reach the state $x = n-1$ to terminate with $x = n$. Additionally, because each coin flip is independent of the others, reaching $n-1$ from n should take the same expected time as reaching 0 from 1. Thus, $T(n) = n\cdot T(1) = n\cdot D$ and consequently $D(n) = T(n) - T(n-1) = D$ for $n \geq 1$. Therefore, we have $D = 2$ and $T(n) = 2\cdot n$.

There are several reasons why this classic method is hard to automate. First, inferring the recurrence relations is not always straightforward, for example, because of complex iteration patterns. Additionally, it is difficult to formally prove a correspondence between the program and the recurrence relation. Second, the method for solving recurrence relations is fragile. If the decrement is $x = x-2$ instead of $x = x-1$ then the above boundary condition does not work anymore. Moreover, recurrence relations become more difficult to solve with the use of bigger constants and multiple variables. Finally, the classic method is not compositional. When programs become larger, it does not provide a principled way to reason independently on smaller components.

3.2 The Potential Method

It has been shown in the past decade that the *potential method* of amortized analysis provides an interesting alternative to classic resource analysis with recurrence relations.

Assume that a program c executes with initial state σ to final state σ' and consumes n resource units as defined by the tick commands, denoted $(c,\sigma) \Downarrow_n \sigma'$. The idea of amortized analysis is to define a *potential function* $\Phi : \Sigma \to \mathbb{Q}_{\geq 0}$ that maps program states to non-negative numbers and to show that $\Phi(\sigma) \geq n$ for all σ such that $(c,\sigma) \Downarrow_n \sigma'$.

To reason compositionally, we have to take into account the state resulting from a program execution. We thus use two potential functions Φ and Φ' that specify the available potential before and after the execution, respectively. The functions must satisfy the constraint $\Phi(\sigma) \geq n + \Phi'(\sigma')$ for all states σ and σ' such that $(c,\sigma) \Downarrow_n \sigma'$. Intuitively, $\Phi(\sigma)$ must be sufficient to pay for the resource cost of the computation and for the potential $\Phi'(\sigma')$ on the resulting state σ'. Thus, if $(\sigma, c_1) \Downarrow_n \sigma'$ and $(\sigma', c_2) \Downarrow_m \sigma''$, we have $\Phi(\sigma) \geq n + \Phi'(\sigma')$ and $\Phi'(\sigma') \geq m + \Phi''(\sigma'')$ and therefore $\Phi(\sigma) \geq (n+m) + \Phi''(\sigma'')$. Note that the initial potential function Φ provides an upper bound on the resource consumption of the whole program. What we have observed is that, if we define $\{\Phi\}c\{\Phi'\}$ to mean

$$\forall \sigma\, n\, \sigma'.\, (\sigma, c) \Downarrow_n \sigma' \implies \Phi(\sigma) \geq n + \Phi'(\sigma')$$

then the following familiar inference rule is valid.

$$\frac{\{\Phi\}c_1\{\Phi'\} \qquad \{\Phi'\}c_2\{\Phi''\}}{\{\Phi\}c_1; c_2\{\Phi''\}}\ (\text{Q:SEQ})$$

Other language constructs lead to rules for the potential functions that look very similar to Hoare logic or effect system rules. These rules enable reasoning about resource usage in a flexible and compositional way, which, as a side effect, produces a certificate for the derived resource bound.

The derivation of a resource bound using potential functions is best explained by example. In the following deterministic example, the worst-case cost can be bounded by $|[x,y]| = \max(0, y-x)$.

```
while (x<y) { x=x+1; tick(1); }
```

To derive this bound, we start with the initial potential $\Phi_0 = |[x,y]|$, which we also use as the loop invariant. For the loop body we have (like in Hoare logic) to derive a triple $\{\Phi_0\}x = x+1;\ \text{tick}(1)\{\Phi_0\}$. We can only do so if we utilize the fact that $x < y$ at the beginning of the loop body. The reasoning then works as follows. We start with the potential $|[x,y]|$ and the fact that $|[x,y]| > 0$ before the assignment. If we denote the updated version of x after the assignment by x' then the relation $|[x,y]| = |[x',y]| + 1$ between the potentials before and after the assignment holds. This means that we have the potential $|[x,y]| + 1$ before $\text{tick}(1)$, which consumes 1 resource unit. We end up with potential $|[x,y]|$ after the loop body and have established the loop invariant.

This reasoning can be automated in three steps (also see Section 5). First, we create a derivation with template potential functions that contain a priori unknown rational coefficients. Second, we use local inference rules to generate constraints on the coefficients. Third, we solve the constraints with linear optimization and minimize the initial potential.

The use of the potential method is not in opposition to recurrence solving. One possible view is that solutions to recurrences are integrated with the local inference rules. The advantages of the method are its compositionality, the efficient reduction to LP solving, and the generation of proofs, which can be easily checked by a separate tool.

3.3 The Expected Potential Method

Maybe surprisingly, the potential method of AARA can be adapted to automatically derive upper bounds on the expected cost of probabilistic programs.

Like in the deterministic case, we would like to work with triples of the form $\{\Phi\}c\{\Phi'\}$ for potential functions $\Phi, \Phi' : \Sigma \to \mathbb{Q}_{\geq 0}$ to ensure compositional reasoning. However, in the case of probabilistic programs we need to take

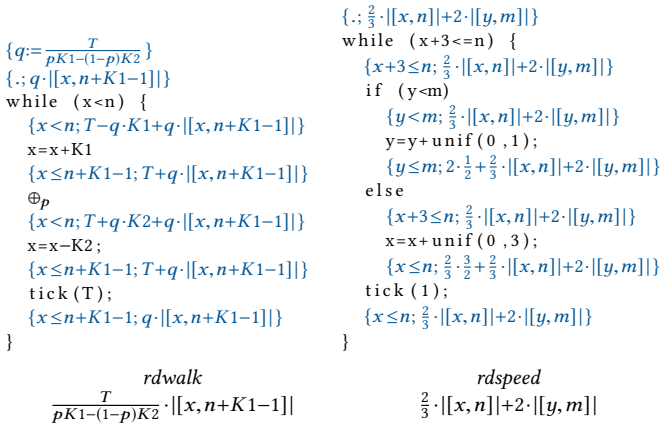

Figure 3. Derivations of bounds for single loop programs.

into account the expected value of the potential function Φ' w.r.t the distribution of final states resulting from the program execution. More precisely, the two functions must satisfy the constraint

$$\Phi(\sigma) \geq \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\text{cost}) + \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\Phi')$$

for all program states σ. Here we write $[\![c,\mathscr{D}]\!](\sigma)$ to denote the probability distribution over the final states as specified by the program $(c,\mathscr{D})$ and initial state σ. Finally, $\mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\text{cost})$ is the expected resource usage of executing c from the initial state σ. The intuitive meaning is that the potential $\Phi(\sigma)$ is sufficient to pay for the expected resource consumption of the execution from σ and the expected potential with respect to the probability distribution over the final states. Let $\Phi(\sigma) \geq \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\text{cost}) + \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\Phi')$ and $\Phi'(\sigma_i') \geq \mathbb{E}_{[\![c',\mathscr{D}]\!](\sigma_i')}(\text{cost}) + \mathbb{E}_{[\![c',\mathscr{D}]\!](\sigma_i')}(\Phi'')$ for all sample states σ_i' from $[\![c,\mathscr{D}]\!](\sigma)$. Then we have for all states σ

$$\Phi(\sigma) \geq \mathbb{E}_{[\![c;c',\mathscr{D}]\!](\sigma)}(\text{cost}) + \mathbb{E}_{[\![c;c',\mathscr{D}]\!](\sigma)}(\Phi'')$$

Hence, the initial potential Φ gives an upper-bound on the expected value of resource consumption of the sequence $c;c'$ like in the sequential version of potential-based reasoning. If we write $\{\Phi\}c\{\Phi'\}$ to mean $\Phi(\sigma) \geq \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\text{cost}) + \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\Phi')$ for all program states σ then we have again the familiar rule Q:Seq for compositional reasoning above.

Note that expected and worst-case resource consumption are identical for deterministic programs. Therefore, the expected potential method derives worst-case bounds for deterministic programs.

Analyzing a Random Walk. Using the expected potential method simplifies the reasoning significantly and, as we show in this article, can be automated using a template-based approach and LP solving like in the deterministic case.

Consider the simple random walk from Section 3.1 again whose expected resource usage is $\Phi(x) = 2|[0,x]|$. This is the potential that we have available before the loop and it will also serve as a loop invariant. We have to prove that the potential right after the probabilistic branching should be $2|[0,x]| + 1$, to pay for the cost of the tick statement and to restore the loop invariant. What remains to justify is how the probabilistic branching turns the potential $2|[0,x]|$ into $2|[0,x]| + 1$. To do so, we reason backwards independently on the two branches. For each branch, what is the initial potential required to ensure an exit potential of $2|[0,x]| + 1$? *(i)* The assignment $x = x - 1$ needs initial potential $\Phi_1(x) = 2|[0,x]| - 1$. Indeed if we write x' for the value of x after the assignment then $2|[0,x]|-1 = 2|[0,x'+1]|-1 = 2|[0,x']|+1$. *(ii)* Similarly, the second branch needs the initial potential $\Phi_2(x) = 2|[0,x]| + 3$. Intuitively, since we enter the first and second branches with probabilities $\frac{3}{4}$ and $\frac{1}{4}$, respectively, thus the initial potential for the probabilistic branching should be the weighted sum $\frac{3}{4}\Phi_1(x)+\frac{1}{4}\Phi_2(x) = 2|[0,x]|$. This reasoning would restore the loop invariant and prove the desired bound.

