Cost Analysis of Nondeterministic Probabilistic Programs

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, Peixin Wang, Xudong Qin, Wenjun Shi

Cost Analysis of Nondeterministic Probabilistic Programs

Krishnendu Chatterjee
IST Austria
krishnendu.chatterjee@ist.ac.at

Hongfei Fu
Shanghai Jiao Tong University
fuhf@cs.sjtu.edu.cn

Amir Kafshdar Goharshady
IST Austria
amir.goharshady@ist.ac.at

Peixin Wang
Shanghai Jiao Tong University
wangpeixin@sjtu.edu.cn

Xudong Qin
East China Normal University
52174501019@stu.ecnu.edu.cn

Wenjun Shi
East China Normal University
51174500041@stu.ecnu.edu.cn

Abstract

We consider the problem of expected cost analysis over nondeterministic probabilistic programs, which aims at automated methods for analyzing the resource-usage of such programs. Previous approaches for this problem could only handle nonnegative bounded costs. However, in many scenarios, such as queuing networks or analysis of cryptocurrency protocols, both positive and negative costs are necessary and the costs are unbounded as well.

In this work, we present a sound and efficient approach to obtain polynomial bounds on the expected accumulated cost of nondeterministic probabilistic programs. Our approach can handle (a) general positive and negative costs with bounded updates in variables; and (b) nonnegative costs with general updates to variables. We show that several natural examples which could not be handled by previous approaches are captured in our framework.

Moreover, our approach leads to an efficient polynomial-time algorithm, while no previous approach for cost analysis of probabilistic programs could guarantee polynomial runtime. Finally, we show the effectiveness of our approach by presenting experimental results on a variety of programs, motivated by real-world applications, for which we efficiently synthesize tight resource-usage bounds.

1 Introduction

In this work, we consider expected cost analysis of nondeterminisitic probabilistic programs, and present a sound and efficient approach for a large class of such programs. We start with the description of probabilistic programs and the cost analysis problem, and then present our contributions.

Probabilistic programs. Extending classical imperative programs with randomization, i.e. generation of random values according to a predefined probability distribution, leads to the class of probabilistic programs [45]. Probabilistic programs are shown to be powerful models for a wide variety of applications, such as analysis of stochastic network protocols [40, 65, 87], machine learning applications [26, 44, 80, 82], and robot planning [90, 91], to name a few. There are also many probabilistic programming languages (such as Church [42], Anglican [92] and WebPPL [43]) and automated analysis of such programs is an active research area in formal methods and programming languages (see [1, 16, 18, 22, 37, 66, 67, 74, 95]).

Nondeterministic programs. Besides probability, another important modeling concept in programming languages is nondeterminism. A classic example is abstraction: for efficient static analysis of large programs, it is often infeasible to keep track of all variables. Abstraction ignores some variables and replaces them with worst-case behavior, which is modeled by nondeterminism [32].

Termination and cost analysis. The most basic liveness question for probabilistic programs is *termination*. The basic qualitative questions for termination of probabilistic programs, such as, whether the program terminates with probability 1 or whether the expected termination time is bounded, have been widely studied [18, 22, 66, 67]. However, in program analysis, the more general quantitative task of obtaining precise bounds on resource-usage is a challenging problem that is of significant interest for the following reasons: (a) in applications such as hard real-time systems, guarantees of worst-case behavior are required; and (b) the bounds are useful in early detection of egregious performance problems in large code bases. Works such as [48, 49, 53, 54] provide excellent motivation for the study of automated methods to obtain worst-case bounds for resource-usage of nonprobabilistic programs. The same motivation applies to the class

of probabilistic programs as well. Thus, the problem we consider is as follows: given a probabilistic program with costs associated to each execution step, compute bounds on its expected accumulated cost until its termination.

Previous approaches. While there is a large body of work for qualitative termination analysis problems (see Section 9 for details), the cost analysis problem has only been considered recently. The most relevant previous work for cost analysis is that of Ngo, Carbonneaux and Hoffmann [74], which considers the stepwise costs to be nonnegative and bounded. While several interesting classes of programs satisfy the above restrictions, there are many natural and important classes of examples that cannot be modeled in this framework. For example, in the analysis of cryptocurrency protocols, such as mining, there are both energy costs (positive costs) and solution rewards (negative costs). Similarly, in the analysis of queuing networks, the cost is proportional to the length of the queues, which might be unbounded. For concrete motivating examples see Section 3.

Our contribution. In this work, we present a novel approach for synthesis of polynomial bounds on the expected accumulated cost of nondeterministic probabilistic programs.

1. Our sound framework can handle the following cases: (a) general positive and negative costs, with bounded updates to the variables at every step of the execution; and (b) nonnegative costs with general updates (i.e. unbounded costs and unbounded updates to the variables). In the first case, our approach obtains both upper and lower bounds, whereas in the second case we only obtain upper bounds. In contrast, previous approaches only provide upper bounds for bounded nonnegative costs. A key technical novelty of our approach is an extension of the classical Optional Stopping Theorem (OST) for martingales.
2. We present a sound algorithmic approach for the synthesis of polynomial bounds. Our algorithm runs in polynomial time. Note that no previous approach provides polynomial runtime guarantee for synthesis of such bounds for nondeterministic probabilistic programs. Our synthesis approach is based on application of results from semi-algebraic geometry.
3. Finally, we present experimental results on a variety of programs, which are motivated from applications such as cryptocurrency protocols, stochastic linear recurrences, and queuing networks, and show that our approach can efficiently obtain tight polynomial resource-usage bounds.

We start with preliminaries (Section 2) and then present a set of motivating examples (Section 3). Then, we provide an overview of the main technical ideas of our approach in Section 4. The following sections each present technical details of one of the steps of our approaches.

2 Preliminaries

In this section, we define some necessary notions from probability theory and probabilistic programs. We also formally define the expected accumulated cost of a program.

2.1 Martingales

We start by reviewing some notions from probability theory. We consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ where Ω is the sample space, $\mathcal{F}$ is the set of events and $\mathbb{P} : \mathcal{F} \to [0, 1]$ is the probability measure.

Random variables. A *random variable* is an $\mathcal{F}$-measurable function $X : \Omega \to \mathbb{R} \cup \{+\infty, -\infty\}$, i.e. a function satisfying the condition that for all $d \in \mathbb{R} \cup \{+\infty, -\infty\}$, the set of all points in the sample space with an X value of less than d belongs to $\mathcal{F}$.

Expectation. The *expected value* of a random variable X, denoted by $\mathbb{E}(X)$, is the Lebesgue integral of X wrt $\mathbb{P}$. See [96] for the formal definition of Lebesgue integration. If the range of X is a countable set A, then $\mathbb{E}(X) = \sum_{\omega \in A} \omega \cdot \mathbb{P}(X = \omega)$.

Filtrations and stopping times. A *filtration* of the probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is an infinite sequence $\{\mathcal{F}_n\}_{n=0}^{\infty}$ such that for every n, the triple $(\Omega, \mathcal{F}_n, \mathbb{P})$ is a probability space and $\mathcal{F}_n \subseteq \mathcal{F}_{n+1} \subseteq \mathcal{F}$. A *stopping time* wrt $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a random variable $U : \Omega \to \mathbb{N} \cup \{0, \infty\}$ such that for every $n \geq 0$, the event $U \leq n$ is in $\mathcal{F}_n$. Intuitively, U is interpreted as the time at which the stochastic process shows a desired behavior.

Discrete-time stochastic processes. A *discrete-time stochastic process* is a sequence $\Gamma = \{X_n\}_{n=0}^{\infty}$ of random variables in $(\Omega, \mathcal{F}, \mathbb{P})$. The process Γ is *adapted* to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$, if for all $n \geq 0$, X_n is a random variable in $(\Omega, \mathcal{F}_n, \mathbb{P})$.

Martingales. A discrete-time stochastic process $\Gamma = \{X_n\}_{n=0}^{\infty}$ adapted to a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a *martingale* (resp. *supermartingale*, *submartingale*) if for all $n \geq 0$, $\mathbb{E}(|X_n|) < \infty$ and it holds almost surely (i.e., with probability 1) that

$\mathbb{E}(X_{n+1}|\mathcal{F}_n) = X_n$ (resp. $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \leq X_n$, $\mathbb{E}(X_{n+1}|\mathcal{F}_n) \geq X_n$). See [96] for details.

Intuitively, a martingale is a discrete-time stochastic process, in which at any time n, the expected value $\mathbb{E}(X_{n+1}|\mathcal{F}_n)$ in the next step, given all previous values, is equal to the current value X_n. In a supermartingale, this expected value is less than or equal to the current value and a submartingale is defined conversely. Applying martingales for termination analysis is a well-studied technique [16, 18, 24].

2.2 Nondeterministic Probabilistic Programs

We now fix the syntax and semantics of the nondeterministic probabilistic programs we consider in this work.

Syntax. Our nondeterministic probabilistic programs are imperative programs with the usual conditional and loop structures (i.e. **if** and **while**), as well as the following new structures: (a) probabilistic branching statements of the form "**if prob**(p) ..." that lead to the **then** part with probability p and to the **else** part with probability $1-p$, (b) nondeterministic branching statements of the form "**if** $\star$ ..." that nondeterministically lead to either the **then** part or the **else** part, and (c) statements of the form **tick**(q) whose execution triggers a cost of q. Moreover, the variables in our programs can either be program variables, which act in the usual way, or sampling variables, whose values are randomly sampled from predefined probability distributions each time they are accessed in the program.

Formally, nondeterministic probabilistic programs are generated by the grammar in Figure 1. In this grammar $\langle pvar \rangle$ (resp. $\langle rvar \rangle$) expressions range over program (resp. sampling) variables. For brevity, we omit the **else** part of the conditional statements if it contains only a single **skip**. See Appendix B for more details about the syntax.

An example program is given in Figure 2(left). Note that the complete specification of the program should also include distributions from which the sampling variables are sampled.

Labels. We refer to the status of the program counter as a *label*, and assign labels ℓ_{in} and ℓ_{out} to the start and end of the program, respectively. Our label types are as follows:

- An *assignment* label corresponds to an assignment statement indicated by := or **skip**. After its execution, the value of the expression on its right hand side is stored in the variable on its left hand side and control flows to the next statement. A **skip** assignment does not change the value of any variable.

$$
\begin{aligned}
\langle stmt \rangle &::= \text{`}\mathbf{skip}\text{'} \\
&\quad | \ \langle pvar \rangle \text{ `:='} \langle expr \rangle \\
&\quad | \ \text{`}\mathbf{if}\text{'} \langle bexpr \rangle \text{ `}\mathbf{then}\text{'} \langle stmt \rangle \text{ `}\mathbf{else}\text{'} \langle stmt \rangle \text{ `}\mathbf{fi}\text{'} \\
&\quad | \ \text{`}\mathbf{if}\text{'} \text{ `}\mathbf{prob}\text{'} \text{ `(}p\text{`)'} \text{ `}\mathbf{then}\text{'} \langle stmt \rangle \text{ `}\mathbf{else}\text{'} \langle stmt \rangle \text{ `}\mathbf{fi}\text{'} \\
&\quad | \ \text{`}\mathbf{if}\text{'} \text{ `}\star\text{'} \text{ `}\mathbf{then}\text{'} \langle stmt \rangle \text{ `}\mathbf{else}\text{'} \langle stmt \rangle \text{ `}\mathbf{fi}\text{'} \\
&\quad | \ \text{`}\mathbf{while}\text{'} \langle bexpr \rangle \text{ `}\mathbf{do}\text{'} \langle stmt \rangle \text{ `}\mathbf{od}\text{'} \\
&\quad | \ \text{`}\mathbf{tick}\text{'`('}\langle pexpr \rangle\text{`)'} \mid \langle stmt \rangle \text{ `;'} \langle stmt \rangle \\
\langle literal \rangle &::= \langle pexpr \rangle \text{ `}\leq\text{'} \langle pexpr \rangle \mid \langle pexpr \rangle \text{ `}\geq\text{'} \langle pexpr \rangle \\
\langle bexpr \rangle &::= \langle literal \rangle \mid \text{`}\neg\text{'}\langle bexpr \rangle \\
&\quad | \ \langle bexpr \rangle \text{ `}\mathbf{or}\text{'} \langle bexpr \rangle \mid \langle bexpr \rangle \text{ `}\mathbf{and}\text{'} \langle bexpr \rangle \\
\langle pexpr \rangle &::= \langle constant \rangle \mid \langle pvar \rangle \mid \langle pexpr \rangle \text{ `}*\text{'} \langle pexpr \rangle \\
&\quad | \ \langle pexpr \rangle \text{ `+'} \langle pexpr \rangle \mid \langle pexpr \rangle \text{ `}-\text{'} \langle pexpr \rangle \\
\langle expr \rangle &::= \langle constant \rangle \mid \langle pvar \rangle \mid \langle rvar \rangle \mid \langle expr \rangle \text{ `}*\text{'} \langle expr \rangle \\
&\quad | \ \langle expr \rangle \text{ `+'} \langle expr \rangle \mid \langle expr \rangle \text{ `}-\text{'} \langle pexpr \rangle
\end{aligned}
$$

Figure 1. Syntax of nondeterministic probabilistic programs.

- A *branching* label corresponds to a conditional statement, i.e. either an "**if** ϕ ..." or a "**while** ϕ ...", where ϕ is a condition on program variables, and the next statement to be executed depends on whether ϕ is satisfied or not.
- A *probabilistic* label corresponds to an "**if prob**(p) ..." with $p \in [0,1]$, and leads to the **then** branch with probability p and the **else** branch with probability $1-p$.
- A *nondeterministic* label corresponds to a nondeterministic branching statement indicated by "**if** $\star$ ...", and is nondeterministically followed by either the **then** branch or the **else** branch.
- A *tick* label corresponds to a statement **tick**(q) that triggers a cost of q, and leads to the next label. Note that q is an arithmetic expression, serving as the stepwise *cost function*, and can depend on the values of program variables.

Valuations. Given a set V of variables, a valuation over V is a function $\mathbf{v} : V \to \mathbb{R}$ that assigns a value to each variable. We denote the set of all valuations on V by Val_V.

Control flow graphs (CFGs) [6]. We define control flow graphs of our programs in the usual way, i.e. a CFG contains one vertex for each label and an edge connects a label ℓ_i to

another label ℓ_j, if ℓ_j can possibly be executed right after ℓ_i by the rules above. Formally, a CFG is a tuple

$$\big(V_\mathrm{p}, V_\mathrm{r}, L, \to\big) \tag{1}$$

where:

- V_p and V_r are finite sets of *program variables* and *sampling (randomized) variables*, respectively;
- L is a finite set of *labels* partitioned into (i) the set L_a of *assignment* labels, (ii) the set L_b of *branching* labels, (iii) the set L_p of *probabilistic* labels, (iv) the set L_nd of *nondeterministic* labels, (v) the set L_t of *tick* labels, and (vi) a special terminal label ℓ_out corresponding to the end of the program. Note that the start label ℓ_in corresponds to the first statement of the program and is therefore covered in cases (i)–(v).
- $\to$ is a transition relation whose every member is a triple of the form (ℓ, α, ℓ') where ℓ is the source and ℓ' is the target of the transition, and α is the rule that must be obeyed when the execution goes from ℓ to ℓ'. The rule α is either an *update function* $F_\ell : \mathit{Val}_{V_\mathrm{p}} \times \mathit{Val}_{V_\mathrm{r}} \to \mathit{Val}_{V_\mathrm{p}}$ if $\ell \in L_\mathrm{a}$, which maps values of program and sampling variables before the assignment to the values of program variables after the assignment, or a condition ϕ over V_p if $\ell \in L_\mathrm{b}$, or a real number $p \in [0,1]$ if $\ell \in L_\mathrm{p}$, or $\star$ if $\ell \in L_\mathrm{nd}$, or a *cost function* $R_\ell : \mathit{Val}_{V_\mathrm{p}} \to \mathbb{R}$ if $\ell \in L_\mathrm{t}$. In the last case, the cost function R_ℓ is specified by the arithmetic expression q in **tick**(q) and maps the values of program variables to the cost of the tick operation.

Example 2.1. Figure 2 provides an example program and its CFG. We assume that the probability distributions for the random variables r and r' are $(1,-1) : (1/4, 3/4)$ and $(1,-1) : (2/3, 1/3)$ respectively. In this program, the value of the variable x is incremented by the sampling variable r, whose value is 1 with probability $1/4$ and -1 with probability $3/4$. Then, the variable y is assigned a random value sampled from the variable r', that is 1 with probability $2/3$ and -1 with probability $1/3$. The **tick** command then incurs a cost of $x \cdot y$, i.e. $x * y$ is used as the cost function.

```
1 : while x ≥ 1 do
2 :     x := x + r;
3 :     y := r';
4 :     tick(x * y)
    od
5 :
```

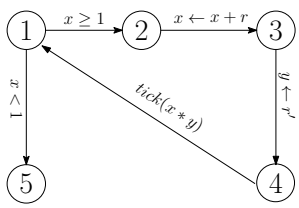

Figure 2. An example program with its labels (left), and its CFG (right). We have $\ell_\mathrm{in} = 1$ and $\ell_\mathrm{out} = 5$.

Runs and schedulers. A(n infinite) *run* of a program is an infinite sequence $\{(\ell_n, \mathbf{v}_n)\}_{n=0}^{\infty}$ of labels ℓ_n and valuations $\mathbf{v}_n$ to program variables that respects the rules of the CFG. A *scheduler* is a policy that chooses the next step, based on the history of the program, when the program reaches a nondeterministic choice. For more formal semantics see Appendix C.

