Quantitative Bounds on Resource Usage of Probabilistic Programs

KRISHNENDU CHATTERJEE, Institute of Science and Technology Austria (ISTA), Austria
AMIR KAFSHDAR GOHARSHADY, Hong Kong University of Science and Technology, Hong Kong
TOBIAS MEGGENDORFER, Lancaster University Leipzig, Germany
ĐORĐE ŽIKELIĆ*, Singapore Management University, Singapore

Cost analysis, also known as resource usage analysis, is the task of finding bounds on the total cost of a program and is a well-studied problem in static analysis. In this work, we consider two classical quantitative problems in cost analysis for probabilistic programs. The first problem is to find a bound on the expected total cost of the program. This is a natural measure for the resource usage of the program and can also be directly applied to average-case runtime analysis. The second problem asks for a tail bound, i.e. given a threshold t the goal is to find a probability bound p such that $\mathbb{P}[\text{total cost} \geq t] \leq p$. Intuitively, given a threshold t on the resource, the problem is to find the likelihood that the total cost exceeds this threshold.

First, for expectation bounds, a major obstacle in previous works on cost analysis is that they can handle only non-negative costs or bounded variable updates. In contrast, we provide a new variant of the standard notion of cost martingales, that allows us to find expectation bounds for a class of programs with general positive or negative costs and no restriction on the variable updates. More specifically, our approach is applicable as long as there is a lower bound on the total cost incurred along every path.

Second, for tail bounds, all previous methods are limited to programs in which the expected total cost is finite. In contrast, we present a novel approach, based on a combination of our martingale-based method for expectation bounds with a quantitative safety analysis, to obtain a solution to the tail bound problem that is applicable even to programs with infinite expected cost. Specifically, this allows us to obtain runtime tail bounds for programs that do not terminate almost-surely.

In summary, we provide a novel combination of martingale-based cost analysis and quantitative safety analysis that is able to find expectation and tail cost bounds for probabilistic programs, without the restrictions of non-negative costs, bounded updates, or finiteness of the expected total cost. Finally, we provide experimental results showcasing that our approach can solve instances that were beyond the reach of previous methods.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Software verification**; **Formal software verification**; • **Theory of computation** → **Probabilistic computation**; **Program analysis**; **Program verification**.

Additional Key Words and Phrases: Probabilistic Programming, Static Analysis, Quantitative Bounds, Cost Analysis, Martingales

*Part of the work was done while the author was at the Institute of Science and Technology Austria (ISTA).

Authors' addresses: Krishnendu Chatterjee, Institute of Science and Technology Austria (ISTA), Austria, krishnendu.chatterjee@ist.ac.at; Amir Kafshdar Goharshady, Hong Kong University of Science and Technology, Hong Kong, goharshady@cse.ust.hk; Tobias Meggendorfer, Lancaster University Leipzig, Germany, tobias@meggendorfer.de; Đorđe Žikelić, Singapore Management University, Singapore, dzikelic@smu.edu.sg.

ACM 2475-1421/2024/4-ART107
https://doi.org/10.1145/3649824

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

1 INTRODUCTION

Probabilistic programs Probabilistic programs extend classical programs with randomization and allow specification and automated inference in expressive probabilistic models [Ghahramani 2015; Gordon et al. 2014; van de Meent et al. 2018]. They have been used in the analysis of randomized algorithms [Barthe et al. 2018], cryptographic protocols [Barthe et al. 2009], machine learning [Ghahramani 2015; Roy et al. 2008], and robotics [Thrun 2000]. There is a significant number of probabilistic programming languages such as Anglican [Tolpin et al. 2016], Church [Goodman et al. 2008], or Pyro [Bingham et al. 2019] and their static analysis is a very active research topic. Probabilistic programs may also be extended with non-determinism in order to model unknown user inputs, interactions with an unknown environment or enabling abstraction towards simplifying their static analysis [McIver and Morgan 2005].

Cost analysis in probabilistic programs Cost analysis is concerned with computing bounds on the total cost, or resource usage, of a program. It has a multitude of important applications such as reasoning about program runtime, memory usage, energy consumption, or the total reward. Cost analysis for non-probabilistic programs is a well-studied problem that is mainly concerned with deriving worst-case bounds on cost of a program execution over a set of inputs [Carbonneaux et al. 2015; Gulwani et al. 2009; Hoffmann and Hofmann 2010; Sinn et al. 2014]. For probabilistic programs, there are two natural quantitative extensions of the cost analysis problem:

(1) ***Finding expectation bounds:*** Let $Cost$ denote a random variable defined as the total cost of a random run/execution in a given probabilistic program. Compute an upper bound on the expected value $\mathbb{E}[Cost]$ of the total cost usage of a random program run under every resolution of non-determinism.
(2) ***Finding tail bounds:*** Given a real-valued threshold t, compute a probability $p \in [0, 1]$ such that the tail bound $\mathbb{P}[Cost \geq t] \leq p$ is satisfied under every resolution of non-determinism.

Both problems above analyze important information about cost/resource usage in probabilistic programs. The expected value of total cost provides an aggregated summary of the probability distribution of the cost and may be used to extract non-trivial properties about the probability distribution. Tail bounds aim to estimate the likelihood of the total cost being larger than some tolerable threshold and are particularly relevant for resource-critical applications.

Previous approaches Earlier works on cost analysis in probabilistic programs mainly focused on deriving expectation and tail bounds on termination time [Batz et al. 2018; Chatterjee et al. 2016, 2018; Kaminski et al. 2018; Kura et al. 2019]. This is equivalent to assigning a unit cost to every step of execution. More recently, [Ngo et al. 2018; Wang et al. 2020b] proposed methods for cost analysis in programs with more general types of cost. However, the incurred costs are required to be non-negative. In [Wang et al. 2019], this assumption was further relaxed by introducing a method for computing bounds on the expected cost usage for programs that either (i) have non-negative costs or (ii) in which all variable updates have bounded size. Finally, [Wang et al. 2021] proposes a method for deriving tail bounds, central bounds and higher-order moment bounds on cost usage, where programs are also assumed to either have non-negative costs or bounded variable updates. Some previous works do lift these assumptions in order to derive expectation cost bounds, however only for restrictive sublcasses of programs such as programs with bounded loops [Gehr et al. 2016] and prob-solvable loops (i.e. loops whose body consists of a sequence probabilistic assignments

but with no conditional branching) [Bartocci et al. 2019]. Moreover, all these works only consider programs that are almost-surely terminating, i.e. programs that terminate with probability 1 for all resolutions of non-determinism.

Limitations of previous approaches While all works discussed above present significant advances in cost analysis of probabilistic programs, they have the following limitations:

(1) ***Expectation bounds:*** The underlying assumptions of the existing methods for general programs restrict their applicability to programs in which either (i) all incurred costs are non-negative or (ii) all variable updates are bounded. This means that no existing method can be applied to general programs with *general costs and unbounded variable updates*.
(2) ***Tail bounds:*** The existing methods for the tail bound problem first compute an upper bound on either the expected cost usage [Chatterjee et al. 2016] or one of its higher moments, i.e. $\mathbb{E}[Cost^k]$ for some integer $k \geq 1$ [Kura et al. 2019; Wang et al. 2021], and then apply concentration bounds. Thus, to solve the tail bound problem, these works essentially first solve the expected value or a higher moment bound problem. As finite higher moments also imply finiteness of the expected value, this means that the existing methods cannot compute tail bounds for programs with *infinite expected cost usage*. As an important special case, given a program whose expected termination time is infinite, e.g. any program that does not terminate almost-surely, no existing method can be used to derive tail bounds on its termination time.

Our approach and contributions Our goal in this work is to close the gaps in cost analysis of probabilistic programs that were discussed above. To that end, we present novel methods for both the expectation bounds problem and the tail bounds problem that overcome these limitations:

(1) ***Expectation bounds:*** We propose a strengthened variant of the notion of *cost supermartingales* that were introduced in [Wang et al. 2019]. Cost supermartingales were the first method for the expectation bound problem that could handle negative costs. However, their applicability is restricted to almost-surely terminating programs in which either (a) all incurred costs are non-negative, or (b) all variable updates are bounded. In this work we prove that, if we strengthen the definition of cost supermartingales to be *non-negative* at every reachable program state, then the restriction to (a) and (b) can be lifted. Hence, their applicability is extended to almost-surely terminating programs that are only required to satisfy the so-called *lower-bounded total cost condition*. This condition only requires that the total incurred cost along every program execution is bounded from below.
(2) ***Tail bounds:*** We present a new method which is applicable to programs with general costs, unbounded variable updates and both finite or infinite expected total cost. The key idea behind our method is to not analyze the cost usage over the whole program, but to restrict the analysis to a core subset of the program states that is left with low probability and hence sufficiently captures the overall behavior of the program. This part of the program is computed in the form of a stochastic invariant. A *stochastic invariant* [Chatterjee et al. 2017] is a pair (SI, p_{SI}) of a set of states SI in the program and a probability p_{SI} of ever leaving the set SI. Our method computes a stochastic invariant (SI, p_{SI}) simultaneously with a *non-negative cost supermartingale* which achieves a tail bound on cost usage of program runs that either never leave SI or terminate immediately upon leaving SI. This tail bound is then combined with the probability p_{SI} of leaving SI in order to derive a tail bound on cost usage in the whole program.
Crucially, the stochastic invariant and the cost supermartingale are computed *simultaneously*, i.e. our approach obtains a single set of constraints that encode all requirements of both the invariant and the supermartingale and passes it to an external solver. Doing this sequentially, i.e. first finding a stochastic invariant and then attempting to synthesize a non-negative cost

supermartingale wrt to this fixed invariant, might not succeed since the computed invariant might not be tight enough and may have infinite cost.

(3) ***Automation:*** For linear/polynomial programs, we present *full automation* of our methods by employing a template-based synthesis approach. In particular, for the tail bounds problem, our method synthesizes the stochastic invariant and the non-negative cost supermartingale *simultaneously* while constraining the synthesis to satisfy the desired tail bound.

(4) ***Experimental results:*** For both analyses, we show, both theoretically and by experiment, that our approach handles instances that were beyond the reach of previous methods. Specifically, our experimental results include programs with general costs and unbounded updates. They also include examples of tail bounds obtained on the runtime of programs that do not terminate almost-surely and have infinite expected termination time. Finally, we consider examples from the blockchain analysis literature to demonstrate the practical relevance of closing the gaps in cost analysis of probabilistic programs that were discussed above. In particular, we have identified example programs containing both positive and negative costs, unbounded updates, having infinite expected total cost and lacking almost-sure termination. We show that our approach is able to derive both expectation and tail bounds for these examples, thus providing the first approach being able to do so.

Limitations As shown above, our approach significantly extends previous methods for cost analysis. However, it still has the following limitations, lifting which is an interesting topic of future work:

- ***Lower-bounded total cost:*** Our proof of soundness of non-negative cost supermartingales (Thereom 5.2) depends on the lower-bounded total cost assumption. So, our approach is not applicable to every probabilistic program. Finding proof concepts for expectation bounds in general probabilistic programs with no extra assumptions remains a challenging open problem, as then the conditions required for martingale-based analysis are not always satisfied. Also note that lower-bounded total cost is theoretically incomparable with the requirement of having bounded updates. As such, the family of programs that can be handled by our approach is neither a subset nor a superset of those handled by [Wang et al. 2019].
- ***Tightness of tail bounds:*** Our approach for tail bound analysis consists of two parts: (i) a quantitative safety analysis that finds a core subset of the program modeled by the stochastic invariant, and (ii) an expectation bound on the total cost incurred by runs that stay within this core subset as long as they have not terminated. In part (ii), we use Markov's inequality which is not tight but is applicable in the general case. An interesting direction of future work is to consider special cases such as programs with bounded updates or non-negative costs and, for these subclasses, combine our approach with either (i) expectation bound approaches that bound higher moments of the total cost, such as [Kura et al. 2019; Wang et al. 2021], or (ii) approaches based on Azuma and Hoeffding's inequalities [Chatterjee et al. 2016], to obtain tighter tail bounds.
- ***Automation:*** We automate our approach using a template-based method that relies on Farkas' Lemma [Avis and Kaluzny 2004; Farkas 1902] and Positivstellensätze [Handelman 1988; Putinar 1993]. Similar methods have previously been used for termination analysis [Asadi et al. 2021; Chatterjee et al. 2016, 2021] and invariant generation [Chatterjee et al. 2020]. However, they are only applicable to imperative programs with real variables in which every assignment's RHS is a polynomial expression in terms of program variables and every loop/branch guard is a boolean combination of polynomial inequalities. While our mathematical results are general, automating our approach over more expressive families of programs is an open problem.

2 MOTIVATING EXAMPLES

In this section, we underline the practical relevance of lifting the limitations identified in the previous section by providing several example use-cases modeling real-world blockchain scenarios. One of the main application domains mentioned in [Wang et al. 2019] is the analysis of probabilistic Bitcoin mining protocols. The following examples are natural extensions of those in [Wang et al. 2019]. In every case, our approach is able to find expectation/tail bounds, but the programs do not satisfy non-negative cost, bounded update or finite total cost conditions. Hence, previous methods are inapplicable. See [Chatterjee et al. 2024, Appendix J] for more details.

```
while x ≥ α do
  x ≔ x − α
  tick(α)
  if prob(p) then
    if prob(p′) then
      tick(−β)
    else if ⋆ then
      tick(−β)
```

```
while x/y ≥ m do
  if prob(x/y) then
    if prob(p) then
      tick(α + β ∗ y)
      x ≔ x + α + β ∗ y
    y ≔ y + α + β ∗ y
```

Fig. 1. Bitcoin Mining [Wang et al. 2019] (left) and Proof-of-stake Mining (right).