General Random Walk. Now consider the generalized version *rdwalk* in Figure 3. It simulates a general *one-dimensional random walk* [45] with arbitrary positive constants K_1, K_2, T, and p. The expected number of loop iterations, and thus the expected cost modeled by the command tick(T), is bounded iff $(\star)$ $pK_1-(1-p)K_2>0$. In this case, the expected distance for "forward walking" is bigger than the expected distance for backward walking.

For the annotations in the figure, we use semicolons to separate the logical assertions from the potential functions. The analysis for this example is very similar to the simple one. It is only valid when the condition $(\star)$ is satisfied since the initial potential function would be negative otherwise. In the implementation, the automatic analysis reports that no bound can be found if the program does not satisfy this requirement. Note that the classic method would be a lot more complex in this more general case. Indeed, the degree of the recurrence relation to solve would be K_1+K_2+1 and if $K_1>1$ the boundary condition argument we gave in Section 3.1 would not be valid anymore.

3.4 Bound Derivations for Challenging Examples

We show how the expected potential method can derive polynomial bounds for challenging probabilistic programs with single, nested, and sequential loops, as well as procedure calls (more examples are given in the TR [75]). All the presented bounds are derived automatically by our tool Absynth whose implementation and inference algorithm are discussed in Sections 5 and 8.

Single Loops. Examples *rdwalk* and *rdspeed* in Figure 3 show that our expected potential method can handle *tricky iteration patterns*. Example *rdwalk* has already been discussed. Example *rdspeed* is randomized version of the one from papers on worst-case bound analysis [20, 47]. The iteration first increases y by 0 or 1 until it reaches m randomly according to the uniform distribution. Then x is increased by $k \in [0,3]$, which is sampled uniformly. Absynth derives the

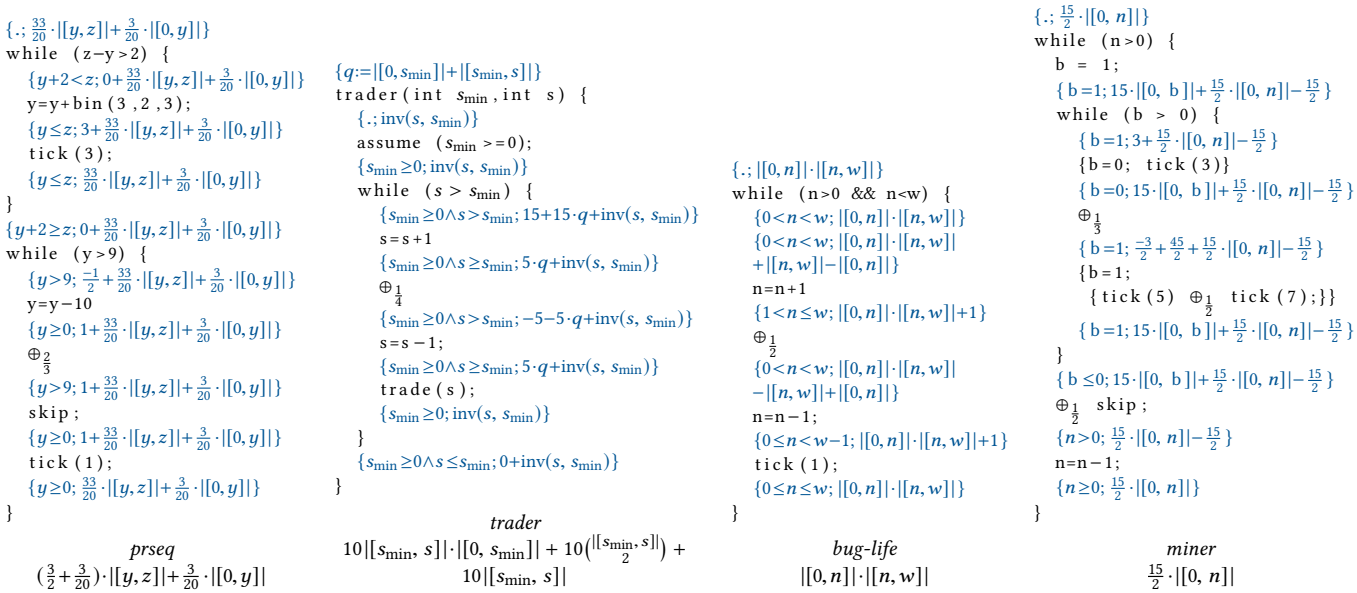

Figure 4. Derivations of bounds on the expected value of ticks for probabilistic programs. Example *prseq* contains a sequential loop so that the iterations of the second loop depend on the the first one. In the derivation of *trader*, the non-linear bound $\text{inv}(s, s_{\min}) := 10|[s_{\min}, s]| \cdot |[0, s_{\min}]| + 10\binom{|[s_{\min}, s]|}{2} + 10|[s_{\min}, s]|$ on the global variable *cost* is derived. Example *miner* shows how Boolean variables can be expressed and handled.

tight bound $\frac{2}{3} \cdot |[x,n]| + 2 \cdot |[y,m]|$. The derivation is similar to the derivation of the bound for *rdwalk*. To reason about the sampling construct, we consider the effect of all possible samples on the potential function. With the same reasoning for probabilistic branching, we then compute the weighted sum on the resulting initial potentials where the weights are assigned following the distribution. In some cases (like in this one), it is sound to just replace the distribution with the expected outcome. However, this does not work in general since the resource consumption could depend on the sample in a non-uniform way.

Nested and Sequential Loops. The examples in Figure 4 show how the expected potential method can effectively handle *interacting nested*, *sequential loops*, and procedure calls to effectively derive polynomial bounds. The example *prseq* shows the capability of the expected potential method to take into account the interactions between sequential loops by deriving the expected value of the size changes of the variables. In the first loop, the variable y is incremented by sampling from a binomial distribution with parameters $n = 3$ and $p = \frac{2}{3}$. In the second loop, y is decreased by 10 with probability $\frac{2}{3}$ or left unchanged otherwise. We accurately derive the expected value of the size change of y by transferring the potential $|[y,z]|$ to $|[0,y]|$. Example *miner* simulates a miner who is trapped in a mine. The miner is sent to the mine for n times independently, for each time, with probability $\frac{1}{2}$ she is trapped and with the same probability she is safe. When she is trapped, there are 3 doors in the mine for using. The first door leads to a tunnel that will take her to safety after 3 hours (representing by 3 ticks). The second door leads to a tunnel that returns her to the mine after 5 hours. And the third door leads to a tunnel that returns her to the mine after 7 hours. At all times, the miner is equally likely to choose any one of the doors, meaning that she chooses any door with equal probability $\frac{1}{3}$. This example demonstrates how the analysis handles Boolean values (the variable b) to get the tight bound $\frac{15}{2} \cdot |[0,n]|$ on the expected time. It assigns the potential $15 \cdot |[0, b]|$ to the variable b for paying the cost of the inner loop, in which $\frac{15}{2} \cdot |[0,n]| - \frac{15}{2} + 15 \cdot |[0, b]|$ is the loop invariant.

Non-linear Bounds. Example *bug-life* simulates the tragic life of an oblivious flea [2]. The flea starts at position $0 < n < w$ inches and his life will end if he falls off the *cliff of doom* (position 0) or falls into the *pit of disaster* (position w). He repeatedly hops 1 inch to the right or 1 inch to the left with equal probability, independently from the direction of all previous hops. The derivation infers the tight bound $n \cdot (w-n)$ on the expected number of hops he takes before falling off the cliff or into the pit. When there is no pit of disaster, the inferred value can be used to prove that the expected lifespan of the flea is not bounded above. Thus, the flea is certain to eventually fall of the cliff of doom but his expected lifespan might be infinite. This demonstrates the *soundness* property of our analysis, that is, if the expected runtime can not be bounded above then the program might be not positive termination. In the implementation, the bound for

bug-life is inferred with user-provided rewrite functions as hints.

Example *trader* has a non-linear bound that demonstrates that our expected potential method can handle programs with nested loops and procedure calls, which often have a super-linear expected resource consumption. In the derivation $\mathrm{inv}(s, s_{\min})$ acts as a loop invariant. We assume that we already established the bound $5|[0,s]| = 5(|[0,s_{\min}]| + |[s_{\min},s]|)$ for the procedure *trade(s)*. The crucial point for understanding the derivation is the reasoning about the probabilistic branching. First, observe that the weighted sum of the two pre-potentials of the branches is equal to $\mathrm{inv}(s, s_{\min})$ if the weights are $\frac{1}{4}$ and $\frac{3}{4}$. Second, verify that the post potential is equal to the potential in the respective pre-potential if we substitute $s+1$ (or $s-1$) for s.