Termination time [38]. The *termination time* is a random variable T defined on program runs as $T(\{(\ell_n, \mathbf{v}_n)\}_{n=0}^{\infty}) := \min\{n \mid \ell_n = \ell_\mathrm{out}\}$. We define $\min \emptyset := \infty$. Note that T is a stopping time on program runs. Intuitively, the termination time of a run is the number of steps it takes for the run to reach the termination label ℓ_out or ∞ if it never terminates.

Types of termination [18, 38, 67]. A program is said to *almost surely terminate* if it terminates with probability 1 using any scheduler. Similarly, a program is *finitely terminating* if it has finite expected termination time over all schedulers. Finally, a program has the *concentration property* or *concentratedly terminates* if there exist positive constants a and b such that for sufficiently large n, we have $\mathbb{P}(T > n) \leq a \cdot e^{-b \cdot n}$ for all schedulers, i.e. if the probability that the program takes n steps or more decreases exponentially as n grows.

Termination analysis of probabilistic programs is a widely-studied topic. For automated approaches, see [1, 16, 18, 71].

2.3 Expected Accumulated Cost

The main notion we use in cost analysis of nondeterministic probabilistic programs is the expected accumulated cost until program termination. This concept naturally models the total cost of execution of a program in the average case. We now formalize this notion.

Cost of a run. We define the random variable C_m as the cost at the m-th step in a run, which is equal to a cost function R_ℓ if the m-th executed statement is a tick statement and is

zero otherwise, i.e. given a run $\rho = \{(\ell_n, \mathbf{v}_n)\}_{n=0}^{\infty}$, we define:

$$C_m(\rho) := \begin{cases} R_{\ell_m}(\mathbf{v}_m) & \text{if } \ell_m \in L_t \\ 0 & \text{otherwise} \end{cases}$$

Moreover, we define the random variable C_∞ as the total cost of all steps, i.e. $C_\infty(\rho) := \sum_{m=0}^{\infty} C_m(\rho)$. Note that when the program terminates, the run remains in the state ℓ_{out} and does not trigger any costs. Hence, C_∞ represents the total accumulated cost until termination. Given a scheduler σ and an initial valuation $\mathbf{v}$ to program variables, we define $\mathbb{E}_{\mathbf{v}}^{\sigma}(C_\infty)$ as the expected value of the random variable C_∞ over all runs that start with $(\ell_{\text{in}}, \mathbf{v})$ and use σ for making choices at nondeterministic points.

Definition 2.2 (Expected Accumulated Cost)**.** Given an initial valuation $\mathbf{v}$ to program variables, the *maximum expected accumulated cost*, *supval*$(\mathbf{v})$, is defined as $\sup_\sigma \mathbb{E}_{\mathbf{v}}^{\sigma}(C_\infty)$, where σ ranges over all possible schedulers.

Intuitively, *supval*$(\mathbf{v})$ is the maximum expected total cost of the program until termination, i.e. assuming a scheduler that resolves nondeterminism to maximize the total accumulated cost.

In this work, we focus on automated approaches to find polynomial bounds for *supval*$(\mathbf{v})$.

3 Motivating Examples

In this section, we present several motivating examples for the expected cost analysis of nondeterministic probabilistic programs. Previous general approaches for probabilistic programs, such as [74], require the following restrictions: (a) stepwise costs are nonnegative; and (b) stepwise costs are bounded. We present natural examples which do not satisfy the above restrictions. Our examples are as follows:

1. In Section 3.1, we present an example of Bitcoin mining, where the costs are both positive and negative, but bounded. Then in Section 3.2, we present an example of Bitcoin pool mining, where the costs are both positive and negative, as well as unbounded, but the updates to the variables at each program execution step are bounded.
2. In Section 3.3, we present an example of queuing networks which also has unbounded costs but bounded updates to the variables.
3. In Section 3.4, we present an example of stochastic linear recurrences, where the costs are nonnegative but unbounded, and the updates to the variable values are also unbounded.

3.1 Bitcoin Mining

Popular decentralized cryptocurrencies, such as Bitcoin and Ethereum, rely on proof-of-work Blockchain protocols to ensure a consensus about ownership of funds and validity of transactions [73, 94]. In these protocols, a subset of the nodes of the cryptocurrency network, called *miners*, repeatedly try to solve a computational puzzle. In Bitcoin, the puzzle is to invert a hash function, i.e. to find a nonce value v, such that the SHA256 hash of the state of the Blockchain and the nonce v becomes less than a predefined value [73]. The first miner to find such a nonce is rewarded by a fixed number of bitcoins. If several miners find correct nonces at almost the same time, which happens with very low probability, only one of them will be rewarded and the solutions found by other miners will get discarded [12].

Given the one-way property of hash functions, the only strategy for a miner is to constantly try randomly-generated nonces until one of them leads to the desired hash value. Therefore, a miner's chance of getting the next reward is proportional to her computational power. Bitcoin mining uses considerable electricity and is therefore very costly [34].

Bitcoin mining can be modeled by the nondeterministic probabilistic program given in Figure 3. In this program, a miner starts with an initial balance of x and mines as long as he has some money left for the electricity costs. At each step, he generates and checks a series of random nonces. This leads to a cost of α for electricity. With probability p, one of the generated nonces solves the puzzle. When this happens, with probability p' the current miner is the only one who has solved the puzzle and receives a reward of β units. However, with probability $1 - p'$, other miners have also solved the same puzzle in roughly the same time. In this case, whether the miner receives his reward or not is decided by nondeterminism. The values of parameters α, β, p, and p' can be found experimentally in the real world. Basically, α is the cost of electricity for the miner, which depends on location, β is the reward for solving the puzzle, which depends on the Bitcoin exchange rate, and p and p' depend on the total computational power of the Bitcoin network, which can be estimated at any time [86]. In the sequel, we assume $\alpha = 1, \beta = 5000, p = 0.0005, p' = 0.99$.

Remark 1. Note that in the example of Figure 3, the costs are both positive (**tick**(α)) and negative (**tick**$(-\beta)$), but bounded by the constants $|\alpha|$ and $|\beta|$. Also all updates to the program variable x are bounded by $|\alpha|$.

```
while x ≥ α do
        x := x − α; tick(α);
        if prob(p) then
           if prob(p′) then tick(−β)
           else if ⋆ then tick(−β)
         fi fi fi od
```

Figure 3. Bitcoin mining

3.2 Bitcoin Pool Mining

As mentioned earlier, a miner's chance of solving the puzzle in Bitcoin is proportional to her computational power. Given that the overall computational power of the Bitcoin network is enormous, there is a great deal of variance in miners' revenues, e.g. a miner might not find a solution for several months or even years, and then suddenly find one and earn a huge reward. To decrease the variance in their revenues, miners often collaborate in *mining pools* [79].

A mining pool is created by a manager who guarantees a steady income for all participating miners. This income is proportional to the miner's computational power. Any miner can join the pool and assign its computational power to solving puzzles for the pool, instead of for himself, i.e. when a puzzle is solved by a miner participating in a pool, the rewards are paid to the pool manager [23]. Pools charge participation fees, so in the long term, the expected income of a participating miner is less than what he is expected to earn by mining on his own.

A pool can be modeled by the probabilistic program in Figure 4. The manager starts the pool with y identical miners[1]. At each time step, the manager has to pay each miner a fixed amount α. Miners perform the mining as in Figure 3. Note that their mining revenue now belongs to the pool manager. Finally, at each time step, a small stochastic change happens in the number of miners, i.e. a miner might choose to leave the pool or a new miner might join the pool. The probability of such changes can also be estimated experimentally. In our example, we have that the number of miners increases by one with probability 0.4, decrease by one with probability 0.5, and does not change with probability 0.1 ($y := y + (-1, 0, 1) : (0.5, 0.1, 0.4)$).

Remark 2. In contrast to Figure 3 where the costs are bounded, in Figure 4, they are not bounded ($\mathbf{tick}(\alpha * y)$). Moreover, they are both positive ($\mathbf{tick}(\alpha * y)$) and negative ($\mathbf{tick}(-\beta)$). However, note that the changes to the program variables i, y are bounded.

[1]This assumption does not affect the generality of our modeling. If the miners have different computational powers, a more powerful miner can be modeled as a union of several less powerful miners.

```
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)
      fi fi fi; i := i + 1 od;
    y := y + (−1, 0, 1) : (0.5, 0.1, 0.4) od
```

Figure 4. Bitcoin pool mining

3.3 Queuing Networks

A well-studied structure for modeling parallel systems is the *Fork and Join* (FJ) queuing network [10]. An FJ network consists of K processors, each with its own dedicated queue (Figure 5). When a job arrives, the network probabilistically divides (*forks*) it into one or more parts and assigns each part to one of the processors by adding it to the respective queue. Each processor processes the jobs in its queue on a first-in-first-out basis. When all of the parts of a job are processed, the results are *joined* and the job is completed. The *processing time* of a job is the amount of time it takes from its arrival until its completion.

FJ networks have been used to model and analyze the efficiency of a wide variety of parallel systems [10], such as web service applications [72], complex network intrusion detection systems [9], MapReduce frameworks [35], programs running on multi-core processors [52], and health care applications such as diagnosing patients based on test results from several laboratories [8].

An FJ network can be modeled as a probabilistic program. For example, the program in Figure 6 models a network with $K = 2$ processors that accepts jobs for n time units. At each unit of time, one unit of work is processed from each queue, and there is a fixed probability 0.02 that a new job arrives. The network then probabilistically decides to assign the job to the first processor (with probability 0.2) or the second processor (with probability 0.4) or to divide it among them (with probability 0.4). We assume that all jobs are identical and for processor 1 it takes 3 time units to process a job, while processor 2 only takes 2 time units. If the job is divided among them, processor 1 takes 2 units to finish its part and

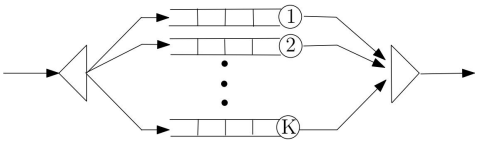

Figure 5. A Fork and Join network with K processors

```
l1 := 0;   l2 := 0;   i := 1;
while i ≤ n do
  if l1 ≥ 1 then l1 := l1 − 1 fi;
  if l2 ≥ 1 then l2 := l2 − 1 fi;
  if prob (0.02) then
    if prob(0.2) then
      l1 := l1 + 3
    else if prob(0.5) then
      l2 := l2 + 2
    else
      l1 := l1 + 2; l2 := l2 + 1
    fi fi;
    if l1 ≥ l2 then tick(l1) else tick(l2) fi
  fi; i := i + 1 od
```

Figure 6. A FJ-network Example with $K = 2$ Processors

processor 2 takes 1 time unit. The variables l_1 and l_2 model the length of the queues for each processor, and the program cost models the total processing time of the jobs.

Note that the processing time is computed from the point-of-view of the jobs and does not model the actual time spent on each job by the processors, instead it is defined as the amount of time from the moment the job enters the network, until the moment it is completed. Hence, the processing time can be computed as soon as the job is assigned to the processors and is equal to the length of the longest queue.

Remark 3. In the example of Figure 6, note that the costs, i.e. **tick**(l_1) and **tick**(l_2), depend on the length of the queues and are therefore unbounded. However, all updates in program variables are bounded, i.e. a queue size is increased by at most 3 at each step of the program. The maximal update appears in the assignment $l_1 := l_1 + 3$.

3.4 Stochastic Linear Recurrences

Linear recurrences are systems that consist of a finite set $\mathbf{x}$ of variables, together with a finite set $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ of linear update rules. At each step of the system's execution, one of the rules is chosen and applied to the variables. Formally, if there are n variables, then we consider $\mathbf{x}$ and each of the $\mathbf{a}_i$'s to be a vector of length n, and applying the rule $\mathbf{a}_i$ corresponds to the assignment $\mathbf{x} := \mathbf{a}_i \cdot \mathbf{x}$. This process continues as long as a condition ϕ is satisfied. Linear recurrences are well-studied and appear in many contexts, e.g. to model linear dynamical systems, in theoretical biology, and in statistical physics (see [7, 76, 77]). A classical example is the so-called species fight in ecology.

A natural extension of linear recurrences is to consider stochastic linear recurrences, where at each step the rule to be applied is chosen probabilistically. Moreover, the cost of the process at each step is a linear combination $\mathbf{c} \cdot \mathbf{x}$ of the variables. Hence, a general stochastic linear recurrence is a program in the form shown in Figure 7.

We present a concrete instantiation of such a program in the context of species fight. Consider a fight between two types of species, a and b, where there are a finite number of each type in the initial population. The types compete and might also prey upon each other. The fitness of the types depends on the environment, which evolves stochastically. For example, the environment may represent the temperature, and a type might have an advantage over the other type in warm/cold environment. The cost we model is the amount of resources consumed by the population. Hence, it is a linear combination of the population of each type (i.e. each individual consumes some resources at each time step).

Figure 8 provides an explicit example, in which with probability $1/2$, the environment becomes hospitable to a, which leads to an increase in its population, and assuming that a preys on b, this leads to a decrease in the population of b. On the other hand, the environment might become hostile to a, which leads to an increase in b's population. Moreover, each individual of either type a or b consumes 1 unit of resource per time unit. We also assume that a population of less than 5 is unsustainable and leads to extinction.

Remark 4. Note that in Figure 8, there are unbounded costs (**tick**($a+b$)) and unbounded updates to the variables (e.g. $a := 1.1 * a$). However, the costs are always nonnegative.

4 Main Ideas and Novelty

In this work, our main contribution is an automated approach for obtaining polynomial bounds on the expected accumulated cost of nondeterministic probabilistic programs. In this section, we present an outline of our main ideas, and a discussion on their novelty in comparison with previous approaches. The key contributions are organized as follows:

```
while φ do
    if prob(p_1) then
        x := a_1 · x
    else if prob(p_2) then
        x := a_2 · x
            ⋮
    else if prob(p_m) then
        x := a_m · x
    fi ... fi;
    tick(c · x)
od
```

Figure 7. A general stochastic linear recurrence

```
while a ≥ 5 and b ≥ 5 do
    tick(a + b);
    if prob(0.5) then b := 0.9 ∗ b; a := 1.1 ∗ a
    else b := 1.1 ∗ b; a := 0.9 ∗ a fi
od
```

Figure 8. A species fight example

(a) mathematical foundations; (b) soundness of the approach; and (c) computational results.

4.1 Mathematical Foundations

Martingale-based approach. The previous approach of [74] can only handle nonnegative bounded costs. Their main technique is to consider *potential functions* and probabilistic extensions of weakest precondition, which relies on monotonicity. This is the key reason why the costs must be nonnegative. Instead, our approach is based on martingales, and can hence handle both positive and negative costs.

Extension of OST. A standard mathematical result for the analysis of martingales is the Optional Stopping Theorem (OST). The OST provides a set of conditions on a (super)martingale $\{X_n\}_{n=0}^{\infty}$ that are sufficient to ensure bounds on its expected value at a stopping time. One of the requirements of the OST is the so-called bounded difference condition, i.e. that there should exist a constant number c, such that the stepwise difference $|X_{n+1} - X_n|$ is always less than c. In program cost analysis, this condition translates to the requirement that the stepwise cost function at each program point must be bounded by a constant. Unfortunately, it is well-known that the bounded difference condition in OST is an essential prerequisite, and thus application of classical OST can only handle programs with bounded costs.

We present an extension of the OST that provides certain new conditions for handling differences $|X_{n+1} - X_n|$ that are not bounded by a constant, but instead by a polynomial on the step number n. Hence, our extended OST can be applied to programs such as the motivating examples in Sections 3.1, 3.2 and 3.3. The details of the OST extension are presented in Section 5.

4.2 Soundness of the Approach

For a sound approach to compute polynomial bounds on expected accumulated cost, we present the following results (details in Section 6):

1. We define the notions of *polynomial upper cost supermartingale (PUCS)* and *polynomial lower cost submartingale (PLCS)* for upper and lower bounds of the expected accumulated cost over probabilistic programs, respectively (see Section 6.1).
2. For the case where the costs can be both positive and negative (bounded or unbounded), but the variable updates are bounded, we use our extended OST to establish that PUCS's and PLCS's provide a sound approach to obtain upper and lower bounds on the expected accumulated cost (see Section 6.2).
3. For costs that are nonnegative (even with unbounded updates), we show that PUCS's provide a sound approach to obtain upper bounds on the expected accumulated cost (see Section 6.3). The key mathematical result we use here is the Monotone Convergence theorem. We do not need OST in this case.

4.3 Computational Results

By our definition of PUCS/PLCS, a candidate polynomial h is a PUCS/PLCS for a given program, if it satisfies a number of polynomial inequalities, which can be obtained from the CFG of the program. Hence, we reduce the problem of synthesis of a PUCS/PLCS to solving a system of polynomial inequalities. Such systems can be solved using quantifier elimination, which is computationally expensive. Instead, we present the alternative sound method of using a Positivstellensatz, i.e. a theorem in real semi-algebraic geometry that characterizes positive polynomials over a semi-algebraic set. In particular, we use Handelman's Theorem to show that given a nondeterministic probabilistic program, a PUCS/PLCS can be synthesized by solving a linear programming instance of

polynomial size (wrt the size of the input program and invariant). Hence, our sound approach for obtaining polynomial bounds on the expected accumulated cost of a program runs in polynomial time. The details are presented in Section 7.