Tail Bounds – Proof-of-stake cryptocurrency mining. Bitcoin is based on a proof-of-work protocol in which miners have to compete in solving a hard computational puzzle by repeatedly trying random candidate hashes. This was modeled as a probabilistic program, Figure 1 (left), in [Wang et al. 2019]. A miner starts with an initial balance of x and keeps performing proof-of-work as long as they can pay for the electricity costs α. The command $\mathbf{tick}(\alpha)$ denotes that a cost of α was incurred for electricity. For each block, the miner has a probability p of successfully solving the puzzle. p is constant and a function of the miner's computational power. If successful, they have to publish their block. There is a high probability p' for the block to be accepted by other nodes and added to the blockchain, leading to a reward of β for its miner*. However, there is also a small probability $1 - p'$ that a different miner solves the puzzle at approximately the same time. In this case, one of the new blocks will eventually end up in the consensus chain, but the process is non-deterministic.

In this example, the expected total cost of the program is bounded. This is a requirement for the soundness of all previous approaches that obtain tail bounds. However, this assumption does not hold if we model newer proof-of-stake protocols [Gilad et al. 2017; Kiayias et al. 2017] used in cryptocurrencies such as Ethereum, Cardano and Tezos. In such currencies, the probability of a miner being selected to add the next block is no longer constant, but instead proportional to the number of coins owned by this miner. Thus, rewards obtained in one iteration can increase the probability of success in future iterations. Additionally, since each block adds new coins, there is a multiplicative inflation factor that has to be considered in any probabilistic program modeling the mining. Thus, the updates to program variables are not bounded, either.

Specifically, proof-of-stake mining can be modeled as a probabilistic program shown in Figure 1 (right). We consider a miner who owns x coins out of a total of y. They can take part in mining if their stake is at least m. Their probability of being chosen to add the next block is proportional to their stake, i.e. x/y. This is in contrast to Bitcoin, in which the probability is constant. When the miner proposes the new block, there are validators (usually a committee of other miners), who can accept or reject this block. If the miner follows the protocol correctly, their block will be accepted

*Currently, the block reward in Bitcoin is 6.25 BTC.

with overwhelming probability p. After the addition of each new block, the total number of coins y increases by a preset multiplicative inflation factor β and an additive factor α. The newly created coins are always paid to the miner who added the block. Thus, if our miner is successful, their number of coins, x, also increases by $\alpha + \beta \cdot y$. This is of course a reward as well, and is modeled by the **tick** operation[†]. The total expected reward is not bounded as the miner is expected to earn more in each iteration.

```
while y ≥ 1 do
  tick(α ∗ y)
  i := 1
  while i ≤ y do
    if prob(p) then
      if prob(p′) then
        tick(−β)
      else if ⋆ then
        tick(−β)
      i := i + 1
  y := y + (−1, 0, 1) : (0.5, 0.1, 0.4)
```

```
while y ≥ 1 do
  tick(α ∗ y)
  i := 1
  while i ≤ y do
    if prob(p) then
      if prob(p′) then
        tick(−β)
      else if ⋆ then
        tick(−β)
      i := i + 1
  y := y ∗ (0.95, 1, 1.05) : (0.5, 0.1, 0.4)
```

Fig. 2. Bitcoin pool mining [Wang et al. 2019] (left) and its multiplicative variant (right).

Expectation bounds – Pool mining. [Wang et al. 2019] also considers an example program modeling Bitcoin pool mining. A *pool* is a collection of y cooperating miners who pool their resources for proof-of-work computations together in order to reduce the variance in their income [Lewenberg et al. 2015]. It also has a manager who takes a portion of the income. The income of the pool's manager can be modeled by a probabilistic program as shown in Figure 2 (left). For each block, the manager has to pay a fee of α to each participating miner[‡]. This is modeled by **tick**$(\alpha * y)$. Then, there is an inner loop which mimicks the Bitcoin mining example of Figure 1 (left) for each miner, except that the reward β is paid to the manager. Finally, the size of a pool changes dynamically (last line) as miners migrate between pools. This program contains both positive and negative costs, but every update to any variable changes it by a bounded amount. Specifically, y increases or decrease by at most 1. Such bounded updates are a requirement for the soundness of the approach in [Wang et al. 2019], as well as all other previous methods for expectation bounds, when we have both positive and negative costs. All previous approaches are unsound for a program having a mixture of positive and negative costs, as well as unbounded updates.

We argue that the program used in Figure 2 (left) is unrealistic. Specifically, the updates to y are kept bounded with no real-world justification. Recall that y is the total number of miners taking part in the pool. Moreover, each iteration of the topmost loop corresponds to a fixed amount of time, i.e. a single block in the blockchain which takes almost 10 minutes in Bitcoin. Thus, it is unrealistic to assume that the number of miners who migrate from/to the pool is constant and independent of the pool's size. In practice, we expect the change to be multiplicative. Intuitively, a low-performing pool will not lose a constant number of miners irrespective of the size of the pool, but will of course lose a fraction of its miners. To that end, we provide a more realistic multiplicative variant in Figure 2 (right) which contains *both positive and negative costs* and *unbounded updates* in its last line. Thus, no previous approach is sound for this program. See [Chatterjee et al. 2024, Appendix J] for more details.

[†]We model rewards as positive costs and find bounds on the total reward.

[‡]We assume all miners have unit power. A large miner can be considered as a set of unit-powered miners.

Tail bounds – Block withholding. Our last example also comes from Bitcoin mining. Bitcoin pools are known to employ dishonest techniques to stifle their competition. One of these techniques is called *block withholding* [Bag et al. 2016; Haghighat and Shajari 2019]. This is achieved by a pool signing up as a miner in a competing pool and sharing in the revenues but not actually contributing to the earnings, i.e. withholding valid proof-of-work solutions instead of publishing them. We show that this classical attack can be modeled as a probabilistic program with infinite expected cost. Intuitively, the cost is infinite since the attacking pool is losing revenue in every block. However, the attackers are interested in the probability of being able to bankrupt their victim before they themselves run out of money. This is of course a tail bound. Since all previous approaches for obtaining tail bounds require the assumption of bounded expected total cost, none of them are applicable to this example. The details can be found in [Chatterjee et al. 2024, Appendix J].

3 OVERVIEW OF OUR APPROACH

3.1 Illustrative Examples

Before presenting technical details, in this section we provide an overview of our approach on two simple illustrative examples shown in Figure 3. The program in Figure 3a contains both positive and negative costs, as well as unbounded variable updates. The program in Figure 3b has infinite expected cost usage. Thus, these illustrative examples are beyond the reach of previous methods.

Example 3.1. The program in Figure 3a contains two variables x and y, initialized to $x = y = 1$. In each loop iteration, x is incremented by a value that is sampled uniformly at random from the continuous interval $[-1, 0.5]$. Then, with probability 0.5, the value of y is either halved or increased by half its value. The program terminates once a state with $x < 1$ is reached. Note that this program terminates almost-surely since x decreases by 0.25 in expectation in every loop iteration. On the other hand, we always have $y \geq 0$ since halving a non-negative value yields a non-negative value. We are interested in deriving tail bounds on the value of y upon termination. Thus, we specify a cost model that incurs cost whenever y is modified, indicated in program syntax by **tick** commands. Note that incurred costs may be both positive and negative and that y does not have bounded updates, since a state with arbitrarily large value of y may be reached with positive probability.

Example 3.2. The program in Figure 3b (taken from [Chatterjee et al. 2022, Figure 1]) contains a variable x initialized to $x = 0$ and it terminates once x becomes negative. In each loop iteration, if $x < 200$ then x is incremented by a value that is sampled uniformly at random from the continuous interval $[-1, 0.5]$, otherwise another increment is sampled from $[-1, 2]$. Note that this program does not terminate almost-surely, as it reaches a state with $x \geq 200$ and then diverges with positive probability due to the probabilistic update of x having a strictly positive increase in expected value if $x \geq 200$. Therefore, the expected number of loop iterations in the program is infinite. We are interested in deriving tail bounds on the number of loop iterations. Thus, we specify a cost model that incurs cost 1 in every loop iteration, again indicated by **tick** commands.

3.2 Non-negative Cost Supermartingales

We now show how non-negative cost supermartingales may be used to compute upper bounds on the expected cost usage in probabilistic programs with both positive and negative costs, as well as unbounded variable updates. To the best of our knowledge, we present the first such method.

Cost supermartingales were introduced in [Wang et al. 2019] for the expected value bound problem in programs that terminate almost-surely and have finite expected cost. A cost supermartingale ϕ is a function that maps program states to real values and satisfies the following two conditions:

```
        x := 1 , y := 1
ℓ_init: tick(y)
ℓ_1:    while x ≥ 1 do
ℓ_2:      r := Uniform([−1, 0.5])
ℓ_3:      x := x + r
ℓ_4:      if prob(0.5) then
ℓ_5:        tick(−y/2)
ℓ_6:        y := y − y/2
          else
ℓ_7:        tick(y/2)
ℓ_8:        y := y + y/2
ℓ_out:
```

(a)

```
        x = 0
ℓ_init: while x ≥ 0 do
ℓ_1:      tick(1)
ℓ_2:      r_1 := Uniform([−1, 0.5])
ℓ_3:      x := x + r_1
ℓ_4:      if x ≥ 200 then
ℓ_5:        r_2 := Uniform([−1, 2])
ℓ_6:        x := x + r_2
ℓ_out:
```

(b)

Fig. 3. Illustrative examples.

- *Expected decrease by incurred cost.* ϕ decreases in expected value at least by the incurred cost upon every one-step execution of the program at a reachable state.
- *Non-negativity upon termination.* ϕ is non-negative at every reachable terminal state.

We formally define cost supermartingales in Section 5. It can be verified by inspection that the following function is a cost supermartingale for the program in Figure 3a:

$$\phi(\ell, x, y, r) = \begin{cases} x + 2 \cdot y, & \text{if } \ell \in \{\ell_{init}, \ell_7\} \quad x, & \text{if } \ell = \ell_5 \\ x + y, & \text{if } \ell \in \{\ell_1, \ell_2, \ell_4, \ell_{out}\} \quad x + y/2, & \text{if } \ell = \ell_6 \\ x + y + r, & \text{if } \ell = \ell_3 \quad x + 3 \cdot y/2, & \text{if } \ell = \ell_8 \end{cases} \tag{1}$$

It was shown in [Wang et al. 2019] that, for an almost-surely terminating program, a cost supermartingale at every state evaluates to an upper bound on the expected cost usage from that state if either (1) all incurred costs in the program are non-negative and the cost supermartingale is non-negative at every reachable state, or (2) the program has bounded variable updates, i.e. the absolute value of changes in variable values upon updates is bounded by some $M > 0$. However, in Example 3.1 we showed that the program in Figure 3a satisfies neither of these two conditions.

Given an almost-surely terminating program, in Section 5 we prove that if the *cost supermartingale is non-negative* and the program satisfies the *lower-bounded total cost* condition, then the non-negative cost supermartingale is indeed an upper bound on the expected cost usage. A program satisfies the lower-bounded total cost condition if there exists $K \in \mathbb{R}$ such that the total incurred cost along every run in the program is bounded from *below* by K. This condition is satisfied in many cost analysis applications. For instance, in memory usage analysis we cannot de-allocate more memory than what has already been allocated so the memory usage is always bounded from below by $K = 0$. Similarly, any execution of a smart contract in Ethereum incurs both positive and negative gas costs, but the total cost of a path can never be less than a fixed minimum fee $K > 0$ [Dameron 2018]. Note that it is natural to allocate/deallocate large parts of memory whose size is not bounded by a constant, e.g. when copying an array. As another example, a robot cannot use a negative amount of energy so the energy usage of a robot is also bounded from below by $K = 0$. Regarding the non-negativity condition that we impose on cost supermartingales, we note that non-negativity of cost supermartingales is also required by [Wang et al. 2019] in the case of programs with non-negative costs.

Since in the program in Figure 3a we always have $y \geq 0$, this program satisfies the lower-bounded total cost condition with $K = 0$. Thus, the non-negative cost supermartingale ϕ in Equation (1) evaluates to an upper bound on expected cost usage of this program. In particular, for the initial state $(x, y) = (1, 1)$, the expected value of y upon termination is at most $\phi(\ell_{init}, 1, 1, r) = 3$.

3.3 Tail Bounds for Programs with Infinite Expected Total Cost

While the results in Section 3.2 extend the applicability of cost supermartingales for the expected value bound problem, their usage still requires either (1) non-negative costs, or (2) almost-sure termination and bounded variable updates, or (3) almost-sure termination and the lower-bounded total cost condition. In addition, previous methods for tail bound analysis require the program to have finite expected cost usage. For instance, this means that they cannot be used to derive tail bounds on termination time in programs that do not terminate almost-surely and hence have infinite expected termination time. We now outline our new method for the tail bound problem that neither imposes any of these assumptions nor requires finite expected cost usage.