3.5 Limitations

Deriving symbolic resource bounds is an undecidable problem and our technique does not find bounds if loops and recursive functions have complex control flow. Like every static analysis we cannot prove facts that follow from mathematical insights like the existence of an infinite number of primes. The current implementation in Absynth is limited to polynomial bounds and linear size changes of loop counters. However, this is not an inherent limitation of the method [19]. Moreover, we only support sampling from discrete distributions with a finite domain. For instance, Absynth cannot find a bound for the following program.

```
while (x>0) { x = x/2; tick(1); }
```

In the technical development, we do not cover local variables and function arguments. However, local variables are implemented in Absynth. While we conjecture that the technique also works for non-monotone resources like memory that can become available during the evaluation, our current meta-theory only covers monotone resources like time.

The set of programs for which the analysis can successfully derive bounds depends on the base and rewrite functions that are selected (see Section 5). A systematic description of such sets is a separate research question that is closely related to guarantees on the precision of the analysis.

4 Derivation System for Probabilistic Quantitative Analysis

In this section, we describe the inference system used by our analysis. It is presented like a program logic and enables compositional reasoning. As we explain in Section 5, the inference of a derivation can be reduced to LP solving.

4.1 Potential Functions

The main idea to automate resource analysis using the potential method is to fix the shape of the potential functions. More formally, potential functions are taken to be linear combinations of more elementary *base functions*. Finding the suitable base functions for a given program is discussed in Section 8. For now we assume a given list of N base functions. For convenience, they are represented as a vector $B = (b_1, \ldots, b_N)$, where each $b_i : \Sigma \to \mathbb{Q}$ maps program states to rational numbers. Building on base functions, a potential function is defined by N coefficients $q_1, \ldots, q_N \in \mathbb{Q}$ as $\Phi(\sigma) = \sum_{i=1}^{N} q_i \cdot b_i(\sigma)$. The coefficients are written as a vector $Q = (q_1, \ldots, q_N)$, called *potential annotation*. Each potential annotation corresponds to a potential function Φ_Q that we can concisely express as the dot product $\langle Q \cdot B \rangle$.

Note that potential annotations form a vector space. Additionally, using the bi-linearity of the dot product, operations on the vector space of annotations correspond directly to operations on potential functions, that is $\Phi_{\lambda Q + \mu Q'} = \langle \lambda Q + \mu Q' \cdot B \rangle = \lambda \langle Q \cdot B \rangle + \mu \langle Q' \cdot B \rangle = \lambda \Phi_Q + \mu \Phi_{Q'}$. In the following, we assume that the constant function $\mathbf{1}$ defined by $\lambda\sigma.1$ is in the list of base functions B. This way, the constant potential function $\lambda\sigma.k$ can be represented with a potential annotation where the coefficient of $\mathbf{1}$ is k and all other coefficients are 0.

4.2 Judgements

The main judgement of our inference system defines the validity of a triple $\vdash \{\Gamma, Q\} c \{\Gamma', Q'\}$. In the triple, c is a command, $\{\Gamma, Q\}$ is the precondition, and $\{\Gamma', Q'\}$ is the postcondition. Γ is a *logical context* and Q is a potential annotation. The logical context $\Gamma \in \mathscr{P}(\Sigma)$ is a predicate on program states inferred by our implementation using abstract interpretation. It describes a set of permitted states at a given program point.

Leaving the logical contexts aside—they have the same semantics as in Hoare logic—a triple $\{\cdot, Q\} c \{\cdot, Q'\}$ expresses that for any continuation command c' with expected cost bounded by $\Phi_{Q'}$, the expected cost of the command $c; c'$ is bounded by Φ_Q. When looking for the expected cost of the command c, one can simply use `skip` as the command c' and derive a triple where $\Phi_{Q'} = \mathbf{0}$. In that case, Φ_Q is a bound on the expected cost of the command c. To handle procedure calls, the judgement for a triple uses a *specification context* Δ. This context assigns *specifications* to the procedures of the program and permits a compositional analysis that also handles recursive procedures. A specification is a valid pair of pre- and post-conditions for the procedure body, denoted $\Delta \vdash \{\Gamma; Q\}\ \mathscr{D}(P)\ \{\Gamma'; Q'\}$. The judgement $\vdash \Delta$, defined by the rule ValidCtx, states that all the procedure specifications in the context Δ are valid, that is, the specifications are correct pre- and post-conditions for the procedure bodies. Note that a context Δ can contain multiple specifications for the same procedure. This enables a context-sensitive analysis.

Notations and Conventions. For a program state $\sigma \in \Sigma$ (e.g., a map from variable identifiers to integers), we write $[\![e]\!]_\sigma$ to denote the value of the expression e in σ and $\sigma[v/x]$ for the program state σ extended with the mapping of x to

$$\text{(Q:Weaken)}\quad \frac{\Gamma \models \Gamma_0 \qquad Q \geq_\Gamma Q_0 \qquad \vdash \{\Gamma_0; Q_0\}c\{\Gamma_0'; Q_0'\} \qquad \Gamma_0' \models \Gamma' \qquad Q_0' \geq_{\Gamma_0'} Q'}{\vdash \{\Gamma; Q\}c\{\Gamma'; Q'\}}$$

$$\text{(ValidCtx)}\quad \frac{P : (\Gamma; Q, \Gamma'; Q') \in \Delta \Rightarrow \Delta \vdash \{\Gamma; Q\}\ \mathscr{D}(P)\ \{\Gamma'; Q'\}}{\vdash \Delta}$$

$$\text{(Q:Assert)}\quad \frac{}{\vdash \{\Gamma; Q\}\ \text{assert}\ e\ \{\Gamma \wedge e; Q\}} \qquad \text{(Q:Tick)}\quad \frac{}{\vdash \{\Gamma; Q\}\ \text{tick}\,(q)\ \{\Gamma; Q - q\}}$$

$$\text{(Q:NonDet)}\quad \frac{\vdash \{\Gamma; Q\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}\ \text{if}\ \star\ c_1\ \text{else}\ c_2\ \{\Gamma'; Q'\}}$$

$$\text{(Q:If)}\quad \frac{\vdash \{\Gamma \wedge e; Q\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma \wedge \neg e; Q\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}\ \text{if}\ e\ c_1\ \text{else}\ c_2\ \{\Gamma'; Q'\}}$$

$$\text{(Q:Loop)}\quad \frac{\vdash \{\Gamma \wedge e; Q\}c\{\Gamma; Q\}}{\vdash \{\Gamma; Q\}\ \text{while}\ e\ c\ \{\Gamma \wedge \neg e; Q\}}$$

$$\text{(Q:PIf)}\quad \frac{Q = p{\cdot}Q_1 + (1-p){\cdot}Q_2 \qquad \vdash \{\Gamma; Q_1\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}c_1 \oplus_p c_2\{\Gamma'; Q'\}}$$

$$\text{(Q:Seq)}\quad \frac{\vdash \{\Gamma; Q\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma'; Q'\}c_2\{\Gamma''; Q''\}}{\vdash \{\Gamma; Q\}c_1; c_2\{\Gamma''; Q''\}}$$

$$\text{(Q:Assign)}\quad \frac{A = (a_{i,j}) \qquad \forall j \in \mathscr{S}_{x=e}.\, b_j[e/x] = \sum_i a_{i,j} \cdot b_i \qquad \forall j \notin \mathscr{S}_{x=e}.\, a_{i,j} = 0 \qquad \forall j \notin \mathscr{S}_{x=e}.\, q_j' = 0 \qquad Q = AQ'}{\vdash \{\Gamma[e/x]; Q\}x = e\{\Gamma; Q'\}} \qquad \text{(Q:Skip)}\quad \frac{}{\vdash \{\Gamma; Q\}\ \text{skip}\ \{\Gamma; Q\}}$$

$$\text{(Q:Sample)}\quad \frac{\Gamma \models R \in [a,b] \qquad \forall v_i \in [a,b].[\![\mu_R : v_i]\!] = p_i \qquad \forall v_i.\, \vdash \{\Gamma; Q_i\}x = e\ \text{bop}\ v_i\{\Gamma'; Q'\} \qquad Q = \sum_i p_i{\cdot}Q_i}{\vdash \{\Gamma; Q\}x = e\ \text{bop}\ R\{\Gamma'; Q'\}} \qquad \text{(Q:Abort)}\quad \frac{}{\vdash \{\Gamma; 0\}\ \text{abort}\ \{\Gamma'; Q'\}}$$

$$\text{(Q:Call)}\quad \frac{P : (\Gamma; Q, \Gamma'; Q') \in \Delta \qquad x \in \mathbb{Q}_{\geq 0}}{\vdash \{\Gamma; Q + x\}\ \text{call}\ P\ \{\Gamma'; Q' + x\}} \qquad \text{(Relax)}\quad \frac{F = (F_1, \ldots, F_N) \qquad \vec{u} = (u_1, \ldots, u_N)^\intercal \qquad \forall i.\Gamma \models \Phi_{F_i} \geq 0 \qquad \forall i.u_i \geq 0 \qquad Q' = Q - F\vec{u}}{Q \geq_\Gamma Q'}$$

Figure 5. Inference rules of the derivation system.

v. For a probability distribution μ, we use $[\![\mu : v]\!]$ to indicate the probability that μ takes value v. We use Σ to denote the set of program states. The entailment relation on logical contexts $\Gamma \models \Gamma'$ means that Γ is stronger than Γ'. We write $\sigma \models \Gamma$ when $\sigma \in \Gamma$. For a proposition p, we write $\Gamma \models p$ to mean that any state σ such that $\sigma \models \Gamma$ satisfies p. For an expression e and a variable x, $\Gamma \wedge e$ stands for the logical context $\{\sigma \mid \sigma \models \Gamma \wedge [\![e]\!]_\sigma \neq 0\}$ and $\Gamma[e/x]$ stands for the logical context $\{\sigma \mid \sigma[e/x] \models \Gamma\}$.