4.4 Novelty

The main novelties of our approach are as follows:

1. In contrast to previous approaches (such as [74]) that can only handle bounded positive costs (due to monotonicity requirements), our approach can handle both positive and negative costs, as well as unbounded costs. In particular, unlike previous approaches, our approach can handle the motivating examples of Section 3. Moreover, our approach presents a novel extension of classical results for martingales.
2. While the previous approach of [74] could only present sound upper bounds with positive bounded costs, our approach for positive and negative costs, with the restriction of bounded updates to the variables, can provide both upper and lower bounds on the expected accumulated costs. Thus, for the examples of Sections 3.1, 3.2 and 3.3, we obtain both upper and lower bounds.
3. We present an *efficient* computational approach for obtaining bounds on the expected accumulated costs. Our algorithm has provable polynomial runtime guarantee. Previous approach of [74] presents compositional inference rules and does not provide any polynomial runtime guarantee for the computation.

4.5 Limitations

We now discuss some limitations of our approach.

1. As in previous approaches, such as [20, 74], we need to assume that the input program terminates.
2. For programs with both positive and negative costs, we handle either bounded updates to variables or bounded costs. The most general case, with both unbounded costs and unbounded updates, remains open.
3. For unbounded updates to variables, we consider non-negative costs, and present only upper bounds, and not lower bounds. However, note that our approach is the first one to present any lower bounds for cost analysis of probabilistic programs (with bounded updates to variables), and no previous approach can obtain lower bounds in any case.
4. While the previous approach of [74] presents compositional inference proof rules, our approach cannot obtain such compositional rules. However, the efficiency of our approach comes from the fact that our algorithm is provably polynomial-time and relies on efficient linear-programming solvers.

5 The Extension of the OST

The Optional Stopping Theorem (OST) states that, given a martingale (resp. supermartingale), if its step-wise difference $X_n - X_{n+1}$ is bounded, then its expected value at a stopping time is equal to (resp. no greater than) its initial value.

Theorem 5.1 (Optional Stopping Theorem (OST) [36, 96])**.** *Consider any stopping time U wrt a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and any martingale (resp. supermartingale) $\{X_n\}_{n=0}^{\infty}$ adapted to $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and let $Y = X_U$. Then the following condition is sufficient to ensure that $\mathbb{E}(|Y|) < \infty$ and $\mathbb{E}(Y) = \mathbb{E}(X_0)$ (resp. $\mathbb{E}(Y) \leq \mathbb{E}(X_0)$):*

- *There exists an $M \in [0, \infty)$ such that for all $n \geq 0$, $|X_{n+1} - X_n| \leq M$ almost surely.*

It is well-known that the stepwise bounded difference condition (i.e. $|X_{n+1} - X_n| \leq M$) is an essential prerequisite [96]. Below we present our extension of OST to unbounded differences.

Theorem 5.2 (The Extended OST)**.** *Consider any stopping time U wrt a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and any martingale (resp. supermartingale) $\{X_n\}_{n=0}^{\infty}$ adapted to $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and let $Y = X_U$. Then the following condition is sufficient to ensure that $\mathbb{E}(|Y|) < \infty$ and $\mathbb{E}(Y) = \mathbb{E}(X_0)$ (resp. $\mathbb{E}(Y) \leq \mathbb{E}(X_0)$):*

- *There exist real numbers $M, c_1, c_2, d > 0$ such that (i) for sufficiently large $n \in \mathbb{N}$, it holds that $\mathbb{P}(U > n) \leq c_1 \cdot e^{-c_2 \cdot n}$ and (ii) for all $n \in \mathbb{N}$, $|X_{n+1} - X_n| \leq M \cdot n^d$ almost surely.*

Intuition and proof idea. We extend the OST so that the stepwise difference $|X_{n+1} - X_n|$ need not be bounded by a constant, but instead by a polynomial in terms of the step counter n. However, we require that the stopping time U satisfies the concentration condition that specifies an exponential decrease in $\mathbb{P}(U > n)$. We present a rigorous proof that uses Monotone and Dominated Convergence Theorems along with the concentration bounds and polynomial differences to establish the above result. For technical details see Appendix D.1.

6 Polynomial Cost Martingales

In this section, we introduce the notion of polynomial cost martingales, which serve as the main tool for reducing the

cost analysis problem over nondeterministic probabilistic programs to the analysis of a stochastic process.

6.1 Definitions

Below, we fix a probabilistic program and its CFG of form (1). In order to apply our extended OST for cost analysis of the program, it should first be translated into a discrete-time stochastic process. This is achieved using the concept of *cost martingales*. To define cost martingales, we first need the notions of invariants and pre-expectation.

Definition 6.1 (Invariants and linear invariants)**.** Given a program, its set L of labels, and an initial valuation $\mathbf{v}^*$ to program variables V_p, an *invariant* is a function $I : L \to P(Val_{V_\mathrm{p}})$ that assigns a set $I(\ell)$ of valuations over V_p to every label ℓ, such that for all configurations $(\ell, \mathbf{v})$ that are reachable from the initial configuration $(\ell_\mathrm{in}, \mathbf{v}^*)$ by a run of the program, it holds that $\mathbf{v} \in I(\ell)$. The invariant I is called linear if every $I(\ell)$ is a finite union of polyhedra.

Intuition. An invariant I is an over-approximation of the reachable valuations at each label of the program. The invariant I is linear if it can be represented by linear inequalities.

Example 6.2. Figure 9 (top), shows the same program as in Example 2.1, together with linear invariants for each label of the program. The invariants are enclosed in square brackets.

Definition 6.3 (Pre-expectation)**.** Consider any function $h : L \times Val_{V_\mathrm{p}} \to \mathbb{R}$. We define its *pre-expectation* as the function $\mathrm{pre}_h : L \times Val_{V_\mathrm{p}} \to \mathbb{R}$ by:

- $\mathrm{pre}_h(\ell, \mathbf{v}) := h(\ell, \mathbf{v})$ if $\ell = \ell_\mathrm{out}$ is the terminal label;
- $\mathrm{pre}_h(\ell, \mathbf{v}) := \mathbb{E}_\mathbf{u}[h(\ell', F_\ell(\mathbf{v}, \mathbf{u}))]$ if $\ell \in L_\mathrm{a}$ is an assignment label with the update function F_ℓ, and the next label is ℓ'. Note that in the expectation $\mathbb{E}_\mathbf{u}[h(\ell', F_\ell(\mathbf{v}, \mathbf{u}))]$, the values of ℓ' and $\mathbf{v}$ are treated as constants and $\mathbf{u}$ observes the probability distributions specified for the sampling variables;
- $\mathrm{pre}_h(\ell, \mathbf{v}) := \mathbf{1}_{\mathbf{v} \models \phi} \cdot h(\ell_1, \mathbf{v}) + \mathbf{1}_{\mathbf{v} \not\models \phi} \cdot h(\ell_2, \mathbf{v})$ if $\ell \in L_\mathrm{b}$ is a branching label and ℓ_1, ℓ_2 are the labels for the **true**-branch and the **false**-branch, respectively. The indicator $\mathbf{1}_{\mathbf{v} \models \phi}$ is equal to 1 when $\mathbf{v}$ satisfies ϕ and 0 otherwise. Conversely, $\mathbf{1}_{\mathbf{v} \not\models \phi}$ is 1 when $\mathbf{v}$ does not satisfy ϕ and 0 when it does;
- $\mathrm{pre}_h(\ell, \mathbf{v}) := \sum_{(\ell, p, \ell') \in \to} p \cdot h(\ell', \mathbf{v})$ if $\ell \in L_\mathrm{p}$ is a probabilistic label;
- $\mathrm{pre}_h(\ell, \mathbf{v}) := R_\ell(\mathbf{v}) + h(\ell', \mathbf{v})$ if $\ell \in L_\mathrm{t}$ is a tick label with the cost function R_ℓ and the successor label ℓ';

```
1 : [x ≥ 0]                    while x ≥ 1 do
2 : [x ≥ 1]                      x := x + r ;
3 : [x ≥ 0]                      y := r' ;
4 : [x ≥ 0 ∧ −1 ≤ y ≤ 1]       tick(x ∗ y)  od
5 : [0 ≤ x ≤ 1]
```

n	$h(\ell_n, x, y)$	$pre_h(\ell_n, x, y)$
1	$\frac{1}{3}x^2 + \frac{1}{3}x$	$1_{x\geq 1} \cdot h(\ell_2, x, y) + 1_{x<1} \cdot h(\ell_5, x, y) =$ $1_{x\geq 1} \cdot (\frac{1}{3}x^2 + \frac{1}{3}x) + 1_{x<1} \cdot 0$
2	$\frac{1}{3}x^2 + \frac{1}{3}x$	$\frac{1}{4}h(\ell_3, x+1, y) + \frac{3}{4}h(\ell_3, x-1, y) =$ $\frac{1}{3}x^2 + \frac{1}{3}x$
3	$\frac{1}{3}x^2 + \frac{2}{3}x$	$\frac{2}{3}h(\ell_4, x, 1) + \frac{1}{3}h(\ell_4, x, -1) =$ $\frac{1}{3}x^2 + \frac{2}{3}x$
4	$\frac{1}{3}x^2 + xy + \frac{1}{3}x$	$x \cdot y + h(\ell_1, x, y) =$ $\frac{1}{3}x^2 + xy + \frac{1}{3}x$
5	0	$h(\ell_5, x, y) = 0$

Figure 9. A program together with an example function h and the corresponding pre-expectation function pre_h.

- $\mathrm{pre}_h(\ell, \mathbf{v}) := \max_{(\ell, \star, \ell') \in \to} h(\ell', \mathbf{v})$ if $\ell \in L_\mathrm{nd}$ is a nondeterministic label.

Intuition. The pre-expectation $\mathrm{pre}_h(\ell, \mathbf{v})$ is the cost of the current step plus the expected value of h in the next step of the program execution, i.e. the step after the configuration $(\ell, \mathbf{v})$. In this expectation, ℓ and $\mathbf{v}$ are treated as constants. For example, the pre-expectation at a probabilistic branching label is the averaged sum over the values of h at all possible successor labels.

Example 6.4. In Figure 9 (top) we consider the same program as in Example 2.1. Recall that the probability distributions used for sampling variables r and r' are $(1, -1) : (1/4, 3/4)$ and $(1, -1) : (2/3, 1/3)$, respectively. The table in Figure 9 (bottom) provides an example function h and the corresponding pre-expectation pre_h. The gray part shows the steps in computing the function pre_h and the black part is the final result[2].

We now define the central notion of *cost martingales*. For algorithmic purposes, we only consider *polynomial* cost martingales in this work. We start with the notion of PUCS which is meant to serve as an upperbound for the expected accumulated cost of a program.

Definition 6.5 (Polynomial Upper Cost Supermartingales)**.** A *polynomial upper cost supermartingale (PUCS)* of degree d

[2]The reason for choosing this particular h will be clarified by Example 6.6.

wrt a given linear invariant I is a function $h : L \times Val_{V_p} \to \mathbb{R}$ that satisfies the following conditions:

(C1) for each label ℓ, $h(\ell)$ is a polynomial of degree at most d over program variables;

(C2) for all valuations $\mathbf{v} \in Val_{V_p}$, we have $h(\ell_{out}, \mathbf{v}) = 0$;

(C3) for all non-terminal labels $\ell \in L \setminus \{\ell_{out}\}$ and reachable valuations $\mathbf{v} \in I(\ell)$, we have $\mathrm{pre}_h(\ell, \mathbf{v}) \leq h(\ell, \mathbf{v})$.

Intuition. Informally, (C1) specifies that the PUCS should be polynomial at each label, (C2) says that the value of the PUCS at the terminal label ℓ_{out} should always be zero, and (C3) specifies that at all reachable configurations $(\ell, \mathbf{v})$, the pre-expectation is no more than the value of the PUCS itself.

Note that if h is polynomial in program variables, then $\mathrm{pre}_h(\ell, -)$ is also polynomial if ℓ is an assignment, probabilistic branching or tick label. For example, in the case of assignment labels, $\mathbb{E}_{\mathbf{u}}[h(\ell', F_\ell(\mathbf{v}, \mathbf{u}))]$ is polynomial in $\mathbf{v}$ if both h and F_ℓ are polynomial.

Example 6.6. By Definition 6.5, the function h given in Example 6.4 is a PUCS. For every label ℓ of the program, $h(\ell, -)$ is a polynomial of degree at most 2, so h satisfies condition (C1). It is straightforward to verify, using the table in Figure 9 (bottom), that h satisfies (C2) and (C3) as well.

We now define the counterpart of PUCS for lower bound.

Definition 6.7 (Polynomial Lower Cost Submartingales)**.** A *polynomial lower cost submartingale (PLCS)* wrt a linear invariant I is a function $h : L \times Val_{V_p} \to \mathbb{R}$ that satisfies (C1) and (C2) above, and the additional condition (C3') below (instead of (C3)):

(C3') for all non-terminal labels $\ell \neq \ell_{out}$ and reachable valuations $\mathbf{v} \in I(\ell)$, we have $\mathrm{pre}_h(\ell, \mathbf{v}) \geq h(\ell, \mathbf{v})$;

Intuitively, a PUCS requires the pre-expectation pre_h to be no more than h itself, while a PLCS requires the converse, i.e. that pre_h should be no less than h.

Example 6.8. As shown in Example 6.6, the function h given in Example 6.4 (Figure 9) satisfies (C1) and (C2). Using the table in Figure 9, one can verify that h satisfies (C3') as well. Hence, h is a PLCS.

In the following sections, we prove that PUCS's and PLCS's are sound methods for obtaining upper and lower bounds on the expected accumulated cost of a program.

6.2 General Unbounded Costs and Bounded Updates

In this section, we consider nondeterministic probabilistic programs with general unbounded costs, i.e. both positive and negative costs, and bounded updates to the program variables. Using our extension of the OST (Theorem 5.2), we show that PUCS's and PLCS's are sound for deriving upper and lower bounds for the expected accumulated cost.

Recall that the extended OST has two prerequisites. One is that, for sufficiently large n, the stopping time U should have exponentially decreasing probability of nontermination, i.e. $\mathbb{P}(U > n) \leq c_1 \cdot e^{-c_2 \cdot n}$. The other is that the stepwise difference $|X_{n+1} - X_n|$ should be bounded by a polynomial on the number n of steps. We first describe how these conditions affect the type of programs that can be considered, and then provide our formal soundness theorems.

The first prerequisite is equivalent to the assumption that the program has the concentration property. To ensure the first prerequisite, we apply the existing approach of difference-bounded ranking-supermartingale maps [18, 22]. We ensure the second prerequisite by assuming the bounded update condition, i.e. that every assignment to each program variable changes the value of the variable by a bounded amount. We first formalize the concept of bounded update and then argue why it is sufficient to ensure the second prerequisite.

Definition 6.9 (Bounded Update)**.** A program P with invariant I has the *bounded update* property over its program variables, if there exists a constant $M > 0$ such that for every assignment label ℓ with update function F_ℓ, we have $\forall \mathbf{v} \in I(\ell)\ \forall \mathbf{u}\ \forall x \in V_p\ |F_\ell(\mathbf{v}, \mathbf{u})(x) - \mathbf{v}(x)| \leq M$.

The reason for assuming bounded update. A consequence of the bounded update condition is that at the n-th execution step of any run of the program, the absolute value of any program variable x is bounded by $M \cdot n + x_0$, where M is the constant bound in the definition above and x_0 is the initial value of the variable x. Hence, for large enough n, the absolute value of any variable x is bounded by $(M + 1) \cdot n$. Therefore, given a PUCS h of degree d, one can verify that the step-wise difference of h is bounded by a polynomial on the number n of steps. More concretely, h is a degree-d polynomial over variables that are bounded by $(M + 1) \cdot n$, so h is bounded by $M' \cdot n^d$ for some constant $M' > 0$. Thus, the bounded update condition is sufficient to fulfill the second prerequisite of our extended OST.

Based on the discussion above, we have the following soundness theorems:

Theorem 6.10 (Soundness of PUCS)**.** *Consider a nondeterministic probabilistic program P, with a linear invariant I and a PUCS h. If P satisfies the concentration property and the*

bounded update property, then $\mathit{supval}(\mathbf{v}) \leq h(\ell_{\text{in}}, \mathbf{v})$ *for all initial valuations* $\mathbf{v} \in I(\ell_{\text{in}})$.

Proof Sketch. We define the stochastic process $\{X_n\}_{n=0}^{\infty}$ as $X_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n)$, where $\overline{\ell}_n$ is the random variable representing the label at the n-th step of a program run, and $\overline{\mathbf{v}}_n$ is a vector of random variables consisting of components $\overline{\mathbf{v}}_n(x)$ which represent values of program variables x at the n-th step. Furthermore, we construct the stochastic process $\{Y_n\}_{n=0}^{\infty}$ such that $Y_n = X_n + \sum_{k=0}^{n-1} C_k$. Recall that C_k is the cost of the k-th step of the run and $C_\infty = \sum_{k=0}^{\infty} C_k$. We consider the termination time T of P and prove that $\{Y_n\}_{n=0}^{\infty}$ satisfies the prerequisites of our extended OST (Theorem 5.2). This proof depends on the assumption that P has concentration and bounded update properties. Then by applying Theorem 5.2, we have that $\mathbb{E}(Y_T) \leq \mathbb{E}(Y_0)$. Since $Y_T = X_T + \sum_{k=0}^{T} C_k = C_\infty$, we obtain the desired result. For a more detailed proof, see Appendix D.3. □

Example 6.11. Given that the h in Example 6.4 is a PUCS, we can conclude that for all initial values x_0 and y_0, we have $\mathit{supval}(x_0, y_0) \leq h(\ell_1, x_0, y_0) = \frac{1}{3}x_0^2 + \frac{1}{3}x_0$.