To illustrate our method, consider the program in Figure 3b. As noted in Example 3.2, this program has infinite expected cost usage. However, by closer inspection, we observe that the challenging parts in the cost analysis of this program are those runs that reach a state with $x \geq 200$ at which time there is a positive probability of a run looping forever and never terminating. If, instead, the program were to terminate once a state with $x \geq 200$ is reached, then the existing approaches could be applied towards deriving cost bounds for this program. Therefore, if we could

(1) identify program runs that reach a state with $x \geq 200$ as challenging for cost analysis;
(2) derive an upper bound on the probability of reaching a challenging state with $x \geq 200$; and
(3) derive a tail bound on cost usage for the modified program that terminates once a challenging state with $x \geq 200$ is reached,

then it seems intuitive that we could combine these two probability bounds in (2) and (3) above in order to obtain a tail bound on cost usage for the original program in Figure 3b. This observation is at the heart of our novel method for the tail bound problem. It computes the following two objects:

(1) *Stochastic invariant.* A *stochastic invariant* [Chatterjee et al. 2017] is a tuple (SI, p_{SI}) where SI is a set of program states that a random program run leaves with probability at most $p_{SI} \in [0, 1]$. As an example, for the program in Figure 3b, a tuple defined via

$$SI(\ell) := \begin{cases} (x < 200), & \text{if } \ell \in \{\ell_{init}, \ell_1, \ell_2, \ell_4, \ell_{out}\} \\ (x + r_1 < 200), & \text{if } \ell = \ell_3 \\ \text{false}, & \text{otherwise} \end{cases} \tag{2}$$

and $p_{SI} = 0.005$ defines a stochastic invariant. The intuition behind the computation of a stochastic invariant is that it will allow us to restrict the tail bound analysis to a core part of the program that can be handled using our technique for expectation bounds.

(2) *Cost bound for the stochastic invariant.* Our method also computes an upper bound p_{cost} on the probability that the cost usage of a *part of a program run that does not leave SI* exceeds the threshold t. This bound is obtained by first computing a non-negative cost supermartingale ϕ for the stochastic invariant, which considers the modified program that terminates whenever a program run in the original program leaves the set of states SI. The construction of this modified program is formalized in Section 6.
Next, our method uses our notion of non-negative cost supermartingales (as illustrated in Section 3.2) to show that ϕ evaluates to an upper bound on the expected cost usage until a program run terminates or leaves the set SI. Note that, to do this, we need to ensure that the modified program is almost-surely terminating and satisfies the lower-bounded total cost

condition with some lower bound K, and we describe in Section 6 how our computation of the stochastic invariant ensures these two conditions. However, these conditions are enforced *only* on the part of the program defined by the stochastic invariant and *not* on the whole program. It is precisely this restriction that makes our method applicable to general programs with infinite expected total cost, including non-terminating programs.

Finally, our method applies Markov's inequality [Williams 1991] to the cost supermartingale ϕ to get an upper bound $p_{cost} = \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K}$ on the probability that the cost usage of the part of a program run that does not leave SI exceeds the threshold t. Here, $(\ell_{init}, \mathbf{x}_{init})$ denotes the initial state in the program. The correctness of this bound is also proven in Section 6.

Going back to our example program in Figure 3b, a non-negative cost supermartingale for the stochastc invariant in Equation (2) is given by

$$\phi(\ell, x, r_1, r_2) = \begin{cases} 4 \cdot x + 1, & \text{if } \ell = \{\ell_{init}, \ell_1, \ell_4, \ell_5, \ell_6, \ell_{out}\} \\ 4 \cdot x, & \text{if } \ell = \ell_2 \\ 4 \cdot x + 1 + 4 \cdot r_1, & \text{if } \ell = \ell_3 \end{cases} . \tag{3}$$

Since the part of the program defined by SI is easily observed to be almost-surely terminating and satisfying the lower-bounded total cost condition with $K = 0$, our method concludes that the probability that the cost usage of a part of a program run that does not leave SI exceeds the threshold t from the initial state with $x = 0$ is bounded from above by $\frac{\phi(0, r_1, r_2) - 0}{t - 0} = \frac{1}{t}$.

Finally, our approach combines p_{SI} and p_{cost} towards obtaining an upper bound on the probability that cost usage of a program run (in the whole original program) exceeds t and concludes that

$$\mathbb{P}\left[Cost \geq t\right] \leq p_{SI} + p_{cost}.$$

Hence, in order to prove a tail bound $\mathbb{P}[Cost \geq t] \leq p$ on cost usage for a given threshold value t and probability parameter $p \in [0, 1]$, it suffices to constrain the computation of the stocastic invariant and the cost bound in a way that ensures $p_{SI} + p_{cost} \leq p$. The intuition behind this result is that, for cost usage to exceed t, a program run must either stay within SI and incur a greater cost or eventually leave SI. However, our approach proves that the probabilities of these two events are bounded from above by p_{cost} and p_{SI}, respectively. Revisiting our example in Figure 3b, our method can thus show that $\mathbb{P}[Cost \geq t] \leq 0.005 + \frac{1}{t}$. Note that this is a symbolic bound based on the threshold t. For example, if t is given as 200, then we obtain $\mathbb{P}[Cost \geq 200] \leq 0.01$.

3.4 Fully Automated Template-based Synthesis

For linear/polynomial programs, we show that our method can be automated by a template-based synthesis approach similar to [Asadi et al. 2021]. In particular, our method synthesizes the stochastic invariant and the non-negative cost supermartingale *simultaneously* while constraining the synthesis to satisfy the tail bound on cost usage that is required to be proven.

4 PRELIMINARIES

4.1 Program Syntax and Semantics

We consider imperative arithmetic probabilistic programs with non-determinism. Our detailed syntax is given in [Chatterjee et al. 2024, Appendix A]. Our programs allow standard programming constructs including conditional branching, loops and variable assignments. In addition, they allow sampling instructions and non-deterministic assignments. *Sampling instructions* are indicated in syntax by a command of the form "$x \coloneqq \textbf{sample}(d)$" which samples a random value from a probability distribution d and assigns it to x. Our programs allow both discrete, e.g. Bernoulli and Poisson,

and continuous, e.g. uniform and normal, probability distributions in sampling instructions. *Non-deterministic assignments* are assignments of the form "$x := \mathbf{ndet}(B)$" that non-deterministically pick an element of the set B and assign it to x. Our syntax also supports probabilistic branching of the form "**if** prob(p) **then ... else ...**" and non-deterministic branching of the form "**if** $\star$ **then ... else ...**". Finally, we have a dedicated command $\mathbf{tick}(e)$, that is used to model cost usage and indicates that the program incurs a cost equal to the value of the arithmetic expression e over program variables.

We assume that variables in our programs are real-valued. Given a finite set of variables V, a *variable valuation* of V is a vector $\mathbf{x} \in \mathbb{R}^{|V|}$ that assigns a value to every variable. A *predicate* over a set of variables is a boolean combination of finitely many inequalities over them. For a predicate ϕ over a finite set of variables V and a valuation $\mathbf{x}$ of V, we write $\mathbf{x} \models \phi$ to denote that the formula obtained by substituting into ϕ the values of variables defined by $\mathbf{x}$ evaluates to true.

We model programs via probabilistic control-flow graphs (pCFGs) [Agrawal et al. 2018; Chatterjee et al. 2018, 2023]. A probabilistic program written in our syntax can be straightforwardly translated to an equivalent pCFG [Chatterjee et al. 2018]. Specifically to cost analysis, the map Tk assigns costs that are incurred by executing a transition in the pCFG.

Probabilistic control-flow graphs (pCFGs) A *probabilistic control-flow graph (pCFG)* is a tuple $C = (L, V, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up, Tk)$, where:

- L is a finite set of *locations*;
- $V = \{x_1, x_2, \ldots, x_{|V|}\}$ is a finite set of *program variables*;
- ℓ_{init} is the *initial program location* and $\mathbf{x}_{init} \in \mathbb{R}^{|V|}$ is the *initial variable valuation*;
- $\mapsto$ is a finite set of *transitions*. Each transition is a tuple of the form $\tau = (\ell, \delta)$, where ℓ is the *source location* and δ is the probability distribution over *target locations* of τ;
- G is a map assigning to each transition $\tau \in \mapsto$ a *guard* $G(\tau)$, which is a logical formula over V specifying whether the transition τ can be executed;
- Up is a map assigning to each transition $\tau \in \mapsto$ an *update* $Up(\tau) = (j, u)$ where $j \in \{1, \ldots, |V|\}$ is a *target variable index* and u is an *update element* which can be:
  - the bottom element $u = \bot$, denoting no update;
  - a Borel-measurable map $u : \mathbb{R}^{|V|} \to \mathbb{R}$, denoting deterministic variable assignment;
  - a probability distribution $u = d$, denoting that the new variable value is sampled according to the probability distribution d;
  - an interval $u = [a, b] \subseteq \mathbb{R} \cup \{\pm\infty\}$, denoting a non-deterministic update. We also allow one or both sides of the interval to be open.
- Tk is a map assigning to each transition $\tau \in \mapsto$ a Borel-measurable *tick expression* $Tk(\tau)$ over program variables, that for each variable valuation $\mathbf{x} \in \mathbb{R}^{|V|}$ evaluates to the cost of executing τ when program variable values are defined by the valuation $\mathbf{x}$.

We assume the existence of a *terminal location* ℓ_{out} that a program enters upon termination. This location only has a self-loop with a trivial guard and update as an outgoing transition. Also, we assume that it is always possible to execute at least one transition, i.e. that for each location ℓ the disjunction $\bigvee_{\tau=(l,\_)} G(\tau)$ of guards of all outgoing transitions is equivalent to *true*. Note that this assumption is imposed without loss of generality as we may introduce additional transitions to ℓ_{out}.

States, paths and runs A *state* in a pCFG C is a tuple $(\ell, \mathbf{x})$, where ℓ is a location and $\mathbf{x} \in \mathbb{R}^{|V|}$ is a variable valuation in C. A transition $\tau = (\ell, \delta)$ is *enabled* at a state $(\ell, \mathbf{x})$ if $\mathbf{x} \models G(\tau)$. A state $(\ell', \mathbf{x}')$ is a *successor* of $(\ell, \mathbf{x})$, if there exists an enabled transition $\tau = (\ell, \delta)$ in C such that $\delta(\ell') > 0$ and such that $(\ell', \mathbf{x}')$ can be reached from $(\ell, \mathbf{x})$ by executing τ and applying the updates of τ to $\mathbf{x}$.

A *finite path* in C is a sequence $(\ell_0, \mathbf{x}_0), (\ell_1, \mathbf{x}_1), \ldots, (\ell_k, \mathbf{x}_k)$ of states with $(\ell_0, \mathbf{x}_0) = (\ell_{init}, \mathbf{x}_{init})$ and with $(\ell_{i+1}, \mathbf{x}_{i+1})$ being a successor of $(\ell_i, \mathbf{x}_i)$ for each $0 \leq i \leq k-1$. A state $(\ell, \mathbf{x})$ is *reachable* in C if there exists a finite path in C with the last state $(\ell, \mathbf{x})$. A *run* (or *execution*) in C is an infinite

sequence of states in which each finite prefix is a finite path. We use $State_C$, $Fpath_C$, Run_C, $Reach_C$ to denote the set of all states, finite paths, runs and reachable states in C, respectively.

Schedulers As we will show below, the semantics of pCFGs are formalized by defining a probability space over the set of all runs in the pCFG. However, due to the possible existence of non-determinism, we cannot define such a probability space directly but need to first resolve the non-determinism. This is achieved by introducing the notion of a scheduler. A *scheduler* in a pCFG C is a map σ which to each finite path $\rho \in Fpath_C$ assigns a probability distribution $\sigma(\rho)$ over successor states of the last state in ρ. We introduce an additional *measurability* assumption on schedulers in order for the semantics of probabilistc programs with non-determinism to be mathematically well-defined. The restriction to measurable schedulers is standard in the analysis of non-deterministic stochastic systems [Neuhäußer and Katoen 2007; Neuhäußer et al. 2009]. Thus, we omit the formal definition.

Semantics of pCFGs A pCFG C together with a scheduler σ define a stochastic process taking values in the set of states of C, whose trajectories correspond to runs in C. This gives rise to a probability space $(Run_C, \mathcal{F}_C, \mathbb{P}^\sigma)$ over the set of all runs in C and a stochastic process $C^\sigma = \{\mathbf{C}_i^\sigma\}_{i=0}^\infty$ in this space where $\mathbf{C}_i^\sigma(\rho)$ is the i-th configuration along ρ for each run $\rho \in Run_C$. The construction of this stochastic process is standard in probabilistic program analysis (see e.g. [Agrawal et al. 2018]), however we provide both the intuitive description and formal details in [Chatterjee et al. 2024, Appendix B]. We denote by $\mathbb{E}^\sigma$ the expectation operator over $(Run_C, \mathcal{F}_C, \mathbb{P}^\sigma)$. We may analogously define a probability space $(Run_{C(\ell,\mathbf{x})}, \mathcal{F}_{C(\ell,\mathbf{x})}, \mathbb{P}^\sigma_{C(\ell,\mathbf{x})})$ over the set of all runs in C that start in some specified state $(\ell, \mathbf{x})$.