For potential annotations Q, Q' and $\diamond \in \{<, =, \ldots\}$, the relation $Q \diamond Q'$ means that their components are constrained point-wise, that is, $\bigwedge_i q_i \diamond q_i'$. Additionally, we write $Q \pm c$ where $c \in \mathbb{Q}$ to denote the annotation Q' obtained from Q by setting the coefficient q_i' of the base function $\mathbf{1}$ to $q_i \pm c$ and leaving the other coefficients unchanged.

Finally, because potential functions always have to be non-negative, any rule that derives a triple $\{\Gamma; Q\}c\{\Gamma'; Q'\}$ has two extra implicit assumptions: $Q \geq_\Gamma 0$ and $Q' \geq_{\Gamma'} 0$. The fact that these assumptions imply the non-negativity of the potential functions becomes clear when we explain the meaning of $\geq_\Gamma$ in the next section.

4.3 Inference Rules

Figure 5 gives the complete set of rules. We informally describe some important rules and justify their validity. Section 6 gives details about the formal soundness proof.

The rule Q:Tick is the only one that accounts for the effect of consuming resources. The tick command does not change the program state, so we require the logical context Γ in the pre- and postcondition to be the same. Let c' be a command with an expected resource bound $\Phi_{Q'} = \Phi_Q - q$. Because the cost of tick (q) is exactly q resource units, the expected resource bound of tick (q); c' is exactly $\Phi_{Q'} + q = \Phi_Q$.

The rule Q:PIf accounts for probabilistic branching. Let c' be a continuation command with an expected resource bound $\Phi_{Q'}$, T_1 and T_2 be the resource usage of executing c_1 and c_2, respectively. Then by the linearity of the expectations, the expected resource bound of the command $(c_1 \oplus_p c_2)$; c' is $T_c = p \cdot (T_1 + \Phi_{Q'} + (1-p) \cdot (T_2 + \Phi_{Q'})$. Using the hypothesis triples for c_1 and c_2, we have $T_c \leq p \cdot \Phi_{Q_1} + (1-p)\Phi_{Q_2} = \Phi_Q$.

The second probabilistic rule Q:Sample deals with sampling assignments. Recall that R is a random variable following a distribution μ_R. The essence of the rule is that, since we assumed that R is bounded, we can treat a sampling assignment as a (nested) probabilistic branching. Each of the

branches contains an assignment $x = x$ bop v with $v \in \mathbb{Z}$ and is executed with probability p, the probability of the event $R = v$. The preconditions of each of the branches are combined like in the Q:PIF rule.

The rules Q:ASSIGN and Q:WEAKEN are similar to the ones of a previous implementation of AARA for the analysis of non-probabilistic programs [19] but have been adapted to our structured probabilistic language. In the rule Q:ASSIGN for $x = e$, we represent the state transformation as a linear transformation on potential functions. If $\Phi_{Q'}$ is a bound on the expected cost of c', then $\Phi_{Q'}[e/x]$ is the expected resource bound of $x = e; c'$ To encode this constraint as a linear program (this is necessary to enable automation using LP solving), we find all the *stable* base functions, denoted $\mathscr{S}_{x=e}$, that is, all the base functions b_j for which there exists $(a_{i,j})_i \in \mathbb{Q}$ such that $b_j[e/x] = \sum_i a_{i,j} \cdot b_i$. That means a function is stable if its extending with the mapping of x to e can be represented by the set of base functions. Finally, to ensure that the transformation w.r.t the assignment $x = e$ on the potential function $\Phi_{Q'}$ is linear, we require that all the base functions that are not stable have their coefficients set to 0 in Q'. With these constraints, we have that $\Phi_{Q'}[e/x] = \Phi_{AQ'} = \Phi_Q$ where A is the $(N \times N)$ matrix with coefficients $(a_{i,j})$, hence justifying the validity.

The essence of the Q:WEAKEN is that it is always safe to add potential in the precondition and remove potential in the postcondition. This concept of more (or less) potential is made precise by the predicate $Q \geq_\Gamma Q'$. Semantically, $Q \geq_\Gamma Q'$ encodes—using linear constraints—the fact that in all states $\sigma \models \Gamma$, we have $\Phi_{Q'}(\sigma) \geq \Phi_Q(\sigma)$ (see the TR [75] for details). The RELAX rule uses *rewrite functions* $(F_i)_i$, as introduced in [19]. Rewrite functions are linear combinations of base functions that can be proved to be non-negative in the logical context Γ. Using rewrite functions, the idea of the judgement $Q \geq_\Gamma Q'$ is that, to obtain Q', one has to subtract a non-negative quantity from Q.

The rule Q:CALL handles procedure calls. The pre- and postcondition for the procedure P are fetched from the specification context Δ. Then, a non-negative *frame* $x \in \mathbb{Q}_{\geq 0}$ is added to the procedure specification. This frame allows to pass some constant potential through the procedure call and is required for the analysis of most non-tail-recursive functions. In the soundness proof, this framing boils down to the "propagation of constants" property of the calculus [83] used in our formal soundness proof.

5 Constraint Generation and Solving Using LP Solvers

The automatic bound derivation is split in two phases. First, derivation templates and constraints are generated by inductively applying the inference rules to the input program. During this first phase, the coefficients of the potential annotations are left as symbolic names and the inequalities are collected as constraints. Each symbolic name corresponds to a variable in a linear program. Second, we feed the linear program to an off-the-shelf LP solver[1]. If the LP solver returns a solution, we obtain a valid derivation and extract the expected resource bound. Otherwise, an error is reported.

Generating Linear Constraints. A detailed example of this process is shown in Figure 6. Note that Q:WEAKEN is applied twice. Since this rule is not syntax-directed, it can be applied at any point during the derivation. In our implementation, we apply it around all assignments. This proved sufficient in practice and limits the number of constraints generated. In the figure, the potential annotations are represented by an upper-case letter P or Q with an optional superscript. For example, Q represents the potential function

$$q_1 \cdot \mathbf{1} + q_{x0} \cdot |[0, x]| + q_{x1} \cdot |[1, x]| + q_{x2} \cdot |[2, x]|$$

The set of base functions is $\mathbf{1}$ and $|[i, x]|$ for $i \in \{0, 1, 2\}$. We will see that they are sufficient to infer a bound. Details of how to select base functions are given in Section 8. To apply weakening, we need rewrite functions, we pick

$$F_0 = 1; F_1 = -1 + |[0, x]| - |[1, x]|; F_2 = -2 + |[0, x]| - |[2, x]|$$

F_1 is applicable (i.e., non-negative) iff $x \geq 1$. Similarly, F_2 is applicable iff $x \geq 2$. This means that both rewrite functions can be used at the beginning of the loop body, when $x \geq 2$ can be proved because of the loop condition.

The constraints given in the table in Figure 6 use shorthand notations to constrain all the coefficients of two annotations. For instance $Q = Q^{sq}$ should be expanded into $q_1 = q_1^{sq} \wedge q_{x0} = q_{x0}^{sq} \wedge q_{x1} = q_{x1}^{sq} \wedge q_{x2} = q_{x2}^{sq}$. The most interesting rules are the probabilistic branching, the two weakenings, and the two assignments. For the probabilistic branching, following Q:PIF, the preconditions of the two branches are linearly combined using the weights $\frac{1}{3}$ and $1 - \frac{1}{3} = \frac{2}{3}$.

We now discuss the first weakening. The second one generates an identical set of constraints—but the LP solver will give it a different solution. The most interesting constraints are the ones for $Q^{w1} \geq_{(x \geq 2)} Q^{d1}$. This relation is defined by the rule RELAX in Figure 5 and involves finding all the applicable rewrite functions in the logical state $x \geq 2$. As discussed above, F_0, F_1, and F_2 are all applicable, thus the following system of constraints is generated.

$$\begin{pmatrix} q_1^{d1} \\ q_{x0}^{d1} \\ q_{x1}^{d1} \\ q_{x2}^{d1} \end{pmatrix} = \begin{pmatrix} q_1^{w1} \\ q_{x0}^{w1} \\ q_{x1}^{w1} \\ q_{x2}^{w1} \end{pmatrix} - \begin{pmatrix} 1 & -1 & -2 \\ 0 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \wedge \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

The columns of the (4×3) matrix correspond, in order, to F_0, F_1, and F_2. The coefficients (u_i) are fresh names that are local to this weakening.

For the first assignment Q:ASSIGN$_1$, the stable set discussed in Section 4.3 is $\mathscr{S}_{x=x-1} = \{1, |[0, x]|, |[1, x]|\}$. Indeed,

[1] We use Coin-Or's CLP.