We showed that PUCS's are sound upper bounds for the expected accumulated cost of a program. The following theorem provides a similar result for PLCS's and lower bounds.

Theorem 6.12 (Soundness of PLCS)**.** *Consider a nondeterministic probabilistic program P, with a linear invariant I and a PLCS h. If P satisfies the concentration property and the bounded update property, then* $\mathit{supval}(\mathbf{v}) \geq h(\ell_{\text{in}}, \mathbf{v})$ *for all initial valuations* $\mathbf{v} \in I(\ell_{\text{in}})$.

The proof is similar to that of Theorem 6.10 and is relegated to Appendix D.4.

Example 6.13. Given that the h in Example 6.4 is a PLCS, we can conclude that for all initial values x_0 and y_0, we have $\mathit{supval}(x_0, y_0) \geq h(\ell_1, x_0, y_0) = \frac{1}{3}x_0^2 + \frac{1}{3}x_0$.

Remark 5. Putting together the results from Examples 6.11 and 6.13, we conclude that the expected accumulated cost of Example 6.4 is precisely $\frac{1}{3}x_0^2 + \frac{1}{3}x_0$.

Remark 6. Note that the motivating examples in Sections 3.1, 3.2 and 3.3, i.e. Bitcoin mining, Bitcoin pool mining and FJ queuing networks, have potentially unbounded costs that can be both positive and negative. Moreover, they satisfy the bounded update property. Therefore, using PUCS's and PLCS's leads to sound bounds on the expected accumulated costs of these programs.

6.3 Unbounded Nonnegative Costs and General Updates

In this section, we consider programs with unbounded *nonnegative* costs, and show that a PUCS is a sound upper bound for their expected accumulated cost. This result holds for programs with arbitrary unbounded updates to the variables.

Our main tool is the well-known Monotone Convergence Theorem (MCT) [96], which states that if X is a random variable and $\{X_n\}_{n=0}^{\infty}$ is a non-decreasing discrete-time stochastic process such that $\lim_{n\to\infty} X_n = X$ almost surely, then $\lim_{n\to\infty} \mathbb{E}(X_n) = \mathbb{E}(X)$.

As in the previous case, the first step is to translate the program to a stochastic process. However, in contrast with the previous case, in this case we only consider *nonnegative* PUCS's. This is because all costs are assumed to be nonnegative. We present the following soundness result:

Theorem 6.14 (Soundness of nonnegative PUCS)**.** *Consider a nondeterministic probabilistic program P, with a linear invariant I and a nonnegative PUCS h. If all the step-wise costs in P are always nonnegative, then* $\mathit{supval}(\mathbf{v}) \leq h(\ell_{\text{in}}, \mathbf{v})$ *for all initial valuations* $\mathbf{v} \in I(\ell_{\text{in}})$.

Proof Sketch. We define the stochastic process $\{X_n\}_{n=0}^{\infty}$ as in Theorem 6.12, i.e. $X_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n)$. By definition, for all n, we have $\mathbb{E}(X_{n+1}) + \mathbb{E}(C_n) \leq \mathbb{E}(X_n)$, hence by induction, we get $\mathbb{E}(X_{n+1}) + \sum_{m=0}^{n} \mathbb{E}(C_m) \leq \mathbb{E}(X_0)$. Given that h is nonnegative, $\mathbb{E}(X_{n+1}) \geq 0$, so $\sum_{m=0}^{n} \mathbb{E}(C_m) \leq \mathbb{E}(X_0)$. By applying the MCT, we obtain $\mathbb{E}(C_\infty) = \mathbb{E}(\lim_{n\to\infty} \sum_{m=0}^{n} C_m) = \lim_{n\to\infty} \sum_{m=0}^{n} \mathbb{E}(C_m) \leq \mathbb{E}(X_0)$, which is the desired result. For a more detailed proof, see Appendix D.5. □

Remark 7. Note that the motivating example in Section 3.4, i.e. the species fight stochastic linear recurrence, has unbounded nonnegative costs. Therefore, nonnegative PUCS's lead to sound upper bounds on the expected accumulated cost of this program.

7 Algorithmic Approach

In the previous section, we showed that in order to derive bounds for the expected accumulated cost of a program, it suffices to synthesize a PUCS/PLCS. In this section, we provide automated algorithms that, given a program P, an initial valuation $\mathbf{v}^*$, a linear invariant I and a constant d, synthesize a PUCS/PLCS of degree d. For brevity, we only describe our algorithm for PUCS synthesis. A PLCS can be synthesized in the same manner. Our algorithms run in polynomial time

and reduce the problem of PUCS/PLCS synthesis to a linear programming instance by applying Handelman's theorem.

In order to present Handelman's theorem, we need a few basic definitions. Let X be a finite set of variables and $\Gamma \subseteq \mathbb{R}[X]$ a finite set of linear functions (degree-1 polynomials) over X. We define $\langle\Gamma\rangle \subseteq \mathit{Val}_X$ as the set of all valuations υ to the variables in X that satisfy $g_i(\upsilon) \geq 0$ for all $g_i \in \Gamma$. We also define the monoid set of Γ as

$$\mathit{Monoid}(\Gamma) := \left\{ \prod_{i=1}^{t} g_i \mid t \in \mathbb{N} \cup \{0\} \ \wedge \ g_1, \ldots, g_t \in \Gamma \right\}.$$

By definition, it is obvious that if $g \in \mathit{Monoid}(\Gamma)$, then for every $\upsilon \in \langle\Gamma\rangle$, we have $g(\upsilon) \geq 0$. Handelman's theorem characterizes every polynomial g that is positive over $\langle\Gamma\rangle$.

Theorem 7.1 (Handelman's Theorem [50]). *Let $g \in \mathbb{R}[X]$ be a polynomial such that $g(\mathbf{x}) > 0$ for all $\mathbf{x} \in \langle\Gamma\rangle$. If $\langle\Gamma\rangle$ is compact, then*

$$g = \sum_{k=1}^{s} c_k \cdot f_k \qquad (\dagger)$$

for some $s \in \mathbb{N}$, $c_1, \ldots, c_s > 0$ and $f_1, \ldots, f_s \in \mathit{Monoid}(\Gamma)$.

Intuitively, Handelman's theorem asserts that every polynomial g that is positive over $\langle\Gamma\rangle$ must be a positive linear combination of polynomials in $\mathit{Monoid}(\Gamma)$. This means that in order to synthesize a polynomial that is positive over $\langle\Gamma\rangle$ we can limit our attention to polynomials of the form (†). When using Handelman's theorem in our algorithm, we fix a constant K and only consider those elements of $\mathit{Monoid}(\Gamma)$ that are obtained by K multiplicands or less.

We now have all the required tools to describe our algorithm for synthesizing a PUCS.

PUCS Synthesis Algorithm. The algorithm has four steps:

(1) *Creating a Template for h.* Let $X = V_{\mathrm{p}}$ be the set of program variables. According to (C1), we aim to synthesize a PUCS h, such that for each label ℓ_i of the program, $h(\ell_i)$ is a polynomial of degree at most d over X. Let $M_d(X) = \{\bar{f}_1, \bar{f}_2, \ldots, \bar{f}_r\}$ be the set of all monomials of degree at most d over the variables X. Then, $h(\ell_i)$ has to be of the form $\sum_{j=1}^{r} a_{ij} \cdot \bar{f}_j$ for some unknown real values a_{ij}. We call this expression a template for $h(\ell_i)$. Note that by condition (C2) the template for $h(\ell_{\mathrm{out}})$ is simply $h(\ell_{\mathrm{out}}) = 0$. The algorithm computes these templates at every label ℓ_i, treating the a_{ij}'s as unknown variables.

(2) *Computing Pre-expectation.* The algorithm symbolically computes a template for pre_h using Definition 6.3 and the template obtained for h in step (1). This template will also contain a_{ij}'s as unknown variables.

(3) *Pattern Extraction.* The algorithm then processes condition (C3) by symbolically computing polynomials $g = h(\ell_i) - \mathrm{pre}_h(\ell_i)$ for every label ℓ_i. Then, as in Handelman's theorem, it rewrites each g on the left-hand-side of the equations above in the form (†), using the linear invariant $I(\ell_i)$ as the set Γ of linear functions. The nonnegativity of h is handled in a similar way. This effectively translates (C3) and the nonnegativity into a system S of linear equalities over the a_{ij}'s and the new nonnegative unknown variables c_k resulting from equation (†).

(4) *Solution via Linear Programming.* The algorithm calls an LP-solver to find a solution of S that optimizes $h(\ell_{\mathrm{in}}, \mathbf{v}^*)$.

If the algorithm is successful, i.e. if the obtained system of linear equalities is feasible, then the solution to the LP contains values for the unknowns a_{ij} and hence, we get the coefficients of the PUCS h. Note that we are optimizing for $h(\ell_{\mathrm{in}}, \mathbf{v}^*)$, so the obtained PUCS is the one that produces the best polynomial upper bound for the expected accumulated cost of P with initial valuation $\mathbf{v}^*$. We use the same algorithm for PLCS synthesis, except that we replace (C3) with (C3').

Theorem 7.2. *The algorithm above has polynomial runtime and synthesizes sound upper and lower bounds for the expected accumulated cost of the given program P.*

Proof. Step (1) ensures that (C1), (C2) are satisfied, while step (3) forces the polynomials h and g's to be nonnegative, ensuring nonnegativity and (C3). So the synthesized h is a PUCS. Steps (1)–(3) are polynomial-time symbolic computations. Step (4) solves an LP of polynomial size. Hence, the runtime is polynomial wrt the length of the program. The reasoning for PLCS synthesis is similar. □

Example 7.3. Consider the program in Figure 9 (Page 10). Suppose that the initial valuation is $x_0 = 100, y_0 = 0$ and that we are looking for a quadratic PUCS, i.e. $d = 2$. Our algorithm proceeds as follows:

(1) A quadratic template is created for h, by setting $h(\ell_n, x, y) := a_{n1} \cdot x^2 + a_{n2} \cdot xy + a_{n3} \cdot x + a_{n4} \cdot y^2 + a_{n5} \cdot y + a_{n6}$. This template contains all monomials of degree 2 or less.

(2) A template for the function pre_h is computed in the same manner as in Example 6.4, except for that the computation is now symbolic and contains the unknown variables a_{ij}. The resulting template is presented in Table 1.

(3) For each label ℓ_i, the algorithm symbolically computes $g = h(\ell_i) - \mathrm{pre}_h(\ell_i)$. For example, for ℓ_3, the algorithm computes $g(x, y) = h(\ell_3, x, y) - \mathrm{pre}_h(\ell_3, x, y) = (a_{31} -$

n	$pre_h(\ell_n, x, y)$
1	$\mathbf{1}_{x\geq 1}\cdot(a_{21}\cdot x^2+a_{22}\cdot xy+a_{23}\cdot x+a_{24}\cdot y^2+a_{25}\cdot y+a_{26})+\mathbf{1}_{x<1}\cdot 0$
2	$a_{31}\cdot x^2+a_{32}\cdot xy+(a_{33}-a_{31})\cdot x+a_{34}\cdot y^2+(a_{35}-\frac{1}{2}a_{32})\cdot y+a_{31}-\frac{1}{2}a_{33}+a_{36}$
3	$a_{41}\cdot x^2+(\frac{1}{3}a_{42}+c_{43})\cdot x+a_{44}+\frac{1}{3}a_{45}+a_{46}$
4	$a_{11}\cdot x^2+(a_{12}+1)\cdot xy+a_{13}\cdot x+a_{14}\cdot y^2+a_{15}\cdot y+a_{16}$
5	0

Table 1. Template for pre_h of the program in Figure 9

$a_{41})\cdot x^2+a_{32}\cdot xy+(a_{33}-\frac{1}{3}a_{42}-a_{43})\cdot x+a_{34}\cdot y^2+a_{35}\cdot y+a_{36}-a_{44}-\frac{1}{3}a_{45}-a_{46}$. It then rewrites g according to (†) using $\Gamma = I(\ell_3) = \{x\}$, i.e. $g(x,y) = \sum c_k \cdot f_k(x,y)$. This is because we need to ensure $g \geq 0$ to fulfill condition (C3). This leads to the polynomial equation $\sum c_k \cdot f_k(x,y) = (a_{31}-a_{41})\cdot x^2+a_{32}\cdot xy+(a_{33}-\frac{1}{3}a_{42}-a_{43})\cdot x+a_{34}\cdot y^2+a_{35}\cdot y+a_{36}-a_{44}-\frac{1}{3}a_{45}-a_{46}$, which can in turn be translated to several linear equations in terms of the c_k's and a_{ij}'s by equating the coefficient of each term on both sides of the polynomial equation. The algorithm generates such linear equations for every label of the program.

(4) The algorithm calls an LP-solver to solve the linear programming instance consisting of all linear equations obtained in step (3). Given that we are looking for an optimal upperbound on the expected accumulated cost with the initial valuation $x_0 = 100, y_0 = 0$, the algorithm minimizes $h(\ell_1, 100, 0) = 10000\cdot a_{11} + 100\cdot a_{13} + a_{16}$ subject to these linear equations.

In this case, the resulting values for a_{ij}'s lead to the same PUCS h as in Figure 9. So the upper bound on the expected accumulated cost is $\frac{1}{3}x_0^2 + \frac{1}{3}x_0 = 3366.\overline{6}$. The algorithm can similarly synthesize a PLCS. In this case, the same function h is reported as a PLCS. Therefore, the exact expected accumulated cost of this program is $3366.\overline{6}$ and our algorithm is able to compute it precisely. See Appendix E for more details on this example.

8 Experimental Results

In this section, we report an implementation of our approach and present experimental results on a variety of programs. We show that our approach is able to obtain bounds on the expected accumulated costs of the motivating examples presented in Section 3 that no previous approach could handle. A key feature of our algorithms is that they are very efficient and only rely on standard tools, such as invariant generators and LP-solvers.

Implementation and Environment. We implemented our approach in Matlab R2018b. We use the Stanford Invariant Generator [81] to find linear invariants and the tool in [18] to ensure the concentrated termination property for the input programs. The results were obtained on a Windows machine with an Intel Core i7 3.6GHz processor and 8GB of RAM.

Experimental Results. Table 2 provides a summary of our experimental results over ten benchmark programs. These include the four motivating examples of Section 3, our running example, and five other classical programs (See Appendix F for details of these programs). Each program is analyzed with three different initial valuations $\mathbf{v}_0$. In each case, we report the upper bound obtained through PUCS, the runtime of our PUCS synthesis algorithm, the lowerbound obtained through PLCS, and the runtime of our PLCS synthesis algorithm. Moreover, we simulated 1000 runs of each program with each initial value, computed the resulting costs, and reported the mean μ and standard deviation σ of the costs. Note that we do not have simulation results for bitcoin mining examples as they involve nondeterminism. Also we do not have lower bounds for species fight example as its update is unbounded.

Discussion. In all cases of Table 2, the obtained lower and upperbounds are very close, and in many cases they meet. Hence, our approach can obtain tight bounds on the expected accumulated cost of a variety of programs that could not be handled by any previous approach. Moreover, as evidenced by the reported runtimes, our algorithm is very efficient. Finally, the simulated mean costs are consistent with our bounds and can be considered as further evidence for their correctness. We put the detailed illustration of the experimental results in Appendix F.

9 Related Work

We discuss several categories of previous related works.

Termination and cost analysis. Termination of programs and other temporal properties have been studied extensively [11, 28–30, 33, 68–70, 93]. Automated amortized cost analysis has also been widely studied [5, 18, 41, 47–49, 53, 55–59, 63, 64, 85]. However, all of these approaches are for non-probabilistic programs. Other approaches for resource analysis are as follows: (a) recurrence relations for worst-case analysis [2–4, 39, 46]; (b) average case analysis through recurrence relations [21] and (c) using theorem proving [89].