State and predicate functions A *state function* f is a map that, to each location $\ell \in L$ in C, assigns a Borel-measurable expression $f(\ell) : \mathbb{R}^{|V|} \to \mathbb{R}$ over program variables. We use $f(\ell, \mathbf{x})$ to denote the value of the expression $f(\ell)$ on valuation $\mathbf{x} \in \mathbb{R}^{|V|}$. A *predicate function* in C is a map ϕ that to every location $\ell \in L$ assigns a predicate $\phi(\ell)$. It naturally induces a set of states at which the assigned predicate is satisfied, i.e. $\bigcup_{\ell \in L}\{(\ell, \mathbf{x}) \mid \mathbf{x} \models \phi(\ell)\}$. For a predicate function ϕ, we use $\neg\phi$ to denote the negated predicate function, i.e. $(\neg\phi)(\ell) = \neg(\phi(\ell))$ for each $\ell \in L$.

Almost-sure termination We say that a state $(\ell, \mathbf{x})$ in C is a *terminal state* if $\ell = \ell_{out}$ and a run $\rho \in Run_C$ is said to be *terminating* if it reaches some terminal state. We use $Term \subseteq Run_C$ to denote the set of all terminating runs in C. A pCFG C is then said to be *almost-surely terminating* if the probability of a random run terminating is 1 with respect to every scheduler, i.e. if $\inf_\sigma \mathbb{P}^\sigma[Term] = 1$.

4.2 Formal Definitions of Expectation and Tail Bounds on Resource Usage

We now formalize the quantitative cost analysis problems in probabilistic programs that we consider in this work. We first need to formally define the notion of a cost of a probabilistic program run.

Cost of a run Given a run $\rho = (\ell_i, \mathbf{x}_i)_{i=0}^\infty \in Run_C$, for every $i \in \mathbb{N}_0$ let $Tk_i(\rho)$ denote the tick expression assigned to the i-th transition along ρ. Then, the *cost* $Cost_C(\rho)$ of ρ is equal to the limit superior value of the sums of costs incurred by the run ρ, i.e. $Cost_C(\rho) = \limsup_{m \geq 0} \sum_{i=0}^m Tk_i(\rho)(\mathbf{x}_i)$.[§]

Quantitative cost analysis problems Let C be a pCFG. We consider the following two problems:

- Given a threshold value $t \in \mathbb{R}$, the *expectation bound problem* aims to prove that for every scheduler the expected value of the total cost does not exceed t, i.e. $\sup_\sigma \mathbb{E}^\sigma[Cost_C] \leq t$.
- Given a threshold value $t \in \mathbb{R}$ and a probability parameter $p \in [0, 1]$, the *tail bound problem* aims to prove that for every scheduler the probability of cost usage exceeding the threshold t is at most p, i.e. $\sup_\sigma \mathbb{P}^\sigma[Cost_C \geq t] \leq p$.

[§]Note that we cannot simply replace lim sup in the definition with lim, since if a run ρ does not terminate then the sequence might not have a limit. Since lim sup of a sequence of real values coincides with its limit whenever the limit exists, this definition agrees with and generalizes the standard definition of cost in almost-surely terminating programs.

In both cases we consider the decision variants of the problems. For optimization, one may use the method for the corresponding decision problem as a subroutine and perform a binary search.

5 COST SUPERMARTINGALES FOR GENERAL COSTS AND UPDATES

We now present our extension of cost supermartingales to probabilistic programs with both positive and negative costs and with unbounded variable updates. Cost supermartingales [Wang et al. 2019] are a class of state functions that at each state evaluate to an upper bound on the expected cost usage from that state. However, they are only applicable to almost-surely terminating programs in which either all incurred costs are non-negative or all variable assignments have bounded updates. We now prove that, if a cost supermartingale is in addition required to be *non-negative* at every reachable state, then it evaluates to an upper bound on the expected cost usage in probabilistic programs with general costs and unbounded variable updates if the program is almost-surely terminating and satisfies the lower-bounded total cost condition. Note that the lower-bounded total cost condition applies to every run of the program and is not part of the martingale constraints.

Assumptions. Our result in this section requires programs to be *almost-surely terminating* and to satisfy the *lower-bounded total cost* condition. For a pCFG C that models a program, we say that it satisfies the lower-bounded total cost condition with lower bound $K \in \mathbb{R}$ if the cost usage along every path is bounded from below by K, i.e. $\sum_{i=0}^{m} Tk_i(\rho)(\mathbf{x}_i) \geq K$ for each $\rho \in Run_C$ and $m \in \mathbb{N}_0$.

Invariants. As mentioned in Section 3, the defining conditions of cost supermartingales impose conditions on their values at reachable states. Since it is infeasible to compute the exact set of reachable states in a program, we define cost supermartingales with respect to a supporting invariant. An *invariant* I is a predicate function in C which contains all reachable states in C.

Cost supermartingales. Let $C = (L, V, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up, Tk)$ be a pCFG and I an invariant in C. Definition 5.1 below formally defines cost supermartingales. Intuitively, the condition (C_1) requires the cost supermartingale to be non-negative at every reachable *terminal* state. The condition (C_2) requires the value of the cost supermartingale at every reachable state to decrease in expected value upon executing a transition in the pCFG at least by the cost of executing the transition. Recall that the cost of executing a transition is specified by the map Tk in the pCFG. We use the supporting invariant I to capture the set of reachable states.

Definition 5.1 (Cost supermartingale [Wang et al. 2019]). A state function ϕ in C is said to be a *cost supermartingale* with respect to an invariant I, if it satisfies the following conditions:

(C_1) *Non-negativity upon termination.* We have $\mathbf{x} \models I(\ell_{out}) \Rightarrow \phi(\ell_{out}, \mathbf{x}) \geq 0$.

(C_2) *Expected decrease by incurred cost.* For every location $\ell \in L$ and transition $\tau = (\ell, \delta) \in \mapsto$ outgoing from ℓ with $Up(\tau) = (j, u)$, depending on the type of the update u we have:

- If $Up(\tau) = (j, \bot)$, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow \phi(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \phi(\ell', \mathbf{x}) + Tk(\tau)(\mathbf{x}).$$
- If $Up(\tau) = (j, u)$ with $u : \mathbb{R}^{|V|} \to \mathbb{R}$ a Borel-measurable map, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow \phi(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \phi(\ell', \mathbf{x}[x_j \leftarrow u(\mathbf{x})]) + Tk(\tau)(\mathbf{x}).$$
- If $Up(\tau) = (j, u)$ with $u = d$ a probability distribution, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow \phi(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \mathbb{E}_{X \sim d}[\phi(\ell', \mathbf{x}[x_j \leftarrow X])] + Tk(\tau)(\mathbf{x}).$$
- If $Up(\tau) = (j, u)$ with $u = [a, b]$ a real-valued interval, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow \phi(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \sup_{X \in [a,b]} \{\phi(\ell', \mathbf{x}[x_j \leftarrow X])\} + Tk(\tau)(\mathbf{x}).$$

Non-negative Cost Supermartingales. We say that a cost supermartingale ϕ is *non-negative* if $\phi(\ell, \mathbf{x}) \geq 0$ for each $\ell \in L$ and $\mathbf{x} \models I(\ell)$. In other words, non-negative cost supermartingales have

non-negative values over all reachable states, not just terminal states. The following theorem is our main result in this section.

Theorem 5.2 (Proof in [Chatterjee et al. 2024]). *Let C be a pCFG, I an invariant in C and ϕ a non-negative cost supermartingale with respect to I. Suppose that C terminates almost-surely and that it satisfies the lower-bounded total cost condition with lower bound $K \in \mathbb{R}$. Then, the expected cost usage with respect to every scheduler in C is bounded from above by the initial value of the cost supermartingale ϕ, i.e.*

$$\sup_\sigma \mathbb{E}^\sigma [Cost_C] \leq \phi(\ell_{init}, \mathbf{x}_{init}).$$

In the proof of Theorem 5.2 in [Chatterjee et al. 2024, Appendix D], we show that a non-negative cost supermartingale ϕ induces an instance of the mathematical notion of a *non-negative supermartingale* [Williams 1991] in the probability space over the set of all pCFG runs. We then prove the theorem claim by using Optional Stopping Theorem for non-negative supermartingales (OST). The key technical novelty compared to the proof of [Wang et al. 2019] for programs with bounded variable updates is that we observe that using this variant of OST allows us to drop the positive costs and the bounded variable updates assumptions while only requiring the lower-bounded total cost assumption. We refer the reader to [Chatterjee et al. 2024, Appendices C and D] for the detailed proof.

Remark 5.1. *We conclude this section by observing that, if all costs in a probabilistic program are non-negative, then we may remove the almost-sure termination assumption. Note that the lower bounded total cost condition in this case is satisfied by default with $K = 0$. This is in fact the result of [Wang et al. 2019] on non-negative cost supermartingales for probabilistic programs with non-negative costs. The work [Wang et al. 2019] considers almost-surely terminating programs, however the proof of [Wang et al. 2019, Theorem 6.14] does not require almost-sure termination assumption.*

6 TAIL BOUNDS ON THE RESOURCE USAGE OF PROBABILISTIC PROGRAMS

We now show how non-negative cost supermartingales and stochastic invariants can be combined in order to obtain, to the best of our knowledge, the first approach to tail bound cost analysis in probabilistic programs that can handle both positive and negative costs, unbounded variable updates and even infinite expected cost usage. This section presents the theory behind our new method for tail bound analysis. We will then present a fully automated algorithm in Section 7.

We start by showing how non-negative cost supermartingales can be used to derive tail bounds in probabilistic programs with finite expected cost usage in Section 6.1. Next, in Section 6.2 we consider stochastic invariants and stochastic invariant indicators. Finally, in Section 6.3 we show how non-negative cost supermartingales and stochastic invariants can together be used to prove tail bounds in general probabilistic programs that need not have finite expected cost usage. Below, let $C = (L, V, \ell_{init}, \mathbf{x}_{init}, \mapsto, G, Up, Tk)$ be a pCFG and I be an invariant in C.

6.1 Tail Bounds via Non-negative Cost Supermartingales

We showed in Theorem 5.2 that non-negative cost supermartingales evaluate to an upper bound on the expected cost usage in probabilistic programs that are almost-surely terminating and satisfy the lower-bounded total cost condition. Thus, one can derive tail bounds on cost usage by first computing a non-negative cost supermartingale in order to bound the expected cost usage, and then applying any theorem or concentration bound from probability theory which uses the expected value of a random variable to bound its tail probabilities. In particular, we use the classical Markov's inequality to derive tail bounds.

Proposition 6.1 (Markov's inequality [Williams 1991]). *Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space, and let X be a non-negative random variable in $(\Omega, \mathcal{F}, \mathbb{P})$. Then, for any $t > 0$, $\mathbb{P}[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$.*

The challenge in applying Markov's inequality is that the random variable defined by the total cost usage need not be non-negative. Nevertheless, as non-negative cost supermartingales already require the lower-bounded total cost condition with some lower bound $K \in \mathbb{R}$, we may increase the total cost usage and therefore its expected value by $-K$ in order to obtain a non-negative random variable to which Markov's inequality becomes applicable. This leads to the following theorem:

Theorem 6.2 (Proof in [Chatterjee et al. 2024]). *Let C be a pCFG, I an invariant in C, and ϕ a non-negative cost supermartingale with respect to I. Suppose that C terminates almost-surely and that it satisfies the lower-bounded total cost condition with lower bound $K \in \mathbb{R}$. Then, for every $t > K$, we have*

$$\sup_\sigma \mathbb{P}^\sigma\left[Cost_C \geq t\right] \leq \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K}.$$

On the other hand, for every $t \leq K$, we have $\sup_\sigma \mathbb{P}^\sigma[Cost_C \geq t] = 1$.

For programs with non-negative costs, we may again relax the almost-sure termination assumption by using Remark 5.1 instead of Theorem 5.2 (note that such programs have lower-bounded total cost with $K = 0$). The proof of the following theorem is then analogous to that of Theorem 6.2.

Theorem 6.3. *Let C be a pCFG, I an invariant in C and ϕ a non-negative cost supermartingale with respect to I. Suppose that all incurred costs in C are non-negative. Then, for every $t > 0$, we have*

$$\sup_\sigma \mathbb{P}^\sigma\left[Cost_C \geq t\right] \leq \frac{\phi(\ell_{init}, \mathbf{x}_{init})}{t}.$$

On the other hand, for every $t \leq 0$, we have $\sup_\sigma \mathbb{P}^\sigma[Cost_C \geq t] = 1$.

6.2 Stochastic Invariants

We now provide an overview of stochastic invariants, which are a proof concept for quantitative safety. The next section will show how to combine stochastic invariants with non-negative cost supermartingales to enable tail bound analysis. Stochastic invariants generalize invariants which are a classical notion in non-probabilistic programs and specify a set of states in the program's pCFG together with an upper bound on the probability of a random program run leaving this set of states.