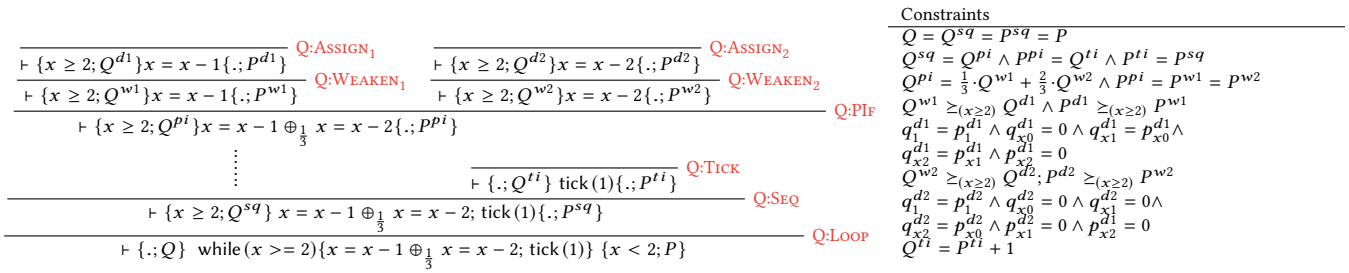

Figure 6. Inference of a derivation using linear constraint solving.

only $|[2,x]|$ is unstable since it becomes $|[3,x]|$ after the assignment $x = x-1$. Since the assignment leaves $\mathbf{1}$ unchanged and changes $|[0,x]|$ into $|[1,x]|$ and $|[1,x]|$ into $|[2,x]|$, the system of constraints generated is

$$\begin{pmatrix} q_1^{d1} \\ q_{x0}^{d1} \\ q_{x1}^{d1} \\ q_{x2}^{d1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} p_1^{d1} \\ p_{x0}^{d1} \\ p_{x1}^{d1} \\ p_{x2}^{d1} \end{pmatrix} \quad \wedge \quad p_{x2}^{d1} = 0$$

or $q_1^{d1} = p_1^{d1} \wedge q_{x0}^{d1} = 0 \wedge q_{x1}^{d1} = p_{x0}^{d1} \wedge q_{x2}^{d1} = p_{x1}^{d1} \wedge p_{x2}^{d1} = 0$.

Solving the Constraints. The LP solver does not only find a solution that satisfies the constraints, it also optimizes a linear objective function. In our case, we would like to find the tightest—i.e, smallest—upper bound on the expected resource consumption. In the implementation, we use an iterative scheme that takes advantage of the incremental solving capabilities of modern LP solvers. Starting at the maximum degree d, we ask the LP solver to minimize the coefficients $(q_i^d)_i$ of all the base functions of degree d. If a solution $(k_i^d)_i$ is returned, we add the constraints $\bigwedge_i q_i^d = k_i^d$ to the linear program. Then, the same scheme is iterated for base functions of degree $d-1, d-2, \ldots, 1$. For our running example, the first objective function for the linear coefficients is $20\cdot q_{x0} + 10\cdot q_{x1} + 1\cdot q_{x2}$. The weight of the coefficients are set to signify facts about the base functions to the LP solver. For instance, q_{x0} gets a smaller weight than q_{x1} because $|[0,x]| \geq |[1,x]|$ for all x. The final solution returned by the LP solver is $q_{x0} = \frac{3}{5}$ and $q_\star = 0$ otherwise. Thus the derived bound is $\frac{3}{5}|[0,x]|$.

6 Soundness of the Analysis

The soundness of the analysis is proved with respect to an operational semantics based on Markov decision processes [75]. It leverages previous work on probabilistic programs by relying on the soundness of a weakest pre-expectation (WP) calculus [63, 83].

6.1 Weakest Pre-expectation Transformer

We can use a WP calculus [44, 72] to express the resource usage of a program $(c, \mathscr{D})$, using an *expected runtime transformer* given in continuation-passing style. Following previous work [63, 83], such a transformer for our language is defined in Table 1. It operates on the set of *expectations* $\mathbb{T} := \{f \mid f : \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}\}$. In the table, $\mathbb{E}_{\mu_R}[h] := \sum_v \mathbb{P}(R = v)\cdot h(v)$ represents the expected value of the random variable h w.r.t the distribution μ_R. $\max\{f_1, f_2\} := \lambda\sigma.\ \max\{f_1(\sigma), f_2(\sigma)\}$. $\mathsf{lfp}\, X.F(X)$ is the least fixed point of the function F. And auxiliary cost transformer $\mathsf{ert}\,[\cdot]^\sharp_X(f)$ is parameterized over another expected cost transformer $X : \mathbb{T} \to \mathbb{T}$. See the TR [75] for details.

Table 1. Expected resource usage transformer.

c	$\mathsf{ert}\,[c, \mathscr{D}](f)$
abort	$\mathbf{0}$
skip	f
tick (q)	$\mathbf{q} + f$
assert e	$[\![e :\ \mathsf{true}]\!]\cdot f$
id $= e$	$f[e/\,\mathsf{id}\,]$
id $= e$ bop R	$\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e \text{ bop } v/\,\mathsf{id}\,]))$
if e c_1 else c_2	$[\![e :\ \mathsf{true}]\!]\cdot \mathsf{ert}\,[c_1, \mathscr{D}](f) + [\![e :\ \mathsf{false}]\!]\cdot \mathsf{ert}\,[c_2, \mathscr{D}](f)$
if $\star$ c_1 else c_2	$\max\{\mathsf{ert}\,[c_1, \mathscr{D}](f),\ \mathsf{ert}\,[c_2, \mathscr{D}](f)\}$
$c_1 \oplus_p c_2$	$p\cdot \mathsf{ert}\,[c_1, \mathscr{D}](f) + (1-p)\cdot \mathsf{ert}\,[c_2, \mathscr{D}](f)$
$c_1; c_2$	$\mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f))$
while e c	$\mathsf{lfp}\, X.([\![e :\ \mathsf{true}]\!]\cdot \mathsf{ert}\,[c, \mathscr{D}](X) + [\![e :\ \mathsf{false}]\!]\cdot f)$
call P	$\mathsf{lfp}\, X.(\mathsf{ert}\,[\mathscr{D}(P)]^\sharp_X)(f)$

More precisely, the transformer $\mathsf{ert}\,[c, \mathscr{D}](f)(\sigma)$ computes the expected number of ticks consumed by the program $(c, \mathscr{D})$ from the input state σ and followed by a computation that has an expected tick consumption given by f. In our case, it is a good intuition to think of expectations as mere potential functions. Olmedo et al. [83] proved the soundness of the $\mathsf{ert}\,(f)$ with respect to an operational model based on Markov decision processes.

6.2 Soundness

We first interpret the pre- and postconditions of the triples as expectations. This interpretation is a function $\mathscr{T}$ that maps $\{\Gamma; Q\}$ to the assertion $\mathscr{T}(\Gamma; Q)$ defined as $\mathscr{T}(\Gamma; Q)(\sigma) := \max(\Gamma(\sigma), \Phi_Q(\sigma))$, where Φ_Q is the potential function associated with the quantitative annotation Q and Γ is lifted as a function on states such that $\Gamma(\sigma)$ is 0 if $\sigma \models \Gamma$ and ∞ otherwise. The soundness of the automatic analysis can now be stated formally w.r.t the WP calculus.

Theorem 6.1 (Soundness of the automatic analysis). *Let c be a command in a larger program $(\_, \mathscr{D})$. If* $\vdash \{\Gamma; Q\}c\{\Gamma'; Q'\}$

is derivable, then $\forall \sigma \in \Sigma$*, the following holds*

$$\mathscr{T}(\Gamma; Q)(\sigma) \geq \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma)$$

Proof. The proof is done by induction on the program structure and the derivation. See the TR [75] for details. □

7 Tail-bound Analysis

One application of our expected resource usage analysis is to combine it with concentration inequalities to bound the probability that the resource usage deviates from some given value. There are many forms of concentration inequalities [34] under various assumptions in probability theory. We focus on two important inequalities—*Markov* and *Chebyshev*—that work well with expected bound analysis.

Markov's Inequality. Let t be the resource usage of a probabilistic program, which is a non-negative random variable. Then Markov's inequality states that for all $a > 0$ the probability of $|t| \geq a$ is bounded by the expectation of $|t|$ divided by a. Since t is non-negative we have

$$\mathbb{P}(t \geq a) \leq \frac{\mathbb{E}(t)}{a}$$

Therefore, by automatically deriving bounds on the expected resource usage, we can bound the probability of a large deviation from the expected resource consumption.

For example, consider the simple random walk from Section 3.1 again with the initial value of $x = n > 0$. Then the expected resource usage is bounded by $2|[0, n]|$. Assume that we want to bound the probability that the resource usage is greater than $10|[0, n]|$. Following the Markov inequality and the derived bound, we have

$$\mathbb{P}(t \geq 10|[0,n]|) \leq \frac{\mathbb{E}(t)}{10|[0,n]|} \leq \frac{2|[0,n]|}{10|[0,n]|} = 0.2$$

Chebyshev's Inequality. If the resource usage t has finite expected value and finite non-zero variance $Var(t)$. Then for all $a \in \mathbb{R}$ such that $a > 0$, Chebyshev's inequality implies

$$\mathbb{P}(|t - \mathbb{E}(t)| \geq a) \leq \frac{Var(t)}{a^2} = \frac{\mathbb{E}((t - \mathbb{E}(t))^2)}{a^2} = \frac{\mathbb{E}(t^2) - \mathbb{E}(t)^2}{a^2}$$

Hence, by deriving a lower bound ℓ on the resource usage t and an upper bound u the squared expected resource usage t^2, we get

$$\mathbb{P}(|t - \mathbb{E}(t)| \geq a) \leq \frac{u - \ell^2}{a^2}$$

It is in general possible to derive such tail bounds with the expected potential method. We can use an auxiliary variable sq_t to encode the square of resource usage, for example by squaring the resource usage t at the exit point of the program. While the potential method generally supports non-linear arithmetic [54], this would have to be implemented with a while loop in the current version of Absynth. Similarly, the potential method can be used to derive lower bounds [76] but this is not yet implemented in Absynth.