Benchmark Program	$\mathbf{v_0}$	PUCS		PLCS		Simulation	
		$h(\ell_{\text{in}}, \mathbf{v}_0)$	T	$h(\ell_{\text{in}}, \mathbf{v}_0)$	T	μ	σ
Bitcoin Mining (Figure 3)	$x_0 = 20$	-28.03	4.69	-30.00	4.73	–	–
	$x_0 = 50$	-72.28	4.66	-75.00	4.63	–	–
	$x_0 = 100$	-146.03	4.62	-150.00	4.62	–	–
Bitcoin Mining Pool (Figure 4)	$y_0 = 20$	-3.73×10^3	14.03	-4.35×10^3	13.73	–	–
	$y_0 = 50$	-2.05×10^4	13.78	-2.21×10^4	13.76	–	–
	$y_0 = 100$	-7.79×10^4	13.96	-8.18×10^4	13.85	–	–
Queuing Network (Figure 6)	$n_0 = 240$	11.82	141.28	9.23	141.32	9.90	4.43
	$n_0 = 280$	13.79	142.16	10.76	140.70	11.15	4.66
	$n_0 = 320$	15.76	141.02	12.30	141.42	12.99	5.29
Species Fight (Figure 8)	$a_0 = 12, b_0 = 10$	1.65×10^3	16.43	–	–	817.40	379.28
	$a_0 = 14, b_0 = 10$	2.09×10^3	16.47	–	–	971.86	453.89
	$a_0 = 16, b_0 = 10$	2.53×10^3	16.30	–	–	1.13×10^3	0.55×10^3
Figure 2	$x_0 = 100$	3.37×10^3	3.05	3.37×10^3	3.03	3.41×10^3	0.90×10^3
	$x_0 = 160$	8.59×10^3	3.00	8.59×10^3	3.02	8.62×10^3	1.76×10^3
	$x_0 = 200$	1.34×10^4	3.00	1.34×10^4	3.00	1.35×10^4	0.25×10^4
Nested Loop	$i_0 = 50$	883.33	15.82	816.67	15.91	872.78	344.29
	$i_0 = 100$	3.43×10^3	16.13	3.30×10^3	15.89	3.43×10^3	0.90×10^3
	$i_0 = 150$	7.65×10^3	15.80	7.45×10^3	15.93	7.66×10^3	1.68×10^3
Random Walk	$x_0 = 4, n_0 = 20$	-40.00	7.00	-42.50	7.07	-42.77	23.46
	$x_0 = 8, n_0 = 20$	-30.00	6.96	-32.50	6.96	-32.32	21.27
	$x_0 = 12, n_0 = 20$	-20.00	7.09	-22.50	7.93	-23.23	18.47
2D Robot	$x_0 = 100, y_0 = 40$	8.23×10^3	20.11	8.11×10^3	20.03	7.96×10^3	5.83×10^3
	$x_0 = 100, y_0 = 60$	4.15×10^3	20.16	4.02×10^3	20.13	4.01×10^3	3.64×10^3
	$x_0 = 100, y_0 = 80$	1.45×10^3	20.15	1.32×10^3	20.13	1.36×10^3	2.00×10^3
Goods Discount	$n_0 = 100, d_0 = 1$	46.30	8.42	37.89	8.45	41.45	4.22
	$n_0 = 150, d_0 = 1$	11.63	8.43	2.56	8.43	6.33	3.84
	$n_0 = 200, d_0 = 1$	-23.02	8.46	-32.77	8.43	-28.26	3.34
Pollutant Disposal	$n_0 = 50$	2.01×10^3	10.04	1.53×10^3	9.85	1.66×10^3	1.02×10^3
	$n_0 = 80$	2.74×10^3	9.78	2.25×10^3	9.88	2.42×10^3	1.13×10^3
	$n_0 = 200$	2.04×10^3	9.75	1.56×10^3	9.78	1.66×10^3	1.56×10^3

Table 2. Experimental Results. All times are reported in seconds.

However, the recurrence relation generation is not automated, and these approaches do not consider probabilistic programs, either.

Ranking functions. Ranking functions have been widely studied for intraprocedural analysis [13, 14, 19, 27, 31, 78, 83, 88, 97]. Most works have focused on linear/polynomial ranking functions and target non-probabilistic programs [27, 31, 78, 83, 88, 97]. They have been extended in various directions, such as: symbolic approaches [15], proof rules for deterministic programs [51], sized types [25, 60, 61], and polynomial resource bounds [84]. Moreover, [47] generates bounds through abstract interpretation using inference systems. However all of these methods are also for non-probabilistic programs only.

Ranking supermartingales and qualitative analysis. Ranking functions have been extended to ranking supermartingales and studied in [1, 16–18, 22, 24, 38]. Proof rules for probabilistic programs are provided in [62, 75].

However, these works consider qualitative termination problems, i.e. whether a probabilistic program terminates almost-surely, or whether the expected termination time is bounded. They do not consider precise cost analysis, which is the focus of our work.

Cost analysis for probabilistic programs. The most closely-related work is the cost analysis for probabilistic programs considered in [74]. A detailed comparison has been already provided in Section 4. In particular, we handle positive and negative costs, as well as unbounded costs, whereas [74] can handle only positive bounded costs. Moreover, our approach is polynomial-time, whereas the approach in [74] is not proven to be polynomial-time. Another related work is [20] which considers succinct Markov decision processes (MDPs) and bounds for such MDPs. However, these MDPs are a very restricted class of programs (i.e. single while loops) and only linear bounds are obtained. Our approach considers polynomial bounds for general nondeterministic probabilistic programs.

10 Conclusion

In this work we considered the problem of cost analysis of nondeterministic probabilistic programs. While previous approaches only handled positive bounded costs, our approach can derive polynomial bounds for programs with both positive and negative costs. It is sound for general costs and bounded updates, and general updates with nonnegative costs. However, finding sound approaches that can handle general costs and general updates remains an interesting direction for future work. Moreover, while we focus on polynomial bounds, finding non-polynomial bounds (such as $O(n \log n)$ or $O(n^{1.5})$) is another interesting direction for future work.

References

[1] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *PACMPL* 2, POPL (2018), 34:1–34:32.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, Diana V. Ramírez-Deantes, Guillermo Román-Díez, and Damiano Zanardini. 2009. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.* 258, 1 (2009), 109–121.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS 2008*. 221–237.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2007. Cost Analysis of Java Bytecode. In *ESOP 2007*. 157–172.

[5] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS 2010*. 117–133.

[6] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.

[7] Shaull Almagor, Brynmor Chapman, Mehran Hosseini, Joël Ouaknine, and James Worrell. 2018. Effective Divergence Analysis for Linear Recurrence Sequences. In *CONCUR 2018*. 42:1–42:15.

[8] Serene Almomen and Daniel A Menascé. 2012. The Design of an Autonomic Controller for Self-managed Emergency Departments. In *HEALTHINF 2012*. Citeseer, 174–182.

[9] Firas Alomari and Daniel A Menasce. 2012. An autonomic framework for integrating security and quality of service support in databases. In *SERE 2012*. IEEE, 51–60.

[10] Firas Alomari and Daniel A Menasce. 2014. Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1437–1446.

[11] Rajeev Alur and Swarat Chaudhuri. 2010. Temporal Reasoning for Procedural Programs. In *VMCAI 2010*. 45–60.

[12] Arati Baliga. 2017. Understanding blockchain consensus models. *Persistent* (2017).

[13] Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *RTA*. 323–337.

[14] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV 2005*. 491–504.

[15] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50.

[16] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *CAV 2013*. 511–526.

[17] Krishnendu Chatterjee and Hongfei Fu. 2017. Termination of Nondeterministic Recursive Probabilistic Programs. *CoRR* abs/1701.02944 (2017).

[18] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV 2016*. 3–22.

[19] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017. Non-polynomial Worst-Case Analysis of Recursive Programs. In *CAV 2017*. 41–63.

[20] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Nastaran Okati. 2018. Computational Approaches for Stochastic Shortest Path on Succinct MDPs. In *IJCAI 2018*. 4700–4707.

[21] Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. 2017. Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 118–139. https://doi.org/10.1007/978-3-319-63387-9_6

[22] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *POPL 2016*. 327–342.

[23] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Yaron Velner. 2018. Ergodic Mean-Payoff Games for the Analysis of Attacks in Crypto-Currencies. In *CONCUR 2018*. 11:1–11:17.

[24] Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. 2017. Stochastic invariants for probabilistic termination. In *POPL 2017*. 145–160.

[25] Wei-Ngan Chin and Siau-Cheng Khoo. 2001. Calculating Sized Types. *Higher-Order and Symbolic Computation* 14, 2-3 (2001), 261–300.

[26] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Joint Meeting on Foundations of Software Engineering*. ACM, 92–102.

[27] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *TACAS 2001*. 67–81.

[28] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI 2006*. 415–426.

[29] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2009. Summarization for termination: no return! *Formal Methods in System Design* 35, 3 (2009), 369–387.

[30] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *TACAS 2013*. 47–61.

[31] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI 2005*. 1–24.

[32] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*. ACM, 238–252.

[33] Patrick Cousot and Radhia Cousot. 2012. An abstract interpretation framework for termination. In *POPL 2012*. 245–258.

[34] Alex de Vries. 2018. Bitcoin's Growing Energy Problem. *Joule* 2, 5 (2018), 801–805.

[35] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[36] Joseph L Doob. 1971. What is a Martingale? *The American Mathematical Monthly* 78, 5 (1971), 451–463.

[37] Javier Esparza, Andreas Gaiser, and Stefan Kiefer. 2012. Proving Termination of Probabilistic Programs Using Patterns. In *CAV 2012*. 123–138.

[38] Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *POPL 2015*. 489–501.

[39] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. 1991. Automatic Average-Case Analysis of Algorithm. *Theor. Comput. Sci.* 79, 1 (1991), 37–109.

[40] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP 2016*. Springer, 282–309.

[41] Stéphane Gimenez and Georg Moser. 2016. The complexity of interaction. In *POPL 2016*. 243–255.

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

[43] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

[44] Andrew D Gordon, Mihhail Aizatulin, Johannes Borgstrom, Guillaume Claret, Thore Graepel, Aditya V Nori, Sriram K Rajamani, and Claudio Russo. 2013. A model-learner pattern for Bayesian reasoning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 403–416.

[45] 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*. ACM, 167–181.

[46] Bernd Grobauer. 2001. Cost Recurrences for DML Programs. In *ICFP 2001*. 253–264.

[47] Bhargav S. Gulavani and Sumit Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV 2008*. 370–384.

[48] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *CAV 2009*. 51–62.

[49] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *POPL 2009*. 127–139.

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

[51] Wim H. Hesselink. 1993. Proof Rules for Recursive Procedures. *Formal Asp. Comput.* 5, 6 (1993), 554–570.

[52] Mark D Hill and Michael R Marty. 2008. Amdahl's law in the multicore era. *Computer* 41, 7 (2008).

[53] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14.

[54] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *CAV 2012*. 781–786.

[55] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *APLAS 2010*. 172–187.

[56] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *ESOP 2010*. 287–306.

[57] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *POPL 2003*. 185–197.

[58] Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *ESOP 2006*. 22–37.

[59] Martin Hofmann and Dulma Rodriguez. 2009. Efficient Type-Checking for Amortised Heap-Space Analysis. In *CSL 2009*. 317–331.

[60] John Hughes and Lars Pareto. 1999. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *ICFP 1999*. 70–81.

[61] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL 1996*. 410–423.

[62] Claire Jones. 1989. *Probabilistic Non-Determinism.* Ph.D. Dissertation. The University of Edinburgh.

[63] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *POPL 2010*. 223–236.

[64] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *FM 2009*. 354–369.

[65] David M. Kahn. 2017. Undecidable Problems for Probabilistic Network Programming. In *MFCS 2017*. 68:1–68:17.

[66] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *ESOP 2016*. 364–389.

[67] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2018. On the hardness of analyzing probabilistic programs. *Acta Informatica* (2018), 1–31.

[68] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *ESOP 2014*. 392–411.

[69] Chin Soon Lee. 2009. Ranking functions for size-change termination. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 10:1–10:42.

[70] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *POPL 2001*. 81–92.

[71] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2017. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 33.

[72] Daniel A Menascé. 2004. Response-time analysis of composite Web services. *IEEE Internet computing* 8, 1 (2004), 90–92.

[73] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[74] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *PLDI 2018*. 496–512.

[75] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *LICS 2016*. 672–681.

[76] Joël Ouaknine and James Worrell. 2012. Decision problems for linear recurrence sequences. In *International Workshop on Reachability Problems*. Springer, 21–28.

[77] Joël Ouaknine and James Worrell. 2015. On linear recurrence sequences and loop termination. *ACM SIGLOG News* 2, 2 (2015), 4–13.

[78] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI 2004*. 239–251.

[79] Meni Rosenfeld. 2011. Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980* (2011).

[80] DM Roy, VK Mansinghka, ND Goodman, and JB Tenenbaum. 2008. A stochastic programming perspective on nonparametric Bayes. In *Nonparametric Bayesian Workshop, Int. Conf. on Machine Learning*, Vol. 22. 26.

[81] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Constraint-based linear-relations analysis. In *SAS 2004*. Springer, 53–68.

[82] Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. 2015. Practical probabilistic programming with monads. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–176.

[83] Liyong Shen, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2013. Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. *J. Systems Science & Complexity* 26, 2 (2013), 291–301.

[84] Olha Shkaravska, Ron van Kesteren, and Marko C. J. D. van Eekelen. 2007. Polynomial Size Analysis of First-Order Functions. In *TLCA 2007*. 351–365.

[85] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *CAV 2014*. 745–761.

[86] Peter Smith et al. 2018. Hash Rate: The estimated number of tera hashes per second (trillions of hashes per second) the Bitcoin network is performing. https://www.blockchain.com/charts/hash-rate?scale=1&timespan=all.

[87] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor meets Scott: semantic foundations for probabilistic networks. In *POPL 2017*. 557–571.

[88] Kirack Sohn and Allen Van Gelder. 1991. Termination Detection in Logic Programs using Argument Sizes. In *PODS 1991*. 216–226.

[89] Akhilesh Srikanth, Burak Sahin, and William R. Harris. 2017. Complexity verification using guided theorem enumeration. In *POPL 2017*. 639–652.

[90] Sebastian Thrun. 2000. Probabilistic algorithms in robotics. *Ai Magazine* 21, 4 (2000), 93.

[91] Sebastian Thrun. 2002. Probabilistic robotics. *Commun. ACM* 45, 3 (2002), 52–57.

[92] 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

[93] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS 2013*. 43–62.

[94] Fabian Vogelsteller, Vitalik Buterin, et al. 2014. Ethereum whitepaper. (2014).

[95] Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *PLDI 2018*. 513–528.

[96] David Williams. 1991. *Probability with martingales*. Cambridge university press.

[97] Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. 2010. Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China* 4, 1 (2010), 1–16.

A Conditional Expectation

Let X be any random variable from a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ such that $\mathbb{E}(|X|) < \infty$. Then given any σ-algebra $\mathcal{G} \subseteq \mathcal{F}$, there exists a random variable (from $(\Omega, \mathcal{F}, \mathbb{P})$), conventionally denoted by $\mathbb{E}(X|\mathcal{G})$, such that

(E1) $\mathbb{E}(X|\mathcal{G})$ is $\mathcal{G}$-measurable, and

(E2) $\mathbb{E}(|\mathbb{E}(X|\mathcal{G})|) < \infty$, and

(E3) for all $A \in \mathcal{G}$, we have $\int_A \mathbb{E}(X|\mathcal{G})\, \mathrm{d}\mathbb{P} = \int_A X\, \mathrm{d}\mathbb{P}$.

The random variable $\mathbb{E}(X|\mathcal{G})$ is called the *conditional expectation* of X given $\mathcal{G}$. The random variable $\mathbb{E}(X|\mathcal{G})$ is a.s. unique in the sense that if Y is another random variable satisfying (E1)–(E3), then $\mathbb{P}(Y = \mathbb{E}(X|\mathcal{G})) = 1$.

Conditional expectation has the following properties for any random variables X, Y and $\{X_n\}_{n\in\mathbb{N}_0}$ (from a same probability space) satisfying $\mathbb{E}(|X|) < \infty, \mathbb{E}(|Y|) < \infty, \mathbb{E}(|X_n|) < \infty$ $(n \geq 0)$ and any suitable sub-σ-algebras $\mathcal{G}, \mathcal{H}$:

(E4) $\mathbb{E}(\mathbb{E}(X|\mathcal{G})) = \mathbb{E}(X)$;

(E5) if X is $\mathcal{G}$-measurable, then $\mathbb{E}(X|\mathcal{G}) = X$ a.s.;

(E6) for any real constants b, d,

$$\mathbb{E}(b \cdot X + d \cdot Y|G) = b \cdot \mathbb{E}(X|G) + d \cdot \mathbb{E}(Y|G) \text{ a.s.};$$

(E7) if $\mathcal{H} \subseteq \mathcal{G}$, then $\mathbb{E}(\mathbb{E}(X|\mathcal{G})|\mathcal{H}) = \mathbb{E}(X|\mathcal{H})$ a.s.;

(E8) if Y is $\mathcal{G}$-measurable and $\mathbb{E}(|Y|) < \infty, \mathbb{E}(|Y \cdot X|) < \infty$, then

$$\mathbb{E}(Y \cdot X|\mathcal{G}) = Y \cdot \mathbb{E}(X|\mathcal{G}) \text{ a.s.};$$

(E9) if X is independent of $\mathcal{H}$, then $\mathbb{E}(X|\mathcal{H}) = \mathbb{E}(X)$ a.s., where $\mathbb{E}(X)$ here is deemed as the random variable with constant value $\mathbb{E}(X)$;

(E10) if it holds a.s that $X \geq 0$, then $\mathbb{E}(X|\mathcal{G}) \geq 0$ a.s.;

(E11) if it holds a.s. that (i) $X_n \geq 0$ and $X_n \leq X_{n+1}$ for all n and (ii) $\lim_{n\to\infty} X_n = X$, then

$$\lim_{n\to\infty} \mathbb{E}(X_n|\mathcal{G}) = \mathbb{E}(X|\mathcal{G}) \text{ a.s.}$$

(E12) if (i) $|X_n| \leq Y$ for all n and (ii) $\lim_{n\to\infty} X_n = X$, then

$$\lim_{n\to\infty} \mathbb{E}(X_n|\mathcal{G}) = \mathbb{E}(X|\mathcal{G}) \text{ a.s.}$$

(E13) if $g : \mathbb{R} \to \mathbb{R}$ is a convex function and $\mathbb{E}(|g(X)|) < \infty$, then $g(\mathbb{E}(X|\mathcal{G})) \leq \mathbb{E}(g(X)|\mathcal{G})$ a.s.

We refer to [96, Chapter 9] for more details.

B Detailed Syntax

In the sequel, we fix two countable sets of *program variables* and *sampling variables*. W.l.o.g, these three sets are pairwise disjoint.

Informally, program variables are variables that are directly related to the control-flow of a program, while sampling variables reflect randomized inputs to the program. Every program variable holds an integer upon instantiation, while every sampling variable is bound to a discrete probability distribution.