Definition 6.4 (Stochastic invariant [Chatterjee et al. 2017]). Let SI be a predicate function in C and $p_{SI} \in [0, 1]$ be a probability parameter. The tuple (SI, p_{SI}) is a *stochastic invariant* if the probability of a run in C ever leaving the set of states defined by SI is at most p_{SI} under any scheduler σ, i.e. if

$$\sup_\sigma \mathbb{P}^\sigma\left[\left\{\rho \in Run_C \mid \rho \text{ reaches } (\ell, \mathbf{x}) \text{ with } \mathbf{x} \not\models SI(\ell)\right\}\right] \leq p_{SI}.$$

Computing stochastic invariants directly from their definition is a challenging task, and previous works have sought alternative characterizations that allow automated computation of stochastic invariants. In particular, in our method we use the recent characterization of [Chatterjee et al. 2022] via *stochastic invariant indicators (SI-indicators)*. Intuitively, an SI-indicator is an ordered tuple (f_{SI}, p_{SI}), consisting of a state function f_{SI} that to each state in the pCFG assigns an upper bound on the probability that a random program run from that state reaches a state at which $f_{SI} \geq 1$, and a probability parameter p_{SI} which is an upper bound on the value of f_{SI} at the initial state in the pCFG. Intuitively, SI is defined implicitly as the set of states in which $f_{SI} < 1$. The defining conditions of SI-indicators indeed ensure that these properties are satisfied.

Definition 6.5 (Stochastic invariant indicator [Chatterjee et al. 2022]). Let f_{SI} be a state function and $p_{SI} \in [0, 1]$ be a probability parameter. The tuple (f_{SI}, p_{SI}) is a *stochastic invariant indicator (SI-indicator)* with respect to an invariant I, if it satisfies the following conditions:

(C_1) *Non-negativity.* For every location $\ell \in L$, we have $\mathbf{x} \models I(\ell) \Rightarrow f_{SI}(\ell, \mathbf{x}) \geq 0$.

(C_2) *Non-increasing expected value.* For every location $\ell \in L$ and transition $\tau = (\ell, \delta) \in \mapsto$ outgoing from ℓ with $Up(\tau) = (j, u)$, depending on the type of the update u we have:

- If $Up(\tau) = (j, \bot)$, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow f_{SI}(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot f_{SI}(\ell', \mathbf{x}).$$
- If $Up(\tau) = (j, u)$ with $u : \mathbb{R}^{|V|} \to \mathbb{R}$ a Borel-measurable expression, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow f_{SI}(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot f_{SI}(\ell', \mathbf{x}[x_j \leftarrow u(\mathbf{x}_i)]).$$
- If $Up(\tau) = (j, u)$ with $u = d$ a probability distribution, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow f_{SI}(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \mathbb{E}_{X \sim d}[f_{SI}(\ell', \mathbf{x}[x_j \leftarrow X])].$$
- If $Up(\tau) = (j, u)$ with $u = [a, b]$ a real-valued interval, then
$$\mathbf{x} \models I(\ell) \wedge G(\tau) \Rightarrow f_{SI}(\ell, \mathbf{x}) \geq \sum_{\ell' \in L} \delta(\ell') \cdot \sup_{X \in [a,b]} \{f_{SI}(\ell', \mathbf{x}[x_j \leftarrow X])\}.$$

(C_3) *Initial condition.* We have $f_{SI}(\ell_{init}, \mathbf{x}_{init}) \leq p_{SI}$.

The first two defining conditions of an SI-indicator (f_{SI}, p_{SI}) require f_{SI} to be a state function that at each state contained in the invariant is non-negative and non-increasing in expected value upon a one-step execution of the pCFG at that state, with respect to every scheduler used to resolve non-determinism. It can be shown that these two conditions together imply that f_{SI} at each reachable state evaluates to an upper bound on the probability of reaching a state with $f_{SI} \geq 1$.

The third condition additionally requires $f(\ell_{init}, \mathbf{x}_{init}) \leq p_{SI}$. Hence, the conditions imply that the set of states that are contained in the invariant I and in which $f_{SI} < 1$ together with p_{SI} define a stochastic invariant. In other words, we have $SI = \{(\ell, \mathbf{x}) \mid \mathbf{x} \models I(\ell) \wedge f_{SI}(\ell, \mathbf{x}) < 1\}$. Thus, every SI-indicator naturally induces a stochastic invariant. This claim is formalized in Theorem 6.6, which shows that characterization of stochastic invariants via SI-indicators is not only *sound* but also *complete*. In particular, it shows that for every stochastic invariant there exists an SI-indicator with this property. Thus, in order to compute stochastic invariants, we may equivalently search for SI-indicators with the same probability threshold.

Theorem 6.6 ([Chatterjee et al. 2022]). *Let C be a pCFG and I be an invariant in C. If (f_{SI}, p_{SI}) is an SI-indicator with respect to I, then the predicate map SI defined as $SI(\ell) = (\mathbf{x} \models I(\ell) \wedge f_{SI}(\ell, \mathbf{x}) < 1)$ for every $\ell \in L$ yields a stochastic invariant (SI, p_{SI}) in C.*

Conversely, if (SI, p_{SI}) is a stochastic invariant, then there exist an invariant I_{SI} and a state function f_{SI} in C such that (f_{SI}, p_{SI}) is an SI-indicator with respect to I_{SI} and such that for each $\ell \in L$ we have $SI(\ell) \supseteq (\mathbf{x} \models I_{SI}(\ell) \wedge f_{SI}(\ell, \mathbf{x}) < 1)$.

6.3 Combining Stochastic Invariants and Non-negative Cost Supermartingales for Tail Bound Analysis

We now have all the ingredients of our approach for tail bounds in probabilistic programs with arbitrary costs and updates and potentially infinite expected total cost. Recall that the goal of the tail bounds problem is to prove the tail bound on cost usage $\sup_\sigma \mathbb{P}^\sigma [Cost_C \geq t] \leq p$, where $t \in \mathbb{R}$ and $p \in [0, 1]$ are a given threshold value and a probability parameter, respectively.

At a high level, our method proves the tail bound by computing a stochastic invariant in the pCFG together with a cost supermartingale for the part of the pCFG defined by the stochastic invariant. As sketched in Section 3, the stochastic invariant restricts cost analysis to a subset of runs over which the expected cost usage is finite and thus using cost analysis methods that bound the expected cost usage becomes feasible. Based on this, our method uses the computed cost supermartingale to derive a tail bound on cost usage for the part of the pCFG defined by the stochastic invariant and adds this tail bound to the probability of a random program run ever leaving the stochastic

invariant in order to prove that the probability of cost usage in the whole program exceeding the threshold t is at most p. We now formalize these ideas.

Non-negative cost supermartingale for a stochastic invariant Let (SI, p_{SI}) be a stochastic invariant in C. We define a new pCFG C_{SI} from C, which is intuitively obtained from C by letting a run in C terminate whenever it leaves SI. Formally, C_{SI} is constructed by conjuncting the guard of each transition $\tau = (\ell, \ell')$ in C with the predicate $SI(\ell)$ and introducing a new zero cost transition from ℓ to ℓ_{out} with guard $\neg SI(\ell)$. As such, the set of runs that never leave SI in C is trivially in a one-to-one correspondence with the set of runs that never take any of the newly added transitions in C_{SI}. A *non-negative cost supermartingale for the stochastic invariant* (SI, p_{SI}) with respect to the invariant I is defined as a non-negative cost supermartingale with respect to the invariant I in C_{SI}.

The following theorem is a technical result which shows how non-negative cost supermartingales for the stochastic invariant (SI, p_{SI}) can be used to derive probability bounds on cost usage in the original pCFG C. The proof follows from our tail bounds in Section 6.1 and the one-to-one correspondence between the runs in C and C_{SI} described above. In the sequel, we use $Reach(\neg SI) \subseteq Run_C$ to denote the set of all runs in C that eventually leave the stochastic invariant SI.

Theorem 6.7 (Proof in [Chatterjee et al. 2024]). *Let C be a pCFG, I an invariant in C, (SI, p_{SI}) a stochastic invariant in C and ϕ a non-negative cost supermartingale for (SI, p_{SI}) with respect to I.*

(1) If all incurred costs in C are non-negative and $t > 0$, then

$$\sup_\sigma \mathbb{P}^\sigma[\{Cost_C \geq t\} \setminus Reach(\neg SI)] \leq \frac{\phi(\ell_{init}, \mathbf{x}_{init})}{t}.$$

(2) If the pCFG C_{SI} is almost-surely terminating and satisfies the lower-bounded total cost condition with lower bound $K \in \mathbb{R}$ and $t > K$, then

$$\sup_\sigma \mathbb{P}^\sigma[\{Cost_C \geq t\} \setminus Reach(\neg SI)] \leq \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K}.$$

Tail bounds for programs with non-negative costs We now describe our method for the case when all incurred costs in the pCFG C are non-negative. We start with this simpler case since it will not require us to worry about the almost-sure termination and the lower-bounded total cost conditions that need to be checked for the pCFG C_{SI} in order to use non-negative cost supermartingales. We will then extend the method to pCFGs with general costs.

Suppose that C is induced by a program in which all incurred costs are non-negative. Note that, if the threshold t is not positive, then $\sup_\sigma \mathbb{P}^\sigma[Cost_C \geq t] \leq p$ is true if and only if $p = 1$ and the tail bound problem becomes trivial. Thus, we assume without loss of generality that $t > 0$. In order to prove the desired tail bound, our method synthesizes the following objects:

(1) an SI-indicator (f_{SI}, p_{SI}) with respect to I that defines a stochastic invariant (SI, p_{SI}), and

(2) a non-negative cost supermartingale ϕ for (SI, p_{SI}) such that $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init})}{t} \leq p$.

The following theorem establishes correctness of our method, i.e. that if such an SI-indicator and non-negative cost supermartingale exist then the desired tail bound on cost usage is guaranteed.

Theorem 6.8 (Proof in [Chatterjee et al. 2024]). *Let C be a pCFG, I an invariant in C, $t > 0$ and $p \in [0, 1]$. Suppose that there exist a stochastic invariant (SI, p_{SI}) and a non-negative cost supermartingale ϕ for the stochastic invariant (SI, p_{SI}) with respect to the invariant I, such that $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init})}{t} \leq p$. Then,*

$$\sup_\sigma \mathbb{P}^\sigma[Cost_C \geq t] \leq p.$$

Tail bounds for programs with general costs We now consider the case of pCFGs with general costs. The key challenge in extending our method for pCFGs with non-negative costs to the general setting is that we need to ensure that the pCFG C_{SI} obtained by considering the part of the pCFG defined by the stochastic invariant is almost-surely terminating and that it satisfies the lower-bounded total cost condition. This is necessary in order to be able to use the non-negative cost

supermartingale for the stochastic invariant towards deriving tail bounds on cost usage. Note that these conditions need to be imposed *only for the part of the pCFG defined by the stochastic invariant* to which we restrict cost analysis. In other words, they apply to C_{SI} and not to C itself. The fact that our method does not impose these conditions on the original pCFG is precisely what makes it applicable to general probabilistic programs. We show how these conditions can be guaranteed:

- *Almost-sure termination.* In addition to computing an SI-indicator and a cost supermartingale, our method also computes a *ranking supermartingale (RSM)* for the target set of states $\{(\ell_{out}, \mathbf{x}) \mid \mathbf{x} \models I(\ell_{out})\} \cup \neg SI$. RSMs are a classical certificate for probability 1 reachability and termination in probabilistic programs [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2018; Takisaka et al. 2021]. An RSM is a state function that is required to be non-negative at every reachable state and to decrease in expected value by some $\varepsilon > 0$ upon every one-step execution of the pCFG. Thus, RSMs are equivalent to non-negative cost supermartingales if each transition were to incur cost ε for some $\varepsilon > 0$.
- *Lower-bounded total cost.* To impose that C_{SI} satisfies the lower-bounded total cost condition, our method first modifies C by adding a new variable *cost* that is used to track the total cost usage and is incremented by the cost of executed transition at each time step. Then, it constraints the SI-indicator (f_{SI}, p_{SI}) to satisfy for some $K \in \mathbb{R}$ and every $\ell \in L$
$$\mathbf{x} \models I(\ell) \wedge f_{SI}(\ell, \mathbf{x}) < 1 \Rightarrow \mathbf{x}[cost] \geq K,$$
where $\mathbf{x}[cost]$ is used to denote the value of the new variable *cost* defined by the variable valuation $\mathbf{x}$. This extra condition ensures that, whenever at some reachable state we have $f_{SI}(\ell, \mathbf{x}) < 1$ so that $\mathbf{x} \in SI(\ell)$, we must also have that the total cost of any run in C with the last state in $(\ell, \mathbf{x})$ is bounded from below by K. Since C_{SI} is obtained from C by terminating all runs that leave SI, this additional condition on the SI-indicator indeed implies that C_{SI} satisfies the lower bounded total cost condition with lower bound K.

Hence, our method for pCFGs with general costs can be summarized as simultaneously synthesizing:
(1) a real value $K \in \mathbb{R}$,
(2) an SI-indicator (f_{SI}, p_{SI}) with respect to I that defines a stochastic invariant (SI, p_{SI}), which is required to satisfy $\mathbf{x} \models I(\ell) \wedge f_{SI}(\ell, \mathbf{x}) < 1 \Rightarrow \mathbf{x}[cost] \geq K$ for every $\ell \in L$,
(3) an RSM g ensuring almost-sure termination in C_{SI},
(4) a non-negative cost supermartingale ϕ for (SI, p_{SI}), such that $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K} \leq p$.