We leave a systematic study of deriving tail bounds with the expected potential method for future work.

8 Implementation and Experiments

In this section, we first describe the implementation of the automatic analysis in the tool Absynth. Then we evaluate the performance of our tool on a set of challenging examples.

8.1 Implementation

Absynth is implemented in OCaml and consists of about 5000 LOC. The tool currently works on imperative integer programs written in a Python-like syntax that supports recursive procedures. It also has a C interface based on LLVM. Currently, Absynth supports four common distributions: Bernoulli, binomial, hyper-geometric, and uniform. However, there are no limitations to the distributions that can be supported as long as they have a finite domain.

Potential Functions. To discovery the bounds on expected resource usage automatically, we focus on inferring polynomial potential functions that are *linear combinations* of *base functions* picked among monomials in Absynth. Formally, they are defined by the following syntax.

$$\begin{array}{lll} M & := \mathbf{1} \mid x \mid M_1 \cdot M_2 \mid |[0, P]| & x \in \mathsf{VID} \\ P & := k \cdot M \mid P_1 + P_2 & k \in \mathbb{Q} \end{array}$$

Generating Base and Rewrite Functions. Our analysis can work with every set of base functions. While it would be possible to to fix a set of functions once and for all as in previous work on resource analysis [20, 49], we found that it is more effective to select the base functions for each program using a heuristic [19].

At each program point, Absynth uses abstract interpretation (AI) to infer logical contexts with linear inequalities between program variables. The linear inequalities of the form $\sum_i a_i \cdot x_i + b \geq 0$, where $a_i, b \in \mathbb{Q}$, are used to generate a set of base functions. One can use a more complex and powerful AI such as the Apron library [60]. In practice, we found that our simple AI is sufficient to infer many bounds and provides good performance. For example, if the AI derives $n - x - 1 \geq 0$ at a program point then the heuristic will add the monomials $|[0, n - x]|$ and $|[0, n - x - 1]|$ as base functions. Higher-degree base functions can be constructed by considering powers and products of simpler base functions, e.g., degree 2 base functions such as $|[0, n - x]|^2$ and $|[0, n - x - 1]| \cdot |[0, n - x]|$.

The base functions at a program point are used to generate a set of rewrite functions, for which a Presburger decision procedure is used to reason about the non-negativity of rewrite functions. Recall that a rewrite function is a linear combination of base functions of the form $\sum_i k_i \cdot b_i$, where $k_i \in \mathbb{Q}$. The set of rewrite functions allows to transfer potential to and from the base functions following the

inference rule Q:Weaken. For instance, for the base functions $|[0, n-x]|$ and $|[0, n-x-1]|$, the heuristic adds the linear combination $F = |[0, n-x]| - |[0, n-x-1]| - 1$ as a rewrite function. As shown in Figure 7, F can be used for an assignment $x = x + 1$ when $n - x > 0$ to turn the potential $|[0, n-x]|$ into $|[0, n-x]| + 1$, effectively extracting one unit of constant potential.

User Interaction. Occasionally, when a program requires a complex potential transformation, our heuristic might not be sophisticated enough to identify an appropriate set of rewrite functions. In this case, the user can manually specify a set of rewrite functions as hints to be used by the analysis. These hints, in contrast with typical assertions, have no runtime effect and do not compromise soundness. In particular, before using a rewrite function, the analyzer checks that its non-negativity condition is satisfied.

```
while (x<n) {
  {|[0, n − x]|} ≥
  {|[0, n − x − 1]|+1}
  x = x + 1;
  {|[0, n − x]|+1}
}
```

Figure 7. Rewriting function example.

8.2 Experimental Evaluation

Evaluation Setup. To evaluate the practicality of our framework, we have designed and collected 39 challenging examples with different loop and recursion patterns that depend on probabilistic branching and sampling assignments. In total, the benchmark consists of more than 1000 LOC.

The programs *bayesian* [43], *filling* [85], *race*, *2drwalk*, *robot*, *roulette* [23], and *sampling* [63] have been described in the literature on probabilistic programs, where their expected resource consumption has been analyzed manually. The programs *C4B_**, *prseq*, *prseq*, *preseq_bin*, *prspeed*, *rdseql*, *rdspeed*, and *recursive* are probabilistic versions of deterministic examples from previous work [19, 20, 47]. The other examples are either adaptations of classic randomized algorithms or handcrafted new programs that demonstrate particular capabilities of our analysis. Section 3 contains some representative listings.

To measure the expected resource usage of all examples by simulation, we uniformly chose the range of inputs to be 1000 to 5000 and allowed only 1 input variable to vary while choosing fixed random values for other inputs.[2] We sampled the resource usage 10000 times for each input using the GSL-GNU scientific library [1]. We then compared the results to our statically computed bounds. The simulation is implemented in C++ and consists of more than 5000 LOC.

The experiments were run on a machine with an Intel Core i5 2.4 GHz processor and 8GB of RAM under macOS 10.13.1. The LP solver we use is CoinOr CLP [88].

[2]We reduced the input ranges of polynomial programs by an order of magnitude because their simulation runtime is very long.

Table 2. Automatically-derived bounds on the expected number of ticks with Absynth.

Program	Expected bound	Error(%)	T(s)
Linear programs			
2drwalk	$2\cdot\|[d, n+1]\|$	0.170	2.278
bayesian	$5\cdot\|[0, n]\|$	0	0.272
ber	$2\cdot\|[x, n]\|$	0.026	0.008
bin	$0.2\cdot\|[0, n+9]\|$	0.290	0.281
C4B_t09	$8.27273\cdot\|[0, x]\|$	5.362	0.061
C4B_t13	$1.25\cdot\|[0, x]\| + \|[0, y]\|$	0.009	0.045
C4B_t15	$2\cdot\|[0, x]\|$	A.S	0.044
C4B_t19	$\|[0, k+i+51]\| + 2\cdot\|[100, i]\|$	2.711	0.058
C4B_t30	$0.5\cdot\|[0, x+2]\| + 0.5\cdot\|[0, y+2]\|$	W.C	0.032
C4B_t61	$0.060606\cdot\|[0, l-1]\| + \|[0, l]\|$	0.754	0.028
condand	$\|[0, m]\| + \|[0, n]\|$	A.S	0.010
cooling	$0.42\cdot\|[0, t+5]\| + \|[\text{st}, \text{mt}]\|$	0.192	0.079
fcall	$2\cdot\|[x, n]\|$	0.025	0.008
filling	$0.037037\cdot\|[0, \text{vol}+2]\|+$ $0.333333\cdot\|[0, \text{vol}+10]\|+$ $0.296296\cdot\|[0, \text{vol}+11]\|$	0.713	0.615
hyper	$5\cdot\|[x, n]\|$	0.061	0.013
linear01	$0.6\cdot\|[0, x]\|$	0.036	0.016
miner	$7.5\cdot\|[0, n]\|$	0.071	0.077
prdwalk	$1.14286\cdot\|[x, n+4]\|$	0.128	0.052
prnes	$68.4795\cdot\|[0, -n]\| + 0.052631\cdot\|[0, y]\|$	0.122	0.057
prseq	$1.65\cdot\|[y, x]\| + 0.15\cdot\|[0, y]\|$	0.144	0.057
prseq_bin	$1.65\cdot\|[y, x]\| + 0.15\cdot\|[0, y]\|$	0.150	0.082
prspeed	$2\cdot\|[y, m]\| + 0.666667\cdot\|[x, n]\|$	0.039	0.057
race	$0.666667\cdot\|[h, t+9]\|$	0.294	0.245
rdseql	$2.25\cdot\|[0, x]\| + \|[0, y]\|$	0.007	0.025
rdspeed	$2\cdot\|[y, m]\| + 0.666667\cdot\|[x, n]\|$	0.039	0.040
rdwalk	$2\cdot\|[x, n+1]\|$	0.075	0.012
robot	$0.384615\cdot\|[0, n+6]\|$	R.D	2.658
roulette	$4.93333\cdot\|[n, 10010]\|$	0.282	1.216
sampling	$2\cdot\|[0, n]\|$	0.026	3.347
sprdwalk	$2\cdot\|[x, n]\|$	0.032	0.017
Polynomial programs			
complex	$6\cdot\|[0, m]\|\cdot\|[0, n]\| + 3\cdot\|[0, n]\|+\|[0, y]\|$	0.118	3.415
multirace	$2\cdot\|[0, m]\|\cdot\|[0, n]\| + 4\cdot\|[0, n]\|$	0.703	9.034
pol04	$4.5\cdot\|[0, x]\|^2 + 7.5\cdot\|[0, x]\|$	0.779	0.585
pol05	$\|[0, x]\|^2 + \|[0, x]\|$	0.431	0.353
pol06	$0.625\cdot\|[\min, s]\|+$ $2\cdot\|[\min, s]\|\cdot\|[0, \min]\| + 0.625\cdot\|[\min, s]\|^2$	A.S	7.066
pol07	$1.5\cdot\|[0, n-2]\|\cdot\|[0, n-1]\|$	0.008	4.534
rdbub	$3\cdot\|[0, n]\|^2$	0.106	0.190
recursive	$0.25\cdot\|[l, h]\|^2+1.75\cdot\|[l, h]\|$	0.281	3.791
trader	$5\cdot\|[s_{\min}, s]\|^2 + 5\cdot\|[s_{\min}, s]\|+$ $10\cdot\|[s_{\min}, s]\|\cdot\|[0, s_{\min}]\|$	0.251	4.625