The Syntax. Below we explain the grammar in in Figure 1.

- *Variables.* Expressions $\langle pvar \rangle$ (resp. $\langle rvar \rangle$) range over program (resp. sampling) variables.
- *Constants.* Expressions $\langle const \rangle$ range over decimal integers.
- *Arithmetic Expressions.* Expressions $\langle expr \rangle$ (resp. $\langle pexpr \rangle$) range over arithmetic expressions over both program and sampling variables (resp. program variables). As a theoretical paper, we do not fix the syntax for $\langle expr \rangle$ and $\langle pexpr \rangle$.
- *Boolean Expressions.* Expressions $\langle bexpr \rangle$ range over propositional arithmetic predicates over program variables.
- *Nondeterminism.* The symbol '$\star$' indicates a nondeterministic choice to be resolved in a demonic way.
- *Statements* $\langle stmt \rangle$. Assignment statements are indicated by ':='; '**skip**' is the statement that does nothing; conditional branches and nondeterminism are both indicated by the keyword '**if**'; while-loops are indicated by the keyword '**while**'; sequential compositions are indicated by semicolon; finally, tick statements are indicated by '**tick**'.

C Detailed Semantics

Informally, a control-flow graph specifies how values for program variables and the program counter change along an execution of a program. We refer to the status of the program counter as a *label*, and assign an initial label ℓ_{in} and a terminal label ℓ_{out} to the start and the end of the program. Moreover, we have five types of labels, namely *assignment*, *branching*, *probabilistic*, *nondeterministic* and *tick* labels.

- An *assignment* label corresponds to an assignment statement indicated by ':=', and leads to the next label right after the statement with change of values specified by the update function determined at the right-hand-side of ':='. The update function gives the next valuation on program variables, based on the current values of program variables and the sampled values for this statement.

- A *branching* label corresponds to a conditional-branching statement indicated by the keyword '**if**' or '**while**' together with a propositional arithmetic predicate ϕ over program variables (as the condition or the loop guard), and leads to the next label determined by ϕ without change on values.
- A *probabilistic* label corresponds to a probabilistic-branching statement indicated by the keywords '**if**' and '**prob**(p)' with $p \in [0,1]$, and leads to the labels of the **then**-branch with probability p and the **else**-branches with probability $1-p$, without change on values.
- A *nondeterministic* label corresponds to a nondeterministic-branching statement indicated by the keywords '**if**' and '$\star$', and leads to the labels of the **then**- and **else**-branches without change on values.
- A *tick* label corresponds to a tick statement '**tick**(q)' that triggers a cost/reward, and leads to the next label without change on values. The arithmetic expression q determines a *cost function* that outputs a real number (as the amount of cost/reward) upon the current values of program variables for this statement.

It is intuitively clear that any probabilistic program can be transformed into a CFG. We refer to existing results [18, 22] for a detailed transformation from programs to CFGs.

Based on CFGs, the semantics of the program is given by general state space Markov chains (GSSMCs) as follows.

Below we fix a probabilistic program W with its CFG in the form (1). To illustrate the semantics, we need the notions of *configurations*, *sampling functions*, *runs* and *schedulers* as follows.

Configurations. A *configuration* is a triple (ℓ, v) where $\ell \in L$ and $v \in Val_{V_p}$. We say that a configuration (ℓ, v) is *terminal* if $\ell = \ell_{\text{out}}$; moreover, it is *nondeterministic* if $\ell \in L_{\text{nd}}$. Informally, a configuration (ℓ, v) specifies that the next statement to be executed is the one labelled with ℓ and the current values of program variables is specified by the valuation v.

Sampling functions. A *sampling function* Υ is a function assigning to every sampling variable $r \in V_r$ a (possibly continuous) probability distribution over $\mathbb{R}$. Informally, a sampling function Υ specifies the probability distributions for the sampling of all sampling variables, i.e., for each $r \in V_r$, its sampled value is drawn from the probability distribution $\Upsilon(r)$.

Finite and infinite runs. A *finite run* ρ is a finite sequence $(\ell_0, v_0), \ldots, (\ell_n, v_n)$ of configurations. An *infinite run* is an infinite sequence $\{(\ell_n, v_n)\}_{n \in \mathbb{N}_0}$ of configurations. The intuition is that each ℓ_n and v_n are the current program counter and respectively the current valuation for program variables at the nth step of a program execution.

Schedulers. A scheduler σ is a function that assigns to every finite run ending in a nondeterministic configuration (ℓ, v) a transition with source label ℓ (in the CFG) that leads to the target label as the next label. Thus, based on the whole history of configuration visited so far, a scheduler resolves the choice between the **then**- and **else**-branch at a nondeterministic branch.

Based on these notions, we can have an intuitive description on an execution of a probabilistic program. Given a scheduler σ, the execution starts in an initial configuration (ℓ_0, v_0). Then in every step $n \in \mathbb{N}_0$, assuming that the current configuration is $c_n = (\ell_n, v_n)$, the following happens.

- If $\ell_n = \ell_{\text{out}}$ (i.e., the program terminates), then $(\ell_{n+1}, v_{n+1}) = (\ell_n, v_n)$. Otherwise, proceed as follows.
- A valuation $\mathbf{r}$ on the sampling variables is sampled w.r.t the probability distributions in the sampling function Υ.
- A transition $\tau = (\ell_n, \alpha^*, \ell^*)$ enabled at the current configuration (ℓ_n, v_n) is chosen, and then the next configuration is determined by the chosen transition. In detail, we have the following.
  - If $\ell_n \in L_a$, then τ is chosen as the unique transition from ℓ_n such that α^* is an update function, and the next configuration (ℓ_{n+1}, v_{n+1}) is set to be $(\ell^*, \alpha^*(v_n, \mathbf{r}))$.
  - If $\ell_n \in L_b$, then τ is chosen as the unique transition such that v_n satisfies the propositional arithmetic predicate α^*, and the next configuration (ℓ_{n+1}, v_{n+1}) is set to be (ℓ^*, v_n).
  - If $\ell_n \in L_p$ with the probability p specified in its corresponding statement, then τ is chosen to be the **then**-branch with probability p and the **else**-branch with probability $1-p$, and the next configuration (ℓ_{n+1}, v_{n+1}) is set to be (ℓ^*, v_n).
  - If $\ell_n \in L_{\text{nd}}$, then τ is chosen by the scheduler σ. That is, if $\rho = c_0 c_1 \cdots c_n$ is the finite path of configurations traversed so far, then τ equals $\sigma(c_0 c_1 \cdots c_n)$, and the next configuration (ℓ_{n+1}, v_{n+1}) is set to be (ℓ^*, v_n).
  - If $\ell_n \in L_t$, then τ is chosen as the unique transition from ℓ_n such that α^* is a cost function, then the next configuration (ℓ_{n+1}, v_{n+1}) is set to be (ℓ^*, v_n) and the statement triggers a cost of amount $\alpha^*(v_n)$.

In this way, the scheduler and random choices eventually produce a random infinite run in a probabilistic program. Then given any scheduler that resolves nondeterminism, the semantics of a probabilistic program is a GSSMC, where the kernel functions can be directly defined over configurations and based on the transitions in the CFG so that they specify the probabilities of the next configuration given the current configuration.

Given a scheduler σ and an initial configuration c, the GSSMC of a probabilistic program induces a probability space where the sample space is the set of all infinite runs, the sigma-algebra is generated from cylinder sets of infinite runs, and the probability measure is determined by the scheduler and the random sampling in the program.

D Proofs for Martingale Results

D.1 The Extended OST

In the proof of the extended OST, for a stopping time U and a nonnegative integer $n \in \mathbb{N}_0$, we denote by $U \wedge n$ the random variable $\min\{U, n\}$.

Theorem 5.2. (The Extended OST) Consider any stopping time U wrt a filtration $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and any martingale (resp. supermartingale) $\{X_n\}_{n=0}^{\infty}$ adapted to $\{\mathcal{F}_n\}_{n=0}^{\infty}$ and let $Y = X_U$. Then the following condition is sufficient to ensure that $\mathbb{E}(|Y|) < \infty$ and $\mathbb{E}(Y) = \mathbb{E}(X_0)$ (resp. $\mathbb{E}(Y) \leq \mathbb{E}(X_0)$):

- There exist real numbers $M, c_1, c_2, d > 0$ such that (i) for sufficiently large $n \in \mathbb{N}$, it holds that $\mathbb{P}(U > n) \leq c_1 \cdot e^{-c_2 \cdot n}$ and (ii) for all $n \in \mathbb{N}$, $|X_{n+1} - X_n| \leq M \cdot n^d$ almost surely.

Proof. We only prove the "$\leq$" case, the "=" case is similar. For every $n \in \mathbb{N}_0$,

$$\begin{aligned}
|X_{U\wedge n}| &= \left|X_0 + \sum_{k=0}^{U\wedge n-1} (X_{k+1} - X_k)\right| \\
&= \left|X_0 + \sum_{k=0}^{\infty} (X_{k+1} - X_k) \cdot \mathbf{1}_{U>k\wedge n>k}\right| \\
&\leq |X_0| + \sum_{k=0}^{\infty} |(X_{k+1} - X_k) \cdot \mathbf{1}_{U>k\wedge n>k}| \\
&\leq |X_0| + \sum_{k=0}^{\infty} |(X_{k+1} - X_k) \cdot \mathbf{1}_{U>k}| \quad .
\end{aligned}$$

Then

$$\begin{aligned}
&\mathbb{E}\left(|X_0| + \sum_{k=0}^{\infty} |(X_{k+1} - X_k) \cdot \mathbf{1}_{U>k}|\right) \\
= \;& \text{(By Monotone Convergence Theorem)} \\
&\mathbb{E}(|X_0|) + \sum_{k=0}^{\infty} \mathbb{E}(|(X_{k+1} - X_k) \cdot \mathbf{1}_{U>k}|) \\
= \;& \mathbb{E}(|X_0|) + \sum_{k=0}^{\infty} \mathbb{E}(|X_{k+1} - X_k| \cdot \mathbf{1}_{U>k}) \\
\leq \;& \mathbb{E}(|X_0|) + \sum_{k=0}^{\infty} \mathbb{E}\left(\lambda \cdot k^d \cdot \mathbf{1}_{U>k}\right) \\
= \;& \mathbb{E}(|X_0|) + \sum_{k=0}^{\infty} M \cdot k^d \cdot \mathbb{P}(U > k) \\
\leq \;& \mathbb{E}(|X_0|) + \sum_{k=0}^{\infty} M \cdot k^d \cdot c_1 \cdot e^{-c_2 \cdot k} \\
= \;& \mathbb{E}(|X_0|) + M \cdot c_1 \cdot \sum_{k=0}^{\infty} k^d \cdot e^{-c_2 \cdot k} \\
< \;& \infty \quad .
\end{aligned}$$

Thus, by Dominated Convergence Theorem and the fact that $X_U = \lim_{n\to\infty} X_{R\wedge n}$ a.s.,

$$\mathbb{E}(X_U) = \mathbb{E}\left(\lim_{n\to\infty} X_{U\wedge n}\right) = \lim_{n\to\infty} \mathbb{E}(X_{U\wedge n}) \quad .$$

Finally the result follows from properties for the stopped process $\{X_{U\wedge n}\}_{n\in\mathbb{N}_0}$ that

$$\mathbb{E}(X_U) \leq \mathbb{E}(X_0) \quad .$$

□

D.2 An Important Lemma

In this part, we prove an important lemma. Below we define the following sequences of (vectors of) random variables:

- $\overline{\mathbf{v}}_0, \overline{\mathbf{v}}_1, \ldots$ where each $\overline{\mathbf{v}}_n$ represents the valuation to program variables at the nth execution step of a probabilistic program;
- $\overline{\mathbf{u}}_0, \overline{\mathbf{u}}_1, \ldots$ where each $\overline{\mathbf{u}}_n$ represents the sampled valuation to sampling variables at the nth execution step of a probabilistic program;
- $\overline{\ell}_0, \overline{\ell}_1, \ldots$ where each $\overline{\ell}_n$ represents the label at the nth execution step of a probabilistic program.

Lemma D.1. *Let h be a PUCS and σ be any scheduler. Let the stochastic process $\{X_n\}_{n\in\mathbb{N}_0}$ be defined such that $X_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n)$.*

Then for all $n \in \mathbb{N}_0$, we have $\mathbb{E}(X_{n+1} + C_n | \mathcal{F}_n) \leq pre_h(\overline{\ell}_n, \overline{\mathbf{v}}_n)$.

Proof. For all $n \in \mathbb{N}_0$, from the program syntax we have

$$X_{n+1} = \mathbf{1}_{\overline{\ell}_n=\ell_{\text{out}}} \cdot X_n + Y_p + Y_a + Y_{nd} + Y_t + Y_b$$

where the terms are described below:

$$Y_p := \sum_{\ell \in L_{\text{p}}} \left[\mathbf{1}_{\overline{\ell}_n=\ell} \cdot \sum_{i \in \{0,1\}} \mathbf{1}_{B_\ell=i} \cdot h(\ell_{B_\ell=i}, \overline{\mathbf{v}}_n) \right]$$

where each random variable B_ℓ is the Bernoulli random variable for the decision of the probabilistic branch and $\ell_{B_\ell=0}, \ell_{B_\ell=1}$ are the corresponding successor locations of ℓ. Note that all B_ℓ 's and $\mathbf{u}$'s are independent of $\mathcal{F}_n$. In other words, Y_p describes the semantics of probabilistic locations.

$$Y_a := \sum_{\ell \in L_{\text{a}}} \mathbf{1}_{\overline{\ell}_n=\ell} \cdot h(\ell', F_\ell(\overline{\mathbf{v}}_n, \mathbf{u}))$$

describes the semantics of assignment locations where ℓ' is its successor label.

$$Y_{nd} := \sum_{\ell \in L_{\text{nd}}} \mathbf{1}_{\overline{\ell}_n=\ell} \cdot h(\sigma(\ell, \overline{\mathbf{v}}_n), \overline{\mathbf{v}}_n)$$

describes the semantics of nondeterministic locations, where $\sigma(-,-)$ here denotes the target location of the transition chosen by the scheduler σ.

$$Y_t := \sum_{\ell \in L_{\text{t}}} \mathbf{1}_{\overline{\ell}_n=\ell} \cdot h(\ell', \overline{\mathbf{v}}_n)$$

describes the semantics of tick locations.

$$Y_b := \sum_{\ell \in L_{\text{b}}} \left[\mathbf{1}_{\overline{\ell}_n=\ell} \cdot \sum_{i \in \{1,2\}} \mathbf{1}_{\overline{\mathbf{v}}_n \models \phi_i} \cdot h(\ell_i, \overline{\mathbf{v}}_n) \right]$$

describes the semantics of branching locations, where $\phi_1 = \phi, \phi_2 = \neg\phi$ and ℓ_1, ℓ_2 are the corresponding successor locations. Then from properties of conditional expectation, one obtains:

$$\begin{aligned} & \mathbb{E}(X_{n+1} + C_n | \mathcal{F}_n) \\ = \;& \mathbb{E}(X_{n+1} | \mathcal{F}_n) + \mathbb{E}(C_n | \mathcal{F}_n) \\ = \;& \mathbf{1}_{\overline{\ell}_n=\ell_{\text{out}}} \cdot X_n + Y'_p + Y'_a + Y_{nd} + Y_t + Y_b \\ & + \mathbf{1}_{\overline{\ell}_n \in L_{\text{t}}} \cdot C_n \end{aligned}$$

where

$$Y'_p := \sum_{\ell \in L_{\text{p}}} \left[\mathbf{1}_{\overline{\ell}_n=\ell} \cdot \sum_{i \in \{0,1\}} \mathbb{P}(B_\ell = i) \cdot h(\ell_{B_\ell=i}, \overline{\mathbf{v}}_n) \right]$$

and

$$Y'_a := \sum_{\ell \in L_{\text{a}}} \mathbf{1}_{\overline{\ell}_n=\ell} \cdot \mathbb{E}_{\mathbf{u}}(h(\ell', F_\ell(\overline{\mathbf{v}}_n, \mathbf{u})))$$

This follows from the facts that (i) $\mathbf{1}_{\overline{\ell}_n=\ell_{\text{out}}} \cdot X_n$, Y_{nd}, Y_t, Y_b are measurable in $\mathcal{F}_n$; (ii) $\mathbb{E}(C_n|\mathcal{F}_n) = \mathbf{1}_{\overline{\ell}_n=L_{\text{t}}} \cdot C_n$; (iii) for Y_p and Y_a, their conditional expectations are resp. Y'_p, Y'_a.

From (C3), when $\overline{\ell}_n \in L_{\text{p}} \cup L_{\text{a}} \cup L_{\text{b}}$, we have $pre_h(\overline{\ell}_n, \overline{\mathbf{v}}_n) = \mathbf{1}_{\overline{\ell}_n=\ell_{\text{out}}} \cdot X_n + Y'_p + Y'_a + Y_b$. When $\overline{\ell}_n \in L_{\text{t}}$, we have $pre_h(\overline{\ell}_n, \overline{\mathbf{v}}_n) = Y_t + C_n$. Then we get:

$$\mathbb{E}(X_{n+1} + C_n | \mathcal{F}_n) = pre_h(\overline{\ell}_n, \overline{\mathbf{v}}_n) \;\; .$$

When $\overline{\ell}_n \in L_{\text{nd}}$, we have $\mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n) = Y_{nd} \leq pre_h(\overline{\ell}_n, \overline{\mathbf{v}}_n)$. Hence the result follows. □

D.3 Polynomial Upper Cost Supermartingales (PUCSs)

Theorem 6.10. (Soundness of PUCS) Consider a nondeterministic probabilistic program P, with a linear invariant I and a PUCS h. If P satisfies the concentration property and the bounded update property, then $supval(\mathbf{v}) \leq h(\ell_{\text{in}}, \mathbf{v})$ for all initial valuations $\mathbf{v} \in I(\ell_{\text{in}})$.