Note that $\frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K}$ is exactly the upper bound on the tail probability in Theorem 6.7, Part 2. The following theorem establishes the correctness of our method.

Theorem 6.9 (Proof in [Chatterjee et al. 2024]). *Let C be a pCFG, I an invariant in C, $t > 0$ and $p \in [0, 1]$. Suppose that there exist a stochastic invariant (SI, p_{SI}) and a non-negative cost supermartingale ϕ for the stochastic invariant (SI, p_{SI}) with respect to the invariant I, such that C_{SI} is almost-surely terminating, satisfies the lower-bounded total cost condition with lower bound $K \in \mathbb{R}$ and such that $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K} \leq p$. Then, we have $\sup_\sigma \mathbb{P}^\sigma [Cost_C \geq t] \leq p$.*

7 TEMPLATE-BASED SYNTHESIS ALGORITHM

We now provide an automated algorithm for template-based synthesis of expectation and tail bounds. Our algorithm relies on classical theorems in polyhedral and real algebraic geometry including Farkas' Lemma [Avis and Kaluzny 2004; Farkas 1902], Handelman's Theorem [Handelman 1988] and Putinar's Positivstellensatz [Putinar 1993]. It is applicable to linear/polynomial programs, i.e. programs in which all arithmetic expressions are linear/polynomial expressions over the program variables. While our theoretical results in the previous sections are applicable to general probabilistic programs, we restrict ourselves to linear/polynomial programs in this section to obtain automation.

In this work, we considered three distinct problems and obtained proof concepts for all of them:

(1) ***Expectation bounds:*** As shown in Section 5, if the pCFG C is almost-surely terminating and satisfies the lower-bounded total cost condition, we need to synthesize a non-negative cost supermartingale ϕ such that $\phi(\ell_{init}, \mathbf{x}_{init}) \leq t$.

(2) ***Tail bounds in presence of non-negative costs:*** As shown in Section 6.3, if all the costs in the pCFG C are non-negative, we need to synthesize (i) an SI-indicator (f_{SI}, p_{SI}), and (ii) a non-negative cost supermartingale ϕ for the SI-indicator, s.t. $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init})}{t} \leq p$.

(3) ***Tail bounds in presence of general costs:*** Finally, as shown in Section 6.3, to establish a tail bound on a pCFG with arbitrary costs, we need to synthesize four objects: (i) a real value K, (ii) an SI-indicator (f_{SI}, p_{SI}) satisfying certain side conditions, (iii) an RSM g proving the almost-sure termination of C_{SI}, and (iv) a non-negative cost supermartingale ϕ for (SI, p_{SI}) such that $p_{SI} + \frac{\phi(\ell_{init}, \mathbf{x}_{init}) - K}{t - K} \leq p$. Note that all these should be synthesized *simultaneously* in order to ensure that the desired tail bound is satisfied.

We present our synthesis algorithm for Case (1) above. The algorithm is exactly the same for the other cases, except that more templates and constraints should be generated to cover all the desired objects that have to be synthesized. Consider a pCFG C, together with an invariant I given as part of the input [¶]. We assume that C models an almost-surely terminating linear/polynomial program that satisfies the lower-bounded total cost condition. Similarly, $I(\ell)$ is a boolean combination of linear/polynomial inequalities at every location $\ell \in L$. Without loss of generality, we assume $I(\ell)$ is in disjunctive normal form. Our goal is to prove the expectation bound $\sup_\sigma \mathbb{E}^\sigma [Cost_C] \leq t$ by synthesizing a linear/polynomial non-negative cost supermartingale ϕ, i.e. for every location $\ell \in L$, $\phi(\ell) \in \mathbb{R}[V]$ should be a linear/polynomial expression over the program variables V. Our algorithm additionally requires a degree bound D for all the polynomials as part of its input. In the linear case, we have $D = 1$. The algorithm synthesizes ϕ in the following four steps:

Step 1. Setting Up Templates Let M be the set of all monomials of degree at most D over the program variables $V = \{x_1, \ldots, x_k\}$. More formally,

$$M = \{m_1, \ldots, m_r\} \coloneqq \{x_1^{a_1} \cdot x_2^{a_2} \cdots x_k^{a_k} \mid a_1, \ldots, a_k \in \mathbb{N}_0 \ \wedge \ a_1 + \cdots + a_k \leq D\}.$$

For every location $\ell \in L$, the algorithm sets up a template by symbolically computing

$$\phi(\ell) \coloneqq \widetilde{c_{\ell,1}} \cdot m_1 + \widetilde{c_{\ell,2}} \cdot m_2 + \cdots + \widetilde{c_{\ell,r}} \cdot m_r.$$

Here, the $\widetilde{c_{\ell,i}}$'s are new unknown variables. We shall call them *template variables*. The goal of the algorithm is to synthesize values for these unknown variables such that the resulting ϕ becomes a valid non-negative cost supermartingale.

Example 7.1. Assuming $D = 2$, our algorithm generates the following template at every location ℓ of the program in Figure 3a:

$$\phi(\ell) = \widetilde{c_{\ell,1}} + \widetilde{c_{\ell,2}} \cdot x + \widetilde{c_{\ell,3}} \cdot y + \widetilde{c_{\ell,4}} \cdot r + \widetilde{c_{\ell,5}} \cdot x^2 + \widetilde{c_{\ell,6}} \cdot x \cdot y + \widetilde{c_{\ell,7}} \cdot x \cdot r + \widetilde{c_{\ell,8}} \cdot y^2 + \widetilde{c_{\ell,9}} \cdot y \cdot r + \widetilde{c_{\ell,10}} \cdot r^2.$$

Basically, $\phi(\ell)$ is the most general template for a polynomial of degree D and our goal is to synthesize values for the coefficients of $\phi(\ell)$'s such that ϕ becomes a valid non-negative cost supermartingale.

Step 2. Generating Polynomial Entailment Constraints The algorithm symbolically computes all the constraints of a non-negative cost supermartingale, i.e. C_1 and C_2 in Definition 5.1, as well as non-negativity, using the templates generated in the previous step. Note that these constraints can all be written in the following standard form:

$$\forall \mathbf{x} \in \mathbb{R}^V \ \left(g_1(\mathbf{x}) \geq 0 \ \wedge \ g_2(\mathbf{x}) \geq 0 \ \wedge \ \cdots \ \wedge \ g_s(\mathbf{x}) \geq 0\right) \Rightarrow g(\mathbf{x}) \geq 0. \tag{4}$$

[¶]We assume that a linear/polynomial invariant is given as part of the input. Linear and polynomial invariant generation are well-studied problems, orthogonal to our work. They can be automated using [Chatterjee et al. 2020; Feautrier and Gonnord 2010; Kincaid et al. 2017; Sankaranarayanan et al. 2004].

where $g, g_1, \ldots, g_s$ are polynomials over program variables V whose coefficients might contain the template variables. We call (4) a polynomial entailment constraint.

Example 7.2. Consider the program in Figure 3a and suppose that $I(\ell_{out}) := \big(x \geq 0 \ \wedge \ y \geq 0\big)$. The algorithm symbolically computes condition C_1 of Definition 5.1 at location ℓ_{out} and obtains:

$$\forall x, y \in \mathbb{R} \quad x \geq 0 \wedge y \geq 0 \Rightarrow \phi(\ell_{out}, x, y) \geq 0.$$

which is then symbolically expanded to

$$\forall x, y \in \mathbb{R} \quad x \geq 0 \wedge y \geq 0 \Rightarrow$$
$$\widetilde{c_{\ell_{out},1}} + \widetilde{c_{\ell,2}} \cdot x + \widetilde{c_{\ell_{out},3}} \cdot y + \widetilde{c_{\ell_{out},4}} \cdot r + \widetilde{c_{\ell_{out},5}} \cdot x^2 + \widetilde{c_{\ell_{out},6}} \cdot x \cdot y + \widetilde{c_{\ell_{out},7}} \cdot x \cdot r + \widetilde{c_{\ell_{out},8}} \cdot y^2 + \widetilde{c_{\ell_{out},9}} \cdot y \cdot r + \widetilde{c_{\ell_{out},10}} \cdot r^2 \geq 0.$$

The algorithm generates similar constraints for every location $\ell \in L$ and both C_1 and C_2.

At this point, we can simply pass our systems of polynomial entailment constraints to an SMT solver. However, this is unlikely to work since the SMT solver has to synthesize values for template variables such that all the quantified constraints hold. In other words, it has to solve a formula in the first order theory of the reals with a quantifier alternation. While such formulas are decidable [Collins 1982], decision procedures for solving them are notoriously unscalable and cannot even handle toy problems [Renegar 1992].

Step 3. Dedicated Quantifier Elimination To circumvent this problem, our algorithm handles polynomial entailment constraints of form (4) using the following three methods:

(1) If g and all the g_i's are linear/affine expressions over program variables, the algorithm tries to write g as a non-negative linear combination of the g_i's. Formally, it symbolically computes:

$$\forall \mathbf{x} \in \mathbb{R}^V \quad g = \lambda_0 + \lambda_1 \cdot g_1 + \cdots + \lambda_s \cdot g_s \qquad \lambda_0, \ldots, \lambda_s \geq 0. \tag{5}$$

Here, the λ_i's are new unknown template variables whose value should be synthesized. It is clear that if g can be written as a non-negative combination of the g_i's, then g is non-negative whenever the g_i's are and hence the entailment holds. Note that both sides of (5) are linear expressions over the program variables V. So, in order for (5) to hold, the constant terms of polynomials on its LHS and its RHS should be equal and the corresponding coefficients of the program variables should also be equal on both sides. The algorithm symbolically computes these equalities and uses them to replace the entailment in (4).

(2) If all the g_i's are affine expressions over program variables but g is of a higher degree, then the algorithm tries to write g as a sum of multiplications of g_i's. More formally, we define

$$\mathcal{M} = \{\mu_1, \mu_2, \ldots, \mu_j\} := \{g_1^{a_1} \cdot g_2^{a_2} \cdots g_s^{a_s} \mid a_1, \ldots, a_s \in \mathbb{N}_0 \ \wedge \ \deg(g_1^{a_1} \cdot g_2^{a_2} \cdots g_s^{a_s}) \leq D\}.$$

In other words, $\mathcal{M}$ is the set of all polynomials of degree at most D that can be obtained as a multiplication of g_i's. It is clear that whenever the g_i's are non-negative, so are the μ_i's and hence so is any non-negative combination of the μ_i's. The algorithm symbolically computes

$$\forall \mathbf{x} \in \mathbb{R}^V \quad g = \lambda_0 + \lambda_1 \cdot \mu_1 + \cdots + \lambda_j \cdot \mu_j; \qquad \lambda_0, \ldots, \lambda_j \geq 0. \tag{6}$$

where the λ_i's are new template variables. It then equates the coefficients of corresponding monomials on the two sides and obtains quadratic constrains over the template variables.

(3) Finally, if some or all of the g_i's have a degree of at least 2, then the algorithm tries to write g as a combination of the g_i's in which every coefficient is a sum-of-squares polynomial. Concretely, the algorithm symbolically computes

$$\forall \mathbf{x} \in \mathbb{R}^V \quad g = h_0 + h_1 \cdot g_1 + \cdots + h_s \cdot g_s, \tag{7}$$

in which every h_i is a sum-of-squares polynomial. To ensure that every h_i is a sum-of-squares, the algorithm first sets up a template for each h_i just as in Step 1: $h_i := \widetilde{d_{i,1}} \cdot m_1 + \widetilde{d_{i,2}} \cdot m_2 + \cdots + \widetilde{d_{i,r}} \cdot m_r$. Here, the $\widetilde{d_{i,j}}$'s are new template variables. It then adds quadratic constraints on $\widetilde{d_{i,j}}$'s that ensure h_i is a sum-of-squares. This is a standard process and we refer to [Asadi et al. 2021,

Appendix F] for details. Finally, the algorithm equates the corresponding coefficients on the two sides of (7) and obtains quadratic constraints on the template variables.

Step 4. Solution via SMT Solver After the previous steps, the conditions in the definition of a non-negative cost supermartingale (Definition 5.1) are now soundly encoded as a Quadratic Programming (QP) instance over the template variables. The algorithm adds the boundary condition $\phi(\ell_{init}, \mathbf{x}_{init}) \leq t$ to the QP instance and passes the resulting system to an SMT solver. When the SMT solver succeeds in finding values for the template variables, the algorithm plugs these values back into the template of ϕ in Step 1 to obtain the non-negative cost supermartingale.

Soundness and Completeness It is easy to see that the synthesis algorithm above is sound since Step 2 simply encodes the definition of a non-negative cost supermartingale and Step 3 is a sound quantifier elimination. However, the procedures used in Step 3 can also preserve completeness, i.e. guarantee to find polynomial stochastic invariants and cost supermartingales of the desired degree if they exist, provided that the chosen degree D large enough. This is because Equations (5), (6) and (7) are respectively taken from Farkas' Lemma [Avis and Kaluzny 2004; Farkas 1902], Handelman's Theorem [Handelman 1988] and Putinar's Positivstellensatz [Putinar 1993], which are classical theorems in polyhedral and real algebraic geometry that provide necessary and sufficient conditions for the non-negativity/positivity of a linear/polynomial expression over a polyhedron/semi-algebraic set. See [Asadi et al. 2021] for a more detailed treatment of completeness and methods for handling non-strict inequalities.