Results. The results of the evaluation are compiled in Table 2. The table is split into linear and non-linear bounds. It contains the inferred bounds, the total time taken by Absynth, and the means (in percentage) of the absolute errors between the measured expected values and the inferred bounds. In general, the analysis finds bounds quickly: Each example is processed in less than 10 seconds. The analysis time mainly depends on three factors: the number of variables in the program, the number of base functions, and the size of the distribution's domain in the sampling commands. The user

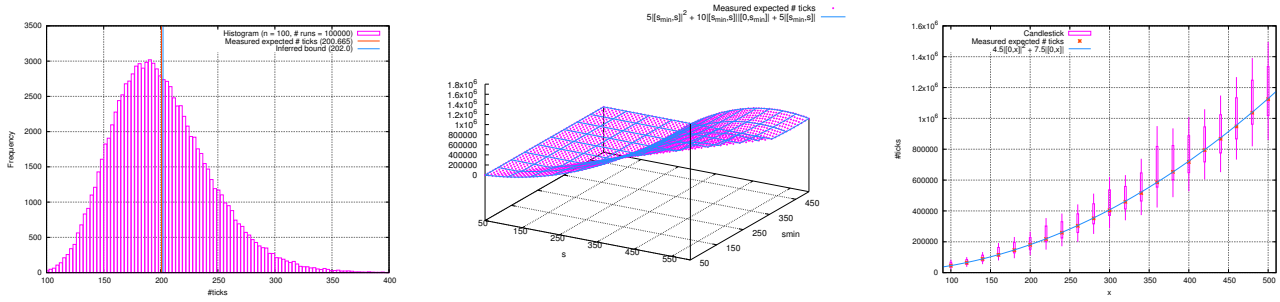

Figure 8. Comparison of automatically derived bounds with measured cost samples. On the left: histogram of the distribution of #ticks for *rdwalk* with $p = \frac{1}{2}, K_1 = 2, K_2 = 1, x = 0$, and $n = 100$. On the right: the inferred bound on the expected #ticks (blue lines) compared to the measured expected values for various input sizes (red crosses) for *trader* (at the center) and *pol04* (on the right). In the latter, the candlesticks represent the highest and lowest sampled values and the second and third quartiles.

can specify a maximal degree of the bounds to control the number of base functions under consideration. Our inference rule for the sampling commands is very precise but the price we pay for the precision is a linear constraint set whose size is proportional to the range of the sampling distribution.

As shown in the *Error* column, the derived bounds are often not only asymptotically tight but also contain very precise constant factors. Figure 8 shows representative plots of comparisons of the inferred bounds and measured cost samples. Our experiments indicate that the computed bounds are close to the measured expected numbers of ticks. See the TR [75] for plots of the other examples.

However, there is no guarantee that Absynth infers asymptotically tight bounds and there are many classes of bounds that Absynth cannot derive. For example, for the programs whose errors are denoted by *A.S* in Table 2, we did not compute asymptotically tight bounds. *C4B_t15* has logarithmic expected cost, thus the best bound that Absynth can derive is a linear bound. Similarly, $|[0,n]|+|[0,m]|$ is the best bound that can be inferred for *condand* whose expected cost is $2{\cdot}\min\{|[0,n]|, |[0,m]|\}$. Another source of imprecise constant factors in the bounds is rounding. The program *robot* has an imprecise constant factor, denoted *R.D* in the table, because it contains a deep nesting of probabilistic choices.

Since we do not assume a particular distribution of the inputs, the bounds on the expected cost have to consider the worst case inputs. If a program does not contain probabilistic constructs then we preform in fact a worst-case analysis. Thus, comparing with the sampled expected cost on the worst-case inputs gives us a very small error even the derived bound is not asymptotically tight. For instance, Absynth derives the loose bound $0.5{\cdot}|[0,x+2]|+0.5{\cdot}|[0,y+2]|$ for *C4B_t30* whose expected cost is $0.5{\cdot}|[0,2{\cdot}(\min\{x,y\}+2)]|$. If we compare the derived bound with the sampled expected cost on the worst-case inputs (e.g., values of x and y such that $x = y$), then we obtain a very small error. We mark the error with W.C in this case.

9 Related Work

Our work is a confluence of ideas from automatic resource bound analysis and analysis of probabilistic programs. They have been extensively studied but developed independently. In spite of abundant related research, we are not aware of existing techniques that can automatically derive symbolic bounds on the expected runtime of probabilistic programs.

Resource Bound Analysis. Most closely related to our work is prior work on AARA for deterministic programs. AARA has been introduced in [55] for automatically deriving linear worst-case bounds for first-order functional programs. The technique has been generalized to derive polynomial bounds [50, 52, 53, 57, 58], lower bounds [76], and to handle (strictly evaluated) programs with arrays and references [70], higher-order functions [51, 61], lazy functional programs [86, 90], object-oriented programs [56, 59], and user defined data types [51, 62]. It also has been integrated into separation logic [6] and proof assistants [25, 81]. A distinctive common theme of sharing is compositionality and automatic bound inference via LP solving.

In contrast to our work, all prior research on AARA targets deterministic programs and derives worst-case bounds rather than bounds on the expected resource usage. In our formulation of AARA for probabilistic programs, we build on prior work that integrated AARA into Hoare logic to derive bounds for imperative code [18–20, 80], a new technique for deriving polynomial bounds on the expected resource usage of programs with probabilistic sampling and branching.

Beyond AARA there exists many other approaches to automatic worst-case resource bound analysis for deterministic programs. They are based on sized types [89], linear dependent types [68, 69], refinement types [29, 92], annotated type systems [31, 32], defunctionalization [7], recurrence relations [3, 4, 13, 33, 38, 46, 66], abstract interpretation [14, 22, 47, 87, 91], template based assume-guarantee reasoning [71], measure functions [27], and techniques from

term rewriting [8, 17, 39, 82]. These techniques do not apply to probabilistic programs and do not derive bounds on expected resource usage.

The decision to base our analysis on AARA is mainly motivated by the strong connection to existing techniques for (manually) analyzing expected runtime (see next paragraph) and the general advantages of AARA, including compositionality, tracking of amortization effects, flexible cost models, and efficient bound inference using LP solving.

We are only aware of few works that study the analysis of expected resource usage of probabilistic programs. Chatterjee et al. [28] propose a technique for solving recurrence relations that arise in the analysis of expected runtime cost. Their technique can derive bounds of the form $O(\log n)$, $O(n)$, and $O(n \log n)$. Similarly, Flajolet et al. [37] describe an automatic for average-case analysis that is based on generating functions and that can be seen as a method for solving recurrences. While these techniques apply to recurrences that describe the resource usage of randomized algorithms, the works do not propose a technique for deriving recurrences from a program. It is therefore not a push-button analysis for probabilistic programs but complementary to our work since they can derive logarithmic bounds.

Analysis of Probabilistic Programs. Considering work on analyzing probabilistic programs, most closely related is a recent line of work by Kaminski et al. [63, 83]. The goal of this work is to characterize the expected runtime of probabilistic programs. However, they use a WP calculus to derive pre-expectations and do not consider any automation. The technique can be seen as a generalization of quantitative Hoare logic [18, 20] for AARA to the probabilistic setting but does not provide support for automatic reasoning. In fact, when attempting to generalize AARA to probabilistic programs we were first unaware of the existing work and rediscovered some of the proof rules. Our contributions are new specialized proof rules that allow for automation using LP solving and a prototype implementation of the new technique. While our soundness proof is original, it leverages the proof by Kaminski et al. by relying on the soundness of the rules for weakest preconditions.

The use of pre-expectations for reasoning about probabilistic programs dates back to the pioneering work of Kozen and others [21, 67, 72]. It has been automated using constraint generation [65] and abstract interpretation [24] to derive quantitative invariants. However, it is unclear how to use them to automatically derive symbolic (polynomial) bounds like in our work.

The recent work of Batz et al. [12] also has the goal of automatically deriving expected runtime bounds using the WP calculus. However, the scope of the work and the techniques used are quite different: Batz et al. derive constant bounds for Bayesian networks that correspond to loop-free programs with finite states; a computationally hard yet decidable problem. In contrast, this works uses constraint solving to derive symbolic bounds for programs with loops and recursion, which is in general undecidable.

Another body of research relies on probabilistic pushdown automata and martingale theory to analyze the termination time [16] and the expected number of steps [35]. The use of martingale theory to automatically analyze probabilistic programs has been pioneered in [23]. While their technique also relies on linear constraints, it is proving almost-sure termination instead of resource bounds and relies on Farka's lemma. More general methods [26] are able to synthesize polynomial ranking-supermartingales for proving termination.