Proof of Theorem 6.10. Fix any scheduler σ and initial valuation $\mathbf{v}$ for our simple while loop. Let T be the random variable that measures the number of loop iterations. By our assumption, $\mathbb{E}(T) < \infty$ under any scheduler. We recall the random variables $C_0, C_1, \ldots$ where each C_n represents the cost/reward accumulated during the nth loop iteration. We define the stochastic process $\{X_n\}_{n \in \mathbb{N}_0}$ by $X_n = h(\overline{\ell}_n, \overline{v}_n)$. Then we define the stochastic process $Y_0, Y_1, \ldots$ by:

$$Y_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n) + \textstyle\sum_{m=0}^{n-1} C_m \;\; .$$

Furthermore, we accompany $Y_0, Y_1, \ldots$ with the filtration $\mathcal{F}_0, \mathcal{F}_1, \ldots$ such that each $\mathcal{F}_n$ is the smallest sigma-algebra that makes all random variables from $\{\overline{\mathbf{v}}_0, \ldots, \overline{\mathbf{v}}_n\}$, $\{\overline{\mathbf{u}}_0, \ldots, \overline{\mathbf{u}}_n\}$ and $\{\overline{\ell}_0, \ldots, \overline{\ell}_{n-1}\}$ measurable. Then by Lemma D.1, we have $\mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n) \leq X_n$.
Thus we get:

$$\begin{aligned} & \mathbb{E}(Y_{n+1}|\mathcal{F}_n) \\ = \;& \mathbb{E}\Big(Y_n + h(\overline{\ell}_{n+1}, \overline{\mathbf{v}}_{n+1}) - h(\overline{\ell}_n, \overline{\mathbf{v}}_n) + C_n | \mathcal{F}_n\Big) \\ = \;& Y_n + \Big(\mathbb{E}\Big(h(\overline{\ell}_{n+1}, \overline{\mathbf{v}}_{n+1}) + C_n | \mathcal{F}_n\Big) - h(\overline{\ell}_n, \overline{\mathbf{v}}_n)\Big) \\ \leq \;& Y_n \end{aligned}$$

Hence, $\{Y_n\}_{n\in\mathbb{N}_0}$ is a supermartingale. Moreover, we have from the bounded-update property that

$$\begin{aligned}|Y_{n+1}-Y_n| &= |h_{n+1}+\sum_{m=1}^{n} C_m - h_n - \sum_{m=1}^{n-1} C_m| \\ &= |h_{n+1}-h_n+C_n| \\ &\le |h_{n+1}-h_n|+|C_n| \\ &\le M\cdot n^d + c''\cdot n \\ &\le \overline{M}^d \cdot n\end{aligned}$$

for some $\overline{M} > 0$.

Thus, by applying Optional Stopping Theorem, we obtain immediately that $\mathbb{E}(Y_T) \le \mathbb{E}(Y_0)$. By definition,

$$Y_T = h(\overline{\ell}_T, \overline{\mathbf{v}}_T) + \textstyle\sum_{m=1}^{T-1} C_m = \sum_{m=1}^{T-1} C_m \ .$$

It follows from (C2) that $\mathbb{E}(C_\infty) = \mathbb{E}(\sum_{m=1}^{T-1} C_m) \le \mathbb{E}(Y_0) = h(\ell_{\text{in}}, \mathbf{v})$. Since the scheduler σ is chosen arbitrarily, we obtain that $\mathit{supval}(\mathbf{v}) \le h(\ell_{\text{in}}, \mathbf{v})$. □

D.4 Polynomial Lower Cost Submartingales (PLCSs)

Theorem 6.12. (Soundness of PLCS) Consider a nondeterministic probabilistic program P, with a linear invariant I and a PLCS h. If P satisfies the concentration property and the bounded update property, then $\mathit{supval}(\mathbf{v}) \ge h(\ell_{\text{in}}, \mathbf{v})$ for all initial valuations $\mathbf{v} \in I(\ell_{\text{in}})$.

Proof of Theorem 6.12. We follow most definitions above. Fix any scheduler σ and initial valuation $\mathbf{v}$ for our simple while loop. Let T be the random variable that measures the number of loop iterations. By our assumption, $\mathbb{E}(T) < \infty$ under σ. We recall the random variables $C_0, C_1, \ldots$ where each C_n represents the cost/reward accumulated during the nth loop iteration. Let $T = \min\{n \mid \overline{\ell}_n = \ell_{\text{out}}\}$. We define the stochastic process $\{X_n\}_{n\in\mathbb{N}_0}$ by $X_n = h(\overline{\ell}_n, \overline{\nu}_n)$. Then we define the stochastic process $Y_0, Y_1, \ldots$ by:

$$Y_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n) + \textstyle\sum_{m=0}^{n-1} C_m \ .$$

Furthermore, we accompany $Y_0, Y_1, \ldots$ with the filtration $\mathcal{F}_0, \mathcal{F}_1, \ldots$ such that each $\mathcal{F}_n$ is the smallest sigma-algebra that makes all random variables from $\{\overline{\mathbf{v}}_0, \ldots, \overline{\mathbf{v}}_n\}$, $\{\overline{\mathbf{u}}_0, \ldots, \overline{\mathbf{u}}_{n-1}\}$ and $\{\overline{\ell}_0, \ldots, \overline{\ell}_n\}$ measurable. Then by (C3'), we have $\mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n) \ge h(\overline{\ell}_n, \mathbf{v}_n)$.

Thus we get:

$$\begin{aligned}&\mathbb{E}(Y_{n+1}|\mathcal{F}_n) \\ =\ &\mathbb{E}\Big(Y_n + h(\overline{\ell}_{n+1}, \overline{\mathbf{v}}_{n+1}) - h(\overline{\ell}_n, \overline{\mathbf{v}}_n) + C_n|\mathcal{F}_n\Big) \\ =\ &Y_n + \Big(\mathbb{E}\Big(h(\overline{\ell}_{n+1}, \overline{\mathbf{v}}_{n+1}) + C_n|\mathcal{F}_n\Big) - h(\overline{\ell}_n, \overline{\mathbf{v}}_n)\Big) \\ \ge\ &Y_n\end{aligned}$$

Hence, $\{Y_n\}_{n\in\mathbb{N}_0}$ is a submartingale, so $\{-Y_n\}_{n\in\mathbb{N}_0}$ is a supermartingale. Moreover, we have from the bounded update property that

$$\begin{aligned}|-Y_{n+1}-(-Y_n)| &= |Y_{n+1}-Y_n| \\ &= |h_{n+1}+\sum_{m=1}^{n} C_m - h_n - \sum_{m=1}^{n-1} C_m| \\ &= |h_{n+1}-h_n+C_n| \\ &\le |h_{n+1}-h_n|+|C_n| \\ &\le M\cdot n^d + c''\cdot n \\ &\le \overline{M}^d \cdot n\end{aligned}$$

for some $\overline{M} > 0$.

Thus, by applying Optional Stopping Theorem, we obtain immediately that $\mathbb{E}(-Y_T) \le \mathbb{E}(-Y_0)$, so $\mathbb{E}(Y_T) \ge \mathbb{E}(Y_0)$. By definition,

$$-Y_T = -h(\overline{\ell}_T, \overline{\mathbf{v}}_T) - \textstyle\sum_{m=1}^{T-1} C_m = -\sum_{m=1}^{T-1} C_m \ .$$

It follows from (C2) that $\mathbb{E}(C_\infty) = \mathbb{E}(\sum_{m=1}^{T-1} C_m) \ge \mathbb{E}(Y_0) = h(\mathbf{v})$. Since the scheduler σ is chosen arbitrarily, we obtain that $\mathit{supval}(\mathbf{v}) \ge h(\mathbf{v})$.

□

D.5 Unbounded Nonnegative Costs and General Updates

Theorem 6.14. (Soundness of nonnegative PUCS) Consider a nondeterministic probabilistic program P, with a linear invariant I and a nonnegative PUCS h. If all the step-wise costs in P are always nonnegative, then $\mathit{supval}(\mathbf{v}) \le h(\ell_{\text{in}}, \mathbf{v})$ for all initial valuations $\mathbf{v} \in I(\ell_{\text{in}})$.

Proof of Theorem 6.14. Fix any scheduler σ and initial valuation $\mathbf{v}$ for our simple while loop. Let T be the random variable that measures the number of loop iterations. By our assumption, $\mathbb{E}(T) < \infty$ under any scheduler. Define the following sequences of (vectors of) random variables:

- $\overline{\mathbf{v}}_0, \overline{\mathbf{v}}_1, \ldots$ where each $\mathbf{v}_n$ represents the valuation before the nth loop iteration of the while loop (so that $\overline{\mathbf{v}}_0 = \mathbf{v}$);
- $\overline{\mathbf{u}}_0, \overline{\mathbf{u}}_1, \ldots$ where each $\mathbf{u}_n$ represents the sampled valuation for the nth loop iteration of the while loop;
- $\overline{\ell}_0, \overline{\ell}_1, \ldots$ where each $\overline{\ell}_n$ represents the label decided by the scheduler for the nth loop iteration.

We recall the random variables $C_0, C_1, \ldots$ where each C_n represents the cost/reward accumulated during the nth loop iteration. Let $T = \min\{n \mid \overline{\ell}_n = \ell_{\text{out}}\}$. Then we define the

stochastic process $X_0, X_1, \ldots$ by:

$$X_n := h(\overline{\ell}_n, \overline{\mathbf{v}}_n) \ .$$

Furthermore, we accompany $X_0, X_1, \ldots$ with the filtration $\mathcal{F}_0, \mathcal{F}_1, \ldots$ such that each $\mathcal{F}_n$ is the smallest sigma-algebra that makes all random variables from $\{\overline{\mathbf{v}}_0, \ldots, \overline{\mathbf{v}}_n\}$, $\{\overline{\mathbf{u}}_0, \ldots, \overline{\mathbf{u}}_{n-1}\}$ and $\{\overline{\ell}_0, \ldots, \overline{\ell}_{n-1}\}$ measurable. Then by C3, we have $\mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n) \leq X_n$.

Thus we get:

$$\begin{aligned}
& \mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n) \leq X_n \\
\Leftrightarrow \quad & \mathbb{E}(\mathbb{E}(X_{n+1} + C_n|\mathcal{F}_n)) \leq \mathbb{E}(X_n) \\
\Leftrightarrow \quad & \mathbb{E}(X_{n+1} + C_n) \leq \mathbb{E}(X_n) \\
\Leftrightarrow \quad & \mathbb{E}(X_{n+1}) + \mathbb{E}(C_n) \leq \mathbb{E}(X_n) \\
& \text{(By Induction)} \\
\Leftrightarrow \quad & \mathbb{E}(X_{n+1}) + \sum_{m=0}^{n} \mathbb{E}(C_m) \leq \mathbb{E}(X_0) \\
& \text{(By C2)} \\
\Leftrightarrow \quad & \sum_{m=0}^{n} \mathbb{E}(C_m) \leq \mathbb{E}(X_0) \\
& \text{(By Monotone Convergence Theorem)} \\
\Leftrightarrow \quad & \mathbb{E}(\sum_{m=0}^{n} C_m) \leq \mathbb{E}(X_0)
\end{aligned}$$

The Induction is:

$$\begin{aligned}
& \mathbb{E}(X_n) + \mathbb{E}(C_{n-1}) \leq \mathbb{E}(X_{n-1}) \\
\Leftrightarrow \quad & \mathbb{E}(X_n) \leq \mathbb{E}(X_{n-1}) - \mathbb{E}(C_{n-1}) \\
& \mathbb{E}(X_{n-1}) + \mathbb{E}(C_{n-2}) \leq \mathbb{E}(X_{n-2}) \\
\Leftrightarrow \quad & \mathbb{E}(X_{n-1}) \leq \mathbb{E}(X_{n-2}) - \mathbb{E}(C_{n-2})
\end{aligned}$$

Then

$$\begin{aligned}
& \mathbb{E}(X_n) \leq \mathbb{E}(X_{n-1}) - \mathbb{E}(C_{n-1}) \\
\Leftrightarrow \quad & \mathbb{E}(X_n) \leq \mathbb{E}(X_0) - \sum_{m=0}^{n-1} \mathbb{E}(C_m)
\end{aligned}$$

Because all the PUCSs are nonnegative, we can get $\mathbb{E}(X_{n+1}) \geq 0$.

When $n \to \infty$, we obtain

$$\mathbb{E}(\sum_{m=0}^{\infty} C_m) \leq \mathbb{E}(X_0) \ .$$

Since the scheduler σ is chosen arbitrarily, we obtain that $\mathit{supval}(\mathbf{v}) \leq h(\mathbf{v})$.

□

E Details of Example 7.3

Since our algorithm is technical, we will illustrate the computational steps of the our algorithms on the example in Figure 2.

Example E.1 (Illustration of our algorithms)**.** We consider the example in Figure 2, and assign the invariant I as in Figure 9.

Firstly, the algorithm sets up a quadratic template h for a PUCS by setting $h(\ell_n, x, y) := a_{n1} \cdot x^2 + a_{n2} \cdot xy + a_{n3} \cdot x + a_{n4} \cdot y^2 + a_{n5} \cdot y + a_{n6}$ for each $\ell_n (n = 1, \ldots, 4)$ and $h(\ell_5, x, y) = 0$ because $\ell_5 = \ell_{\text{out}}$, where a_{np} are scalar variables for $n = 1, \ldots, 4$ and $p = 1, \ldots, 6$.

Next we compute the pre-expectations of this example.

- $\ell_1 \in L_{\text{b}}$,

$$\begin{aligned}
pre_h(\ell_1, x, y) &= \mathbf{1}_{\ell'=\ell_2} \cdot h(\ell_2, x, y) + \mathbf{1}_{\ell'=\ell_5} \cdot h(\ell_5, x, y) \\
&= \mathbf{1}_{\ell'=\ell_2} \cdot (a_{21} \cdot x^2 + a_{22} \cdot xy + a_{23} \cdot x \\
&\quad + a_{24} \cdot y^2 + a_{25} \cdot y + a_{26}) + \mathbf{1}_{\ell'=\ell_5} \cdot 0
\end{aligned}$$

- $\ell_2 \in L_{\text{a}}$,

$$\begin{aligned}
pre_h(\ell_2, x, y) &= \mathbb{E}_R[h(\ell_3, x + r, y)] \\
&= \mathbb{E}_R[a_{31} \cdot (x + r)^2 + a_{32} \cdot (x + r)y \\
&\quad + a_{33} \cdot (x + r) + a_{34} \cdot y^2 + a_{35} \cdot y + a_{36}] \\
&= a_{31} \cdot x^2 + a_{32} \cdot xy + (a_{33} - a_{31}) \cdot x \\
&\quad + a_{34} \cdot y^2 + (a_{35} - \frac{1}{2} a_{32}) \cdot y + a_{31} \\
&\quad - \frac{1}{2} a_{33} + a_{36}
\end{aligned}$$

- $\ell_3 \in L_{\text{a}}$,

$$\begin{aligned}
pre_h(\ell_3, x, y) &= \mathbb{E}_R[h(\ell_4, x, r')] \\
&= \mathbb{E}_R[a_{41} \cdot x^2 + a_{42} \cdot x \cdot r' + a_{43} \cdot x \\
&\quad + a_{44} \cdot r'^2 + a_{45} \cdot r' + a_{46}] \\
&= a_{41} \cdot x^2 + (\frac{1}{3} a_{42} + a_{43}) \cdot x + a_{44} \\
&\quad + \frac{1}{3} a_{45} + a_{46}
\end{aligned}$$

- $\ell_4 \in L_{\mathrm{t}}$,

$$\begin{aligned} pre_h(\ell_4, x, y) &= h(\ell_1, x, y) + \mathbb{E}_R(x \cdot y) \\ &= a_{11} \cdot x^2 + a_{12} \cdot xy + a_{13} \cdot x + a_{14} \cdot y^2 \\ &\quad + a_{15} \cdot y + a_{16} + xy \\ &= a_{11} \cdot x^2 + (a_{12} + 1) \cdot xy + a_{13} \cdot x \\ &\quad + a_{14} \cdot y^2 + a_{15} \cdot y + a_{16} \end{aligned}$$

Let the maximal number of multiplicands t in $Monoid(\Gamma)$ be 2, the form of Eq. (♯) is as following:

- (label 1) $(1)\ell' = \ell_2$

$$\begin{aligned} \Gamma &= \{x, x-1\} \\ u_1 &= 1, u_2 = x, u_3 = x-1, u_4 = x^2 - x, \\ u_5 &= x^2, u_6 = x^2 - 2x + 1; \\ g(\mathbf{x}) &= b_1 + b_2 x + b_3(x-1) + b_4(x^2 - x) + b_5 x^2 \\ &\quad + b_6(x^2 - 2x + 1) \\ &= (b_4 + b_5 + b_6)x^2 + (b_2 + b_3 - b_4 - 2b_6)x \\ &\quad + b_1 - b_3 + b_6 \end{aligned}$$

$(2)\ell' = \ell_5$

$$\begin{aligned} \Gamma &= \{x, 1-x\} \\ u_1 &= 1, u_2 = x, u_3 = 1-x, u_4 = x - x^2, \\ u_5 &= x^2, u_6 = 1 - 2x + x^2; \\ g(\mathbf{x}) &= b_7 + b_8 x + b_9(1-x) + b_{10}(x - x^2) + b_{11} x^2 \\ &\quad + b_{12}(1 - 2x + x^2) \\ &= (b_{11} + b_{12} - b_{10})x^2 + (b_8 - b_9 + b_{10} - 2b_{12})x \\ &\quad + b_7 + b_{12} \end{aligned}$$

for $b_i \geq 0, i = 1, \ldots, 12$.

- (label 2)

$$\begin{aligned} \Gamma &= \{x-1\} \\ u_1 &= 1, u_2 = x-1, u_3 = x^2 - 2x + 1; \\ g(\mathbf{x}) &= c_1 + c_2(x-1) + c_3(x^2 - 2x + 1) \\ &= c_3 x^2 + (c_2 - 2c_3)x + c_1 - c_2 + c_3 \end{aligned}$$

for $c_j \geq 0, j = 1, 2, 3$.

- (label 3)

$$\begin{aligned} \Gamma &= \{x\} \\ u_1 &= 1, u_2 = x, u_3 = x^2; \\ g(\mathbf{x}) &= d_1 + d_2 x + d_3 x^2 \\ &= d_3 x^2 + d_2 x + d_1 \end{aligned}$$

for $d_l \geq 0, l = 1, 2, 3$.

- (label 4)

$$\begin{aligned} \Gamma &= \{x, 1-y, 1+y\} \\ u_1 &= 1, u_2 = x, u_3 = 1-y, u_4 = 1+y, \\ u_5 &= x(1-y), u_6 = x(1+y), u_7 = (1-y)(1+y), \\ u_8 &= x^2, u_9 = (1-y)^2, u_{10} = (1+y)^2; \\ g(\mathbf{x}) &= e_1 + e_2 x + e_3(1-y) + e_4(1+y) + e_5 x(1-y) \\ &\quad + e_6 x(1+y) + e_7(1-y)(1+y) + e_8 x^2 \\ &\quad + e_9(1-y)^2 + e_{10}(1+y)^2 \\ &= e_8 x^2 + (e_6 - e_5)xy + (e_2 + e_5 + e_6)x \\ &\quad + (e_9 + e_{10} - e_7)y^2 + (e_4 - e_3 - 2e_9 + 2e_{10})y \\ &\quad + e_1 + e_3 + e_4 + e_7 + e_9 + e_{10} \end{aligned}$$

for $e_m \geq 0, m = 1, \ldots, 10$.

Then we extract instances conforming to pattern $g = h(\ell, v) - \mathrm{pre}_h(\ell, v)$ from C3.

- (C3, label 1)
  $(1)\ell' = \ell_2$

$$\begin{aligned} g(\mathbf{x}) &= h(\ell_1, x, y) - pre_h(\ell_1, x, y) \\ &= h(\ell_1, x, y) - h(\ell_2, x, y) \\ &= (a_{11} - a_{21})x^2 + (a_{12} - a_{22})xy + (a_{13} - a_{23})x \\ &\quad + (a_{14} - a_{24})y^2 + (a_{15} - a_{25})y + a_{16} - a_{26} \end{aligned}$$

$(2)\ell' = \ell_5$

$$\begin{aligned} g(\mathbf{x}) &= h(\ell_1, x, y) - pre_h(\ell_1, x, y) \\ &= h(\ell_1, x, y) - h(\ell_5, x, y) \\ &= a_{11} \cdot x^2 + a_{12} \cdot xy + a_{13} \cdot x + a_{14} \cdot y^2 \\ &\quad + a_{15} \cdot y + a_{16} \end{aligned}$$

- (C3, label 2)

$$\begin{aligned} g(\mathbf{x}) &= h(\ell_2, x, y) - pre_h(\ell_2, x, y) \\ &= (a_{21} - a_{31})x^2 + (a_{22} - a_{32})xy \\ &\quad +(a_{23} - a_{33} + a_{31})x + (a_{24} - a_{34})y^2 \\ &\quad +(a_{25} - a_{35} + \frac{1}{2}a_{32})y + a_{26} - a_{31} + \frac{1}{2}a_{33} - a_{36} \end{aligned}$$

- (C3, label 3)

$$\begin{aligned} g(\mathbf{x}) &= h(\ell_3, x, y) - pre_h(\ell_3, x, y) \\ &= (a_{31} - a_{41})x^2 + a_{32}xy + (a_{33} - \frac{1}{3}a_{42} - a_{43})x \\ &\quad +a_{34}y^2 + a_{35}y + a_{36} - a_{44} - \frac{1}{3}a_{45} - a_{46} \end{aligned}$$

- (C3, label 4)

$$\begin{aligned} g(\mathbf{x}) &= h(\ell_4, x, y) - pre_h(\ell_4, x, y) \\ &= (a_{41} - a_{11})x^2 + (a_{42} - a_{12} - 1)xy + (a_{43} - a_{13})x \\ &\quad +(a_{44} - a_{14})y^2 + (a_{45} - a_{15})y + a_{46} - a_{16} \end{aligned}$$

So we can translate them into systems of linear equalities.
(I) For label 1,

$$\begin{cases} a_{11} - a_{21} = b_4 + b_5 + b_6 \\ a_{12} - a_{22} = 0 \\ a_{13} - a_{23} = b_2 + b_3 - b_4 - 2b_6 \\ a_{14} - a_{24} = 0 \\ a_{15} - a_{25} = 0 \\ a_{16} - a_{26} = b_1 - b_3 + b_6 \end{cases}$$

and

$$\begin{cases} a_{11} = b_{11} + b_{12} - b_{10} \\ a_{12} = 0 \\ a_{13} = b_8 - b_9 + b_{10} - 2b_{12} \\ a_{14} = 0 \\ a_{15} = 0 \\ a_{16} = b_7 + b_{12} \end{cases}$$

(II) For label 2,

$$\begin{cases} a_{21} - a_{31} = c_3 \\ a_{22} - a_{32} = 0 \\ a_{23} - a_{33} + a_{31} = c_2 - 2c_3 \\ a_{24} - a_{34} = 0 \\ a_{25} - a_{35} + \frac{1}{2}a_{32} = 0 \\ a_{26} - a_{31} + \frac{1}{2}a_{33} - a_{36} = c_1 - c_2 + c_3 \end{cases}$$

(III) For label 3,

$$\begin{cases} a_{31} - a_{41} = d_3 \\ a_{32} = 0 \\ a_{33} - \frac{1}{3}a_{42} - a_{43} = d_2 \\ a_{34} = 0 \\ a_{35} = 0 \\ a_{36} - a_{44} - \frac{1}{3}a_{45} - a_{46} = d_1 \end{cases}$$

(IV) For label 4,

$$\begin{cases} a_{41} - a_{11} = e_8 \\ a_{42} - a_{12} - 1 = e_6 - e_5 \\ a_{43} - a_{13} = e_2 + e_5 + e_6 \\ a_{44} - a_{14} = e_9 + e_{10} - e_7 \\ a_{45} - a_{15} = e_4 - e_3 - 2e_9 + 2e_{10} \\ a_{46} - a_{16} = e_1 + e_3 + e_4 + e_7 + e_9 + e_{10} \end{cases}$$

Our target function is $h(\ell_1, x_0, y_0)$, where x_0, y_0 are the initial inputs and we fix x_0 to be a proper large integer, i.e. $x_0 = 100$, and y_0 to be 0.

$$\begin{aligned} min\ a_{11}x_0^2 + a_{13}x_0 + a_{16} \\ subject\ to\ (I), (II), (III), (IV) \\ b_i, c_j, d_l, e_m \geq 0, \forall i, j, l, m \end{aligned}$$

Finally, the algorithm gives the optimal solutions through linear programming such that:

$$\begin{aligned} h(\ell_1, x, y) &= \frac{1}{3} \cdot x^2 + \frac{1}{3} \cdot x \\ h(\ell_2, x, y) &= \frac{1}{3} \cdot x^2 + \frac{1}{3} \cdot x \\ h(\ell_3, x, y) &= \frac{1}{3} \cdot x^2 + \frac{2}{3} \cdot x \\ h(\ell_4, x, y) &= \frac{1}{3} \cdot x^2 + xy + \frac{1}{3} \cdot x \end{aligned}$$

To find a PLCS for this example, the steps are similar. The algorithm sets up a quadratic template h' for a PLCS with the similar form of the above PUCS h. By the same way, we

get the optimal solutions of the template h' and find they are the same as the PUCS's.

By the definition of PUCS and PLCS(see Section 6), we can conclude that this template h is both PUCS and PLCS, and we can get the accurate value of expected resource consumption that $\mathbb{E}(C_\infty) = h(\ell_1, x_0, y_0) = \frac{1}{3}x_0^2 + \frac{1}{3}x_0$.

F Experimental Results

We display ten examples in our paper: (1)Bitcoin Mining (see Figure 3); (2)Bitcoin Mining Pool (see Figure 4); (3)Queuing Network (see Figure 6); (4)Species Fight (see Figure 8); (5)Simple Loop (see Figure 2); (6)Nested Loop (see Figure 10); (7)Random Walk (see Figure 11); (8)2D Robot (see Figure 12); (9)Goods Discount (see Figure 13); (10)Pollutant Disposal (see Figure 14).

Firstly, we make a brief introduction about Goods discount,Pollutant disposal and Robot walking as follows:

Goods Discount. In most shops, goods will be sold at a discount after a certain amount of time, which will cause losses. And the remaining goods take up space, which also cause losses. When one piece of goods is sold, it will cause a reward. In this example, we model such goods discount. n is the number of goods which are newly on sale. d means the days after the goods are manufactured and each time one piece of goods is sold, d will be incremented by a random variable r which has a uniform distribution [1,2]. The program starts with the initial value $n = a$(big enough),$d = b$ and terminates if d exceeds 30 days, which happens with probability 1.

Pollutant Disposal. We consider pollutant disposal task. One pollutant treatment company has two machines A,B. When the company gains some pollutants, it has the probability 0.6 to assign these pollutants to Machine A, and the probability 0.4 to assign these pollutants to Machine B. Machine A can reduce r_1 pollutants ,but the remaining pollutant will cause r_1' pollutants, while Machine B can reduce r_2 pollutants ,but the remaining pollutant will cause r_2' pollutants. r_1, r_2 are *integer-valued* random variables which have an equivalent sampling rate between 1 and 10. r_1', r_2' are *integer-valued* random variables which have an equivalent sampling rate between 2 and 8. When the machine reduce one pollutant ,it will cause a reward. At the end of each iteration, the remaining pollutants will also cause losses.

2D Robot. We consider robot walking in 2D. Suppose the robot is located below the line $y = x$, and we want it to cross this line. There are nine direction orders: {0:Nord,1:South,2:East,3:West,4:Northeast,5:Southeast,6:Northwest,7:Southwest,8:Stay}. At each iteration, the direction order d chooses a moving direction by some probability. And the robot has a step size which is a uniformly random variable between 1 and 3. At the end of each iteration, it will cause a loss decided by the distance between the robot and the line $y = x$.

Next, we choose one instance of each examples (see Table 2), and list the corresponding $h(\ell_{\text{in}}, \mathbf{v})$ for PUCSs and PLCSs. (see Table 3)

Finally, we make a sampling of each example, compare upper and lower bounds with its simulations (except the two Bitcoin examples due to their nondeterminism).

```
1 : while i ≥ 1 do
2 :     x := i ;
3 :     while x ≥ 1 do
4 :         x := x + r ;
5 :         y := r' ;
6 :         tick(y)
          od
7 :   i := i + r'' ;
8     z := r''' ;
9 :   tick(−z ∗ i)
      od
10 :
```

Figure 10. A Nested Loop Example

```
ℙ(r = 1) = 0.25, ℙ(r = −1) = 0.75
while x ≤ n do
    if prob(0.6) then
        x := x+1
    else
        x := x−1
    fi ;
    y = r ;
    tick(y)
od
```

Figure 11. rdwalk

Benchmark Program	$\mathbf{v_0}$	**PUCS** $h(\ell_{\text{in}}, \mathbf{v})$	**PLCS** $h(\ell_{\text{in}}, \mathbf{v})$
Bitcoin Mining (Figure 3)	$x_0 = 100$ $(degree = 2, K = 2)$	$1.475 - 1.475 \cdot x$	$-1.5 \cdot x$
Bitcoin Mining Pool (Figure 4)	$y_0 = 100$ $(degree = 2, K = 2)$	$-7.375 \cdot y^2 - 41.62 \cdot y + 49.0$	$-7.5 \cdot y^2 - 67.5 \cdot y$
Queuing Network (Figure 6)	$n_0 = 320$ $(degree = 3, K = 3)$	$0.0492 \cdot n - 0.0492 \cdot i + 0.0103 \cdot l_1^2 + 0.00342 \cdot l_2^3 + 0.00726 \cdot l_2^2 + 0.0492$	$0.0384 \cdot n - 0.0384 \cdot i - (1.76 \times 10^{-4}) \cdot l_1^2 - 0.00854 \cdot l_1 \cdot l_2^2 - (8.16 \times 10^{-5}) \cdot l_2^3 - 0.00173 \cdot l_2^2 + 0.0384$
Species Fight (Figure 8)	$a_0 = 16, b_0 = 10$ $(degree = 3, K = 3)$	$40 \cdot a \cdot b - 180 \cdot b - 180 \cdot a + 810$	–
Figure 2	$x_0 = 200$ $(degree = 2, K = 2)$	$\frac{1}{3} \cdot x^2 + \frac{1}{3} \cdot x$	$\frac{1}{3} \cdot x^2 + \frac{1}{3} \cdot x - \frac{2}{3}$
Nested Loop	$i_0 = 150$ $(degree = 2, K = 2)$	$\frac{1}{3} \cdot i^2 + i$	$\frac{1}{3} \cdot i^2 - \frac{1}{3} \cdot i$
Random Walk	$x_0 = 12, n_0 = 20$ $(degree = 2, K = 2)$	$2.5 \cdot x - 2.5 \cdot n$	$2.5 \cdot x - 2.5 \cdot n - 2.5$
2D Robot	$x_0 = 100, y_0 = 80$ $(degree = 2, K = 2)$	$1.728 \cdot x^2 - 3.456 \cdot x \cdot y + 31.45 \cdot x + 1.728 \cdot y^2 - 31.45 \cdot y + 126.5$	$1.728 \cdot x^2 - 3.456 \cdot x \cdot y + 31.45 \cdot x + 1.728 \cdot y^2 - 31.45 \cdot y$
Goods Discount	$n_0 = 200, d_0 = 1$ $(degree = 2, K = 2)$	$0.00667 \cdot d \cdot n - 0.7 \cdot n - 3.803 \cdot d + 0.00222 \cdot d^2 + 119.4$	$0.00667 \cdot d \cdot n - 0.7133 \cdot n - 3.812 \cdot d + 0.00222 \cdot d^2 + 112.4$
Pollutant Disposal	$n_0 = 200$ $(degree = 2, K = 2)$	$-0.2 \cdot n^2 + 50.2 \cdot n$	$-0.2 \cdot n^2 + 50.2 \cdot n - 482.0$

Table 3. Corresponding $h(\ell_{\text{in}}, \mathbf{v})$ for PUCSs and PLCSs.

```
x := a ; y := b ;
while y ≤ x do
    if prob(0.2) then
        y := y + r_0
    else if prob(0.125) then
            y := y − r_1
    else if prob(0.143) then
            x := x + r_2
    else if prob(0.167) then
            x := x − r_3
    else if prob(0.2) then
            x := x + r_4 ;
            y := y + r'_4
    else if prob(0.25)then
            x := x + r_5 ;
            y := y − r'_5
    else if prob(0.333) then
            x := x − r_6 ;
            y := y + r'_6
    else if prob(0.5) then
            x := x − r_7 ;
            y := y − r'_7
        else skip
    fi fi fi fi fi fi fi fi ;
    tick(0.707 ∗ (x − y))
od
```

Figure 12. 2D Robot

```
n := a ; d := 1 ;
while d ≤ 30 and n >= 1 do
    n := n − 1 ;
    tick(5) ;
    d := d + r ;
    tick(−0.01 ∗ n)
od ;
tick(−0.5 ∗ n)
```

Figure 13. Goods Discount

```
n := a ;
while n ≥ 10 do
    if prob(0.6) then
        x := r_1 ;
        n := n − x + r'_1 ;
        tick(5 ∗ x)
    else
        y := r_2 ;
        n := n − y + r'_2 ;
        tick(5 ∗ y)
    fi ;
    tick(−0.2 ∗ n)
od
```

Figure 14. Pollutant Disposal

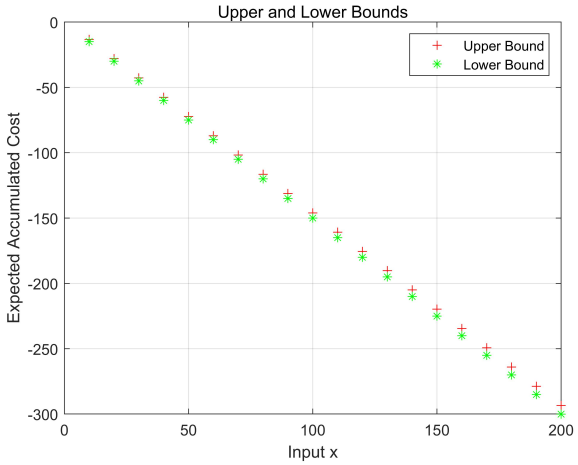

Figure 15. Bitcoin Mining