8 EXPERIMENTAL RESULTS

We implemented a prototype of our approach for the purpose of experimentally evaluating our algorithms in Section 7. Our goal is to answer the following three research questions:

RQ1 *Expectation bound problem – new class of programs.* Can our method compute a non-negative cost supermartingale that proves an *upper bound on the expected cost* in almost-surely terminating programs that have lower-bounded total cost but which may have both positive and negative costs and unbounded variable updates, to which prior methods are not applicable?

RQ2 *Expectation bound problem – comparison to prior methods.* For programs that have either (i) non-negative costs or (ii) bounded variable updates, how do the proved upper bounds on the expected cost and the runtime of our method compare to those of prior methods?

RQ3 *Tail bound problem – new class of programs.* Can our method compute a non-negative cost supermartingale and a stochastic invariant that prove a *tail bound on cost usage* in programs with infinite expected cost usage, to which prior methods are not applicable?

As discussed in the "Limitations" paragraph in Section 1, our method for the tail bound problem uses Markov's inequality in order to be applicable to the general class of programs with possibly infinite expected cost usage. For subclasses of programs that satisfy additional assumptions, one can use tighter concentration inequalities in order to obtain better tail bounds. Hence, given that our focus is not on tight tail bounds, we do not compare our tail bound method to prior methods.

Benchmarks and Baselines To answer these questions, we consider the following benchmarks: To answer RQ1, in Table 1 we consider 6 programs that incur both positive and negative costs and have unbounded variable updates. Example 1 is the program in Fig. 3a. Examples 2-5 are crafted by the authors of this work and are provided in [Chatterjee et al. 2024, Appendix I]. Example 2 modifies the program in Fig. 3a so that updates of variables x and y are correlated. Example 3 extends the program in Fig. 3a with non-determinism. Example 4 models a stochastic chemical reaction with two types of molecules until one type becomes extinct, where cost analysis is concerned with the size of one type upon termination. Example 5 considers stochastic evolutionary process of two

species, represented by a multiplicative random walk. "Mult. Pool Mining" is the program in Fig. 2 (right).

To answer RQ2, in Table 1 we also consider 24 programs collected from [Wang et al. 2019]. The first 14[‖] programs have non-negative costs and are collected from [Wang et al. 2019, Table 2] (originally from [Ngo et al. 2018]). The latter 10 programs either have non-negative costs or bounded variable updates and are collected from [Wang et al. 2019, Table 3]. We run our method for the expectation bound problem on these programs and compare it with two baselines. The first baseline is the method of [Wang et al. 2019], which computes (not necessarily non-negative) cost supermartingales in almost-surely terminating programs with non-negative costs or with bounded variable updates. Since both methods are based on cost supermartingales, we implement the first baseline on top of our tool to allow for a fair runtime comparison. The second baseline is [Ngo et al. 2018], which follows a different approach and is applicable to almost-surely terminating programs with non-negative costs. For this, we reuse the experimental results of [Wang et al. 2019].

To answer RQ3, in Table 2 we consider 18 programs which all have infinite expected cost usage. The first 5 programs are modifications of programs in Table 1 that have infinite expected cost usage. They also have both positive and negative costs and unbounded variable updates and are provided in [Chatterjee et al. 2024, Appendix I]. The next 10 programs are probabilistic reachability examples that do not terminate almost-surely and where the goal is to compute tail bounds on termination time. These include our illustrative example in Fig. 3b and 9 more programs collected from [Chatterjee et al. 2022]. While [Chatterjee et al. 2022] considered them to bound termination probability, we consider them for the tail bounds on runtime. Finally, the last 2 programs model applications from blockchain analysis discussed in Section 2. See [Chatterjee et al. 2024, Appendix J] for details.

Experimental Setup Since in Section 4.2 we define the decision variants of both the expected value and the tail bound problem, we consider the following experimental setup:

(1) For the expectation bound problem, for each benchmark in Table 1 we evaluate our method and the first baseline by manually performing a binary search in the value of the threshold t and report the smallest threshold for which the method manages to prove the desired bound (rounded up to the closest integer value). For the second baseline, we report the results of the experimental evaluation in [Wang et al. 2019, Table 2].
(2) For the tail bound problem, for each benchmark in Table 2 we consider two different values of the threshold t. We chose values of t arbitrarily, with the goal of showing that our approach is robust and can find bounds for different threshold values. We then evaluate our method on each value of t by manually performing a binary search in the value of the probability $p \in [0, 1]$ and report the smallest value of p for which the tail bound is proved.

For both problems, we first run our tool with the maximal polynomial degree bound $D = 1$. For benchmarks where this was insufficient (results marked with the symbol †), we then use $D = 4$.

Results The data of Table 1 and 2 lead to the following conclusions:

RQ1. Our prototype tool proves an upper bound on the expected cost usage for Examples 1-5 in Table 1. Hence we conclude that our method is *practically applicable* to programs with both positive and negative costs and with unbounded variable updates, that are beyond the reach of prior methods. The runtimes of our tool are consistently small and mostly below 60s.

[‖]We exclude the benchmark `bin` as it contains a construct for the sampling instruction from binomial distribution, which is currently not supported in our prototype tool.

[**]These two benchmarks have non-negative costs and therefore satisfy the assumptions of [Ngo et al. 2018]. However, no results were reported in [Wang et al. 2019].

Table 1. Experimental results for the expectation bound problem. For each program and for each method, we list whether (1) the program satisfies the method assumptions (Sat), (2) the smallest threshold t for which the method proved the expected value bound (Bound) and (3) runtime in seconds (Time). For each benchmark and method, we manually check whether the benchmark satisfies the method assumptions. We do not include runtimes for the method of [Ngo et al. 2018] as the results are collected from [Wang et al. 2019]. The symbol † denotes usage of polynomial degree $D = 4$. We set the timeout for each experiment to 600s.

Benchmark	Our method			[Wang et al. 2019]			[Ngo et al. 2018]	
	Sat	Bound	Time	Sat	Bound	Time	Sat	Bound
Example 1 (Figure 3a)	✓	2	2.2s	✗	-	-	✗	-
Example 2	✓	2	1.0s	✗	-	-	✗	-
Example 3	✓	2	2.6s	✗	-	-	✗	-
Example 4	✓	23	1.2s	✗	-	-	✗	-
Example 5	✓	22	0.8s	✗	-	-	✗	-
Mult. Pool Mining (Figure 2 right)	✓	9795050	1.1s	✗	-	-	✗	-
[Wang et al. 2019, Table 2]								
ber	✓	201	1.0s	✓	200	1.0s	✓	200
linear01	✓	60	0.8s	✓	60	0.9s	✓	60
prdwalk	✓	120	1.3s	✓	120	1.3s	✓	120
race	✓	27	1.2s	✓	27	1.2s	✓	27
rdseql	✓	325	1.1s	✓	325	1.1s	✓	325
rdwalk	✓	202	1.1s	✓	202	1.0s	✓	202
sprdwalk	✓	202	1.1s	✓	202	1.0s	✓	200
C4B_t13	✓	225	1.4s	✓	225	1.4s	✓	225
prnes	✓	5848	1.4s	✓	5848	1.3s	✓	5848
condand	✓	199	1.2s	✓	199	1.1s	✓	200
pol04	✓	46050	2.2$s^{\dagger}$	✓	46050	1.9$s^{\dagger}$	✓	45750
pol05	✓	7975	6.4$s^{\dagger}$	✓	7975	5.2$s^{\dagger}$	✓	10100
rdbub	✓	30150	8.0$s^{\dagger}$	✓	30150	6.1$s^{\dagger}$	✓	30000
trader	✓	4954500	6.3$s^{\dagger}$	✓	4954500	4.3$s^{\dagger}$	✓	4954500
[Wang et al. 2019, Table 3]								
Bitcoin Mining (Figure 1 left)	✓	100	1.0s	✓	-146	1.0s	✗	-
Bitcoin Mining Pool	✗	-	-	✓	-77863	3.0s	✗	-
Queuing Network	✓	1000000	29$s^{\dagger}$	✓	1000000	18$s^{\dagger}$	✓	not reported**
Species Fight	✓	2531	9.0$s^{\dagger}$	✓	2531	6.7$s^{\dagger}$	✓	not reported
Figure 2	✗	-	-	✓	13400	1.8$s^{\dagger}$	✗	-
Nested Loop	✗	-	-	✓	7650	4.8$s^{\dagger}$	✗	-
Random Walk	✗	-	-	✓	-20	1.1s	✗	-
2D Robot	✗	-	-	✓	1000000	82$s^{\dagger}$	✗	-
Goods Discount	✓	1000	6.6$s^{\dagger}$	✓	-23	4.9$s^{\dagger}$	✗	-
Pollutant Disposal	✗	-	-	✓	3020	31$s^{\dagger}$	✗	-

RQ2. The results on 14 programs taken from [Wang et al. 2019, Table 2] show that our method performs on par with the existing methods on programs with non-negative costs. Our method proves the *same expectation bounds* as the method of [Wang et al. 2019] for all programs but `ber`. This is not surprising, since the restriction to *non-negative* cost supermartingales should not matter in programs with non-negative costs. The restriction does lead to a slight increase in runtime due to additional non-negativity constraints in the resulting quadratic programming instance in Step 4 of our algorithm, however the runtime increase is not significant. Our expectation bounds also mostly coincide with those of [Ngo et al. 2018], with the exception of slightly looser bounds for `ber`, `sprdwalk`, `pol04` and `rdbub` and slightly tighter bounds for `condand` and `pol05`.

Table 2. Experimental results for the tail bound problem. For each program, we list two triples consisting of (1) a threshold t, (2) the smallest probability p for which our tool proved the tail bound and (3) runtime. Each Example X Inf refers to the variant of Example X in Table 1 in with infinite expected cost usage. Each P X refers to the program in Figure X in [Chatterjee et al. 2022]. † denotes usage of polynomial degree $D = 4$.

Benchmark	Prob. p	Thresh. t	Time	Prob. p	Thresh. t	Time
Example 1 Inf	.4	40	12s	.05	200	39s
Example 2 Inf	.5	300	1.7s	.2	800	4.4s
Example 3 Inf	.1	200	22s	.05	1,750	7.3s
Example 4 Inf	.4	10^7	9.9s	.2	$1.8 \cdot 10^7$	3.5s
Example 5 Inf	.1	2,200	1.7$s^\dagger$	.05	6,000	1.6$s^\dagger$
Termination Time Bounds						
Figure 3b	.1	50	7.1$s^\dagger$	.05	90	5.6$s^\dagger$
P 1	.1	200	1.5s	.01	500	1.7s
P 5	.2	100	1.2s	.1	190	1.2s
P 7	.4	10	0.6s	.3	40	0.6s
P 8	.1	1,000	0.8s	.001	10^6	0.8s
P 10	.3	1,000	1.0s	.2	2,000	1.0s
P 11	.75	5,000	6.7s	.7	9,000	171s
P 12	.3	750	1.5s	.2	1,400	1.2s
P 17	.5	500	1.5s	.41	4,500	1.4s
P 22	.1	2,000	1.4s	.05	5,000	2.4s
Blockchain Applications						
Proof-of-Stake (Figure 1 right)	.2	1000	7.1$s^\dagger$	.01	9000	7.2$s^\dagger$
Block Withholding	.1	10^5	526s	.01	10^6	114s

The results on 10 programs taken from [Wang et al. 2019, Table 3] show that non-negative cost supermartingales and the lower bounded total cost condition are impeding factors for computing expectation bounds in programs that have both positive and negative costs. In particular, by manual inspection, we observe that `Bitcoin Pool Mining`, `Figure 2`, `Nested Loop`, `Random Walk`, `2D Robot` and `Pollutant Disposal` all model random walk-like processes where in each step the incurred cost may be either positive or negative. Such programs do not satisfy the lower bounded total cost condition, since an infinite run in the random walk which at each step incurs negative costs can achieve an arbitrarily negative total cost. While this indicates a limitation of our method, one should not view our method and [Wang et al. 2019] as competing approaches. Rather, the two methods provide different conditions under which cost supermartingales are applicable to computing expectation bounds on cost usage. In particular, our answer to the RQ1 shows that our work significantly extends the class of programs to which cost supermartingales are applicable.

RQ3. Our tool proves tail bounds on cost usage for all benchmarks considered in Table 2. This demonstrates that our method is also *practically applicable* to computing tail bounds in programs with infinite expected cost usage. The runtimes mostly remain below 60s.

Implementation Details We implemented our prototype in Python 3, using `Lark` [Shinan et al. 2023] to parse the programs, `SymPy` [Meurer et al. 2017] to manipulate symbolic expressions, `pySMT` [Gario and Micheli 2015] to handle SMT solving, and `Z3` [Moura and Bjørner 2008] and `mathsat5` [Cimatti et al. 2013] as SMT solvers. In most cases, `mathsat5` terminated faster. We ran experiments inside a Docker container on a machine with an AMD Ryzen 5 3600 CPU and 16GB of RAM.