Abstract interpretation has also been applied to probabilistic programs [30, 73, 74] but we are not aware of its application to derive bounds on the expected resource usage. Another approach to automatically analyze probabilistic programs is based on symbolic inference [40] and analyzing execution paths with statistical techniques [15, 41, 77–79, 85]. In the context of analyzing differential privacy, there are works with limited automation that focus on deriving bounds on the privacy budget for probabilistic programs [10, 48].

10 Conclusion

We have introduced a new technique for automatically inferring polynomial bounds on the expected resource consumption of probabilistic programs. The technique is a combination of existing manual quantitative reasoning for probabilistic programs and an automatic worst-case bound analysis for deterministic programs. The effectiveness of the technique is demonstrated with an implementation and the automatic analysis of challenging examples from previous work.

In the future, we plan to study how to build on the introduced technique to automatically derive tail bounds, that is, worst-case bounds that hold with high probability. We are also working on a more direct soundness argument that also works for non-monotone resources. Finally, we plan to build on Resource Aware ML [51] to apply the expected potential method to (higher-order) functional programs.

Acknowledgments

This article is based on research supported by AFRL under DARPA STAC award FA8750-15-C-0082 and by the Eric and Wendy Schmidt Fund for Strategic Innovation. 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

[1] [n. d.]. GSL-GNU Scientific Library. https://www.gnu.org/software/gsl/. ([n. d.]). Accessed: 2017-10-19.

[2] [n. d.]. Mathematics for Computer Science. MIT Course Number 6.042J/18.062J. ([n. d.]). Accessed: 2018-03-28.

[3] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*.

[4] Diego Esteban Alonso-Blas and Samir Genaim. 2012. On the Limits of the Classical Approach to Cost Analysis. In *19th Int. Static Analysis Symp. (SAS'12)*.

[5] R. B. Ash and C. Doléans-Dade. 2000. *Probability and Measure Theory*. Academic Press.

[6] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*.

[7] Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2012. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

[8] Martin Avanzini and Georg Moser. 2013. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*.

[9] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. *Formal Certification of Randomized Algorithms*. Technical Report. http://justinh.su/files/papers/ellora.pdf.

[10] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society.

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages (POPL'09)*. ACM, New York, NY, USA.

[12] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. 2018. How long, O Bayesian network, will I sample thee? A program analysis perspective on expected sampling times. *ArXiv e-prints* (Feb. 2018). arXiv:cs.PL/1802.10433 To appear at ESOP'18.

[13] Ralph Benzinger. 2004. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.* 318, 1-2 (2004).

[14] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*.

[15] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Pasareanu, and Willem Visser. 2014. Compositional solution space quantification for probabilistic software analysis. In *Conference on Programming Language Design and Implementation (PLDI'14)*.

[16] Tomás Brázdil, Stefan Kiefer, Antonín Kucera, and Ivana Hutarová Vareková. 2015. Runtime analysis of probabilistic programs with unbounded recursion. *J. Comput. Syst. Sci.* 81, 1 (2015). https://doi.org/10.1016/j.jcss.2014.06.005

[17] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*.

[18] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *35th Conference on Programming Language Design and Implementation (PLDI'14)*. Artifact submitted and approved.

[19] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *29th International Conference on Computer-Aided Verification (CAV'17)*.

[20] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*. Artifact submitted and approved.

[21] Orieta Celiku and Annabelle McIver. 2005. Compositional Specification and Analysis of Cost-Based Properties in Probabilistic Programs. In *Formal Methods, International Symposium of Formal Methods Europe (FM'05)*.

[22] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. 2015. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*.

[23] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis using Martingales. In *Computer-Aided Verification (CAV'13) (Lecture Notes in Computer Science)*, Vol. 8044. Springer-Verlag.

[24] Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants as Fixed Points of Probabilistic Programs. In *Static Analysis Symposium (SAS'14) (Lecture Notes in Computer Science)*, Vol. 8723. Springer-Verlag.

[25] Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*.

[26] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference (CAV'16)*.

[27] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017. Non-polynomial Worst-Case Analysis of Recursive Programs. In *Computer Aided Verification - 29th Int. Conf. (CAV'17)*.

[28] Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. 2017. Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds. In *Computer Aided Verification - 29th Int. Conf. (CAV'17)*.

[29] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*.

[30] Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *Programming Languages and Systems - 21st European Symposium on Programming (ESOP'12)*.

[31] Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*.

[32] Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*.

[33] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2012. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

[34] D. Dubhashi and A. Panconesi. 2009. Concentration of Measure for the Analysis of Randomized Algorithms. *Cambridge University Press* (2009).

[35] Javier Esparza, Antonín Kucera, and Richard Mayr. 2005. Quantitative Analysis of Probabilistic Pushdown Automata: Expectations and Variances. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*.

[36] Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 13. https://doi.org/10.1145/2676726.2677001

[37] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. 1991. Automatic Average-case Analysis of Algorithms. *Theoret. Comput. Sci.* 79, 1 (1991).

[38] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*.

[39] Florian Frohn, M. Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. 2016. Lower Runtime Bounds for Integer Programs. In *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*.

[40] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON,*

Canada, July 17-23, 2016, Proceedings, Part I.

[41] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA'12)*.

[42] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521 (2015).

[43] 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 (FOSE'14)*.

[44] F. Gretz, J. Katoen, and A. McIver. 2014. Operational versus Weakest Pre–Expectation Semantics for the Probabilistic Guarded Command Language. *Performance Evaluation* 73 (2014).

[45] G. Grimmett and D. Stirzaker. 1992. *Probability and Random Processes*. Oxford University Press.

[46] Bernd Grobauer. 2001. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*.

[47] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*.

[48] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, 1.

[49] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*.

[50] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.* (2012).

[51] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[52] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*.

[53] Jan Hoffmann and Zhong Shao. 2014. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*.

[54] Jan Hoffmann and Zhong Shao. 2015. Type-Based Amortized Resource Analysis with Integers and Arrays. *J. Funct. Program.* (2015).

[55] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*.

[56] Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*.

[57] Martin Hofmann and Georg Moser. 2014. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*.

[58] Martin Hofmann and Georg Moser. 2015. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*.

[59] Martin Hofmann and Dulma Rodriguez. 2013. Automatic Type Inference for Amortised Heap-Space Analysis. In *22nd Euro. Symp. on Prog. (ESOP'13)*.

[60] B. Jeannet and A. Miné. 2009. APRON: A library of numerical abstract domains for static analysis. In *Proceedings Computer Aided Verification CAV'2009*. LNCS.

[61] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*.

[62] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*.

[63] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP'16)*. Springer.

[64] Joost-Pieter Katoen. 2016. The Probabilistic Model Checking Landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*.

[65] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *Static Analysis - 17th International Symposium (SAS'10)*.

[66] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *Conference on Programming Language Design and Implementation (PLDI'17)*.

[67] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981). https://doi.org/10.1016/0022-0000(81)90036-2

[68] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*.

[69] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*.

[70] Benjamin Lichtman and Jan Hoffmann. 2017. Arrays and References in Resource Aware ML. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)*.

[71] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th Symposium on Principles of Programming Languages (POPL'17)*.

[72] Annabelle McIver and Carroll Morgan. 2004. *Abstraction, Refinement and Proof For Probabilistic Systems (Monographs in Computer Science)*. Springer Verlag.

[73] David Monniaux. 2001. Backwards Abstract Interpretation of Probabilistic Programs. In *Programming Languages and Systems, 10th European Symposium on Programming (ESOP'01)*.

[74] David Monniaux. 2005. Abstract interpretation of programs as Markov decision processes. *Sci. Comput. Program.* 58, 1-2 (2005).

[75] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2017. Bounded Expectations: Resource Analysis for Probabilistic Programs. *CoRR* abs/1711.08847 (2017). arXiv:1711.08847 http://arxiv.org/abs/1711.08847

[76] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*.

[77] Van Chan Ngo and Axel Legay. 2018. PSCV: A Runtime Verification Tool for Probabilistic SystemC Models. *J. of Software: Evolution and Process* (2018).

[78] Van Chan Ngo, Axel Legay, and Vania Joloboff. 2016. PSCV: A Runtime Verification Tool for Probabilistic SystemC Models. In *28th International Conference on Computer Aided Verification (CAV'16)*.

[79] Van Chan Ngo, Axel Legay, and Jean Quilbeuf. 2016. Statistical Model Checking for SystemC Models. In *17th IEEE High Assurance Systems Engineering Symposium (HASE'16)*.

[80] Hanne Riis Nielson. 1987. A Hoare-Like Proof System for Analysing the Computation Time of Programs. *Sci. Comput. Program.* 9, 2 (1987). https://doi.org/10.1016/0167-6423(87)90029-3

[81] Tobias Nipkow. 2015. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*.

[82] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning* 51, 1 (2013).

[83] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on*

Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016.

[84] Avi Pfeffer. 2016. *Practical Probabilistic Programming*. Manning. https://books.google.com/books?id=qyfksgEACAAJ

[85] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *ACM conference on Programming Languages Design and Implementation (PLDI'13)*. ACM Press.

[86] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*.

[87] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*.

[88] The CLP Team. 2018. CLP. https://projects.coin-or.org/Clp. (2018).

[89] Pedro Vasconcelos. 2008. *Space Cost Analysis Using Sized Types*. Ph.D. Dissertation. School of Computer Science, University of St Andrews.

[90] Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*.

[91] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*.

[92] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*.