9 RELATED WORKS

Existing approaches to cost analysis of probabilistic programs have been discussed in Section 1.

Supermartingales for other probabilistic program analyses Supermartingales are an established approaches to probabilistic program analysis. Originally, the work of [Chakarov and Sankaranarayanan 2013] introduced ranking supermartingales (RSMs) for proving almost-sure termination and since then several extensions of RSMs have been proposed for termination analysis in probabilistic programs [Abate et al. 2021; Agrawal et al. 2018; Chatterjee et al. 2016, 2018, 2023; Chen and He 2020; Fioriti and Hermanns 2015; Fu and Chatterjee 2019; Huang et al. 2019; Kenyon-Roberts and Ong 2021; McIver et al. 2018; Moosbrugger et al. 2021]. In addition to termination and cost analysis, supermartingales were also proposed for analyzing properties such as reachability and safety [Chatterjee et al. 2022, 2017; Takisaka et al. 2021] and sensitivity [Wang et al. 2020a].

Other approaches to probabilistic program analysis An established approach is based on the weakest pre-expectation calculus. Logical calculi for reasoning about probabilistic programs with non-determinism and properties such as termination, expected runtime, safety and sensitivity have been developed in [Aguirre et al. 2021; Batz et al. 2021; Feldman 1984; Kaminski et al. 2018; Kozen 1981; McIver and Morgan 2004, 2005; Olmedo et al. 2018, 2016]. The expressiveness of these logical calculi makes them applicable to a wide class of programs. However, the proofs in this calculi usually require human assistance particularly in the presence of loops. In contrast, we aim for fully automated methods for cost analysis. The work of [Monniaux 2001] used abstract interpretation to prove almost-sure termination. Sampling-based approaches with formal guarantees for termination and safety were considered in [Beutner et al. 2022; Beutner and Ong 2021].

Cores in MDPs Cores [Křetínský and Meggendorfer 2020] for finite-state Markov decision processes (MDPs) are an equivalent notion to stochastic invariants for programs [Chatterjee et al. 2017]. [Křetínský and Meggendorfer 2020] present a sampling-based method for their computation.

Cost analysis in non-probabilistic programs There are many existing methods for cost analysis in non-probabilsitic programs. Existing automated approaches are based on amortized analysis [Carbonneaux et al. 2015; Hoffmann et al. 2012a,b, 2017; Hoffmann and Hofmann 2010], abstract interpretation [Gulwani et al. 2009; Gulwani and Zuleger 2010], invariant generation [Kincaid et al. 2017], ranking functions [Alias et al. 2010], and analysis of abstract resource models [Sinn et al. 2014, 2017; Zuleger et al. 2011]. Recent works have also considered differential cost analysis that is concerned with bounding the difference in cost between two programs [Çiçek et al. 2017, 2019; Qu et al. 2019; Radicek et al. 2018; Žikelić et al. 2022].

10 CONCLUSION

We studied cost analysis of probabilistic programs with non-determinism. For the expectation bound problem, we proposed a strengthened variant of cost supermartingales which are required to be non-negative at every reachable program state and proved that they evaluate to an upper bound on cost usage in almost-surely terminating programs that satisfy the lower-bounded total cost condition. Our strengthened variant can handle programs in which incurred costs can be both positive and negative and variable updates can be unbounded, which were beyond the reach of previous methods. For tail bounds, we proposed a new method which combines quantitative safety analysis and cost analysis and can for the first time handle programs with infinite expected cost. For both analyses we presented fully automated algorithms based on template-based synthesis.

ACKNOWLEDGMENTS

This work was supported in part by the European Research Council (ERC) under Grant No. 863818 (ForM-SMArt) and the Hong Kong Research Grants Council under ECS Project No. 26208122.

DATA-AVAILABILITY STATEMENT

The software that supports Section 8 is available on Zenodo [Meggendorfer and Žikelić 2024].

REFERENCES

Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2021. Learning Probabilistic Termination Proofs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 3–26. https://doi.org/10.1007/978-3-030-81688-9_1

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

Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. A pre-expectation calculus for probabilistic sensitivity. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434333

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 772–787. https://doi.org/10.1145/3453483.3454076

David Avis and Bohdan Kaluzny. 2004. Solving inequalities and proving Farkas's lemma made easy. *The American Mathematical Monthly* 111, 2 (2004), 152–157.

Samiran Bag, Sushmita Ruj, and Kouichi Sakurai. 2016. Bitcoin block withholding attack: Analysis and mitigation. *IEEE Transactions on Information Forensics and Security* 12, 8 (2016), 1967–1978.

Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. An Assertion-Based Program Logic for Probabilistic Programs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 117–144. https://doi.org/10.1007/978-3-319-89884-1_5

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer, 255–276. https://doi.org/10.1007/978-3-030-31784-3_15

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2018. How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 186–213. https://doi.org/10.1007/978-3-319-89884-1_7

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2021. Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434320

Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. 2022. Guaranteed bounds for posterior inference in universal probabilistic programming. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 536–551. https://doi.org/10.1145/3519939.3523721

Raven Beutner and Luke Ong. 2021. On probabilistic termination of functional programs with continuous distributions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1312–1326. https://doi.org/10.1145/3453483.3454111

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 467–478. https://doi.org/10.1145/2737924.2737955

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

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 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 672–687. https://doi.org/10.1145/3385412.3385969

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2018. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *TOPLAS* 40, 2 (2018), 7:1–7:45. https://doi.org/10.1145/3174800

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Đorđe Žikelić. 2022. Sound and Complete Certificates for Quantitative Termination Analysis of Probabilistic Programs. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 55–78. https://doi.org/10.1007/978-3-031-13185-1_4

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Đorđe Žikelić. 2024. Quantitative Bounds on Resource Usage of Probabilistic Programs. https://hal.science/hal-04491689

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2021. Proving non-termination by program reversal. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1033–1048. https://doi.org/10.1145/3453483.3454093

Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, Jiri Zárevúcky, and Dorde Zikelic. 2023. On Lexicographic Proof Rules for Probabilistic Termination. *Formal Aspects Comput.* 35, 2 (2023), 11:1–11:25. https://doi.org/10.1145/3585391

Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. 2017. Stochastic Invariants for Probabilistic Termination. In *POPL*. 145–160. https://doi.org/10.1145/3009837.3009873

Jianhui Chen and Fei He. 2020. Proving almost-sure termination by omega-regular decomposition. In *PLDI*. 869–882. https://doi.org/10.1145/3385412.3386002

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 316–329. https://doi.org/10.1145/3009837.3009858

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 533–547. https://doi.org/10.1145/3314221.3314603

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 93–107.

GE Collins. 1982. Quantifier elimination for real closed fields: a guide to the literature. *Computer Algebra* (1982), 79–81.

Micah Dameron. 2018. Beigepaper: an ethereum technical specification. *Ethereum Foundation* (2018).

Julius Farkas. 1902. Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1902, 124 (1902), 1–27.

Paul Feautrier and Laure Gonnord. 2010. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. *Electronic Notes in Theoretical Computer Science* 267, 2 (2010), 3 – 13. https://doi.org/10.1016/j.entcs.2010.09.014 Proceedings of the Tools for Automatic Program AnalysiS (TAPAS).

Yishai A. Feldman. 1984. A decidable propositional dynamic logic with explicit probabilities. *Information and Control* 63, 1 (1984), 11–38. https://doi.org/10.1016/S0019-9958(84)80039-X

Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *POPL*. 489–501. https://doi.org/10.1145/2676726.2677001

Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *VMCAI*. 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

Marco Gario and Andrea Micheli. 2015. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*.

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 (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nat.* 521, 7553 (2015), 452–459. https://doi.org/10.1038/nature14541

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP*. 51–68.

Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: a language for generative models. In *UAI*. AUAI Press, 220–229.

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 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 167–181. https://doi.org/10.1145/2593882.2593900

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 292–304. https://doi.org/10.1145/1806596.1806630

Alireza Toroghi Haghighat and Mehdi Shajari. 2019. Block withholding game among bitcoin mining pools. *Future Gener. Comput. Syst.* 97 (2019), 482–491.

David Handelman. 1988. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.* 132, 1 (1988), 35–62.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012a. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012b. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular verification for almost-sure termination of probabilistic programs. In *OOPSLA*. 129:1–129:29. https://doi.org/10.1145/3360555

Jan Křetínský and Tobias Meggendorfer. 2020. Of Cores: A Partial-Exploration Framework for Markov Decision Processes. *Logical Methods in Computer Science* (Oct. 2020). https://doi.org/10.23638/LMCS-16(4:3)2020

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018), 30:1–30:68. https://doi.org/10.1145/3208102

Andrew Kenyon-Roberts and C.-H. Luke Ong. 2021. Supermartingales, Ranking Functions and Probabilistic Lambda Calculus. In *LICS*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470550

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *CRYPTO (1) (Lecture Notes in Computer Science, Vol. 10401)*. Springer, 357–388.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 248–262. https://doi.org/10.1145/3062341.3062373

Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. System Sci.* 22, 3 (1981), 328–350. https://doi.org/10.1016/0022-0000(81)90036-2

Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. 2019. Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11428)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 135–153. https://doi.org/10.1007/978-3-030-17465-1_8

Yoad Lewenberg, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S. Rosenschein. 2015. Bitcoin Mining Pools: A Cooperative Game Theoretic Analysis. In *AAMAS*. 919–927.

Annabelle McIver and Carroll Morgan. 2004. Developing and Reasoning About Probabilistic Programs in *pGCL*. In *Refinement Techniques in Software Engineering, First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3167)*, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock (Eds.). Springer, 123–155. https://doi.org/10.1007/11889229_4

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer. https://doi.org/10.1007/b138392

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

Tobias Meggendorfer and Đorđe Žikelić. 2024. *Artefact for: Quantitative Bounds on Resource Usage of Probabilistic Programs*. https://doi.org/10.5281/zenodo.10457566

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

David Monniaux. 2001. An Abstract Analysis of the Probabilistic Termination of Programs. In *SAS*. 111–126. https://doi.org/10.1007/3-540-47764-0_7

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021. Automated Termination Analysis of Polynomial Probabilistic Programs. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 491–518. https://doi.org/10.1007/978-3-030-72019-3_18

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Martin R. Neuhäußer and Joost-Pieter Katoen. 2007. Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 412–427. https://doi.org/10.1007/978-3-540-74407-8_28

Martin R. Neuhäußer, Mariëlle Stoelinga, and Joost-Pieter Katoen. 2009. Delayed Nondeterminism in Continuous-Time Markov Decision Processes. In *FOSSACS*. 364–379. https://doi.org/10.1007/978-3-642-00596-1_26

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 496–512. https://doi.org/10.1145/3192366.3192394

Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. 2018. Conditioning in Probabilistic Programming. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 4:1–4:50. https://doi.org/10.1145/3156018

Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning About Recursive Probabilistic Programs. In *LICS*. 672–681. https://doi.org/10.1145/2933575.2935317

Mihai Putinar. 1993. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984.

Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational cost analysis for functional-imperative programs. *Proc. ACM Program. Lang.* 3, ICFP (2019), 92:1–92:29. https://doi.org/10.1145/3341696

Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

James Renegar. 1992. On the computational complexity and geometry of the first-order theory of the reals. Part III: Quantifier elimination. *Journal of Symbolic Computation* 13, 3 (1992), 329–352.

DM Roy, VK Mansinghka, ND Goodman, and JB Tenenbaum. 2008. A stochastic programming perspective on nonparametric Bayes. In *ICML*, Vol. 22. 26.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

Erez Shinan et al. 2023. Lark - a parsing toolkit for Python. https://github.com/lark-parser/lark/

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *J. Autom. Reason.* 59, 1 (2017), 3–45. https://doi.org/10.1007/s10817-016-9402-4

Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 5:1–5:46. https://doi.org/10.1145/3450967

Sebastian Thrun. 2000. Probabilistic Algorithms in Robotics. *AI Mag.* 21, 4 (2000), 93–109. https://doi.org/10.1609/aimag.v21i4.1534

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *IFL 2016*. ACM, 6:1–6:12. https://doi.org/10.1145/3064899.3064910

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. *CoRR* abs/1809.10756 (2018). arXiv:1809.10756 http://arxiv.org/abs/1809.10756

Đorđe Žikelić, Bor-Yuh Evan Chang, Pauline Bolignano, and Franco Raimondi. 2022. Differential cost analysis with simultaneous potentials and anti-potentials. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 442–457. https://doi.org/10.1145/3519939.3523435

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Central moment analysis for cost accumulators in probabilistic programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 559–573. https://doi.org/10.1145/3453483.3454062

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

Peixin Wang, Hongfei Fu, Krishnendu Chatterjee, Yuxin Deng, and Ming Xu. 2020a. Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time. In *POPL*. 25:1–25:30. https://doi.org/10.1145/3371093

Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi. 2019. Cost analysis of nondeterministic probabilistic programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 204–220. https://doi.org/10.1145/3314221.3314581

D. Williams. 1991. *Probability with Martingales.* Cambridge University Press, Cambridge, UK. 251 pages.

Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22

Received 21-OCT-2023; accepted 2024-02-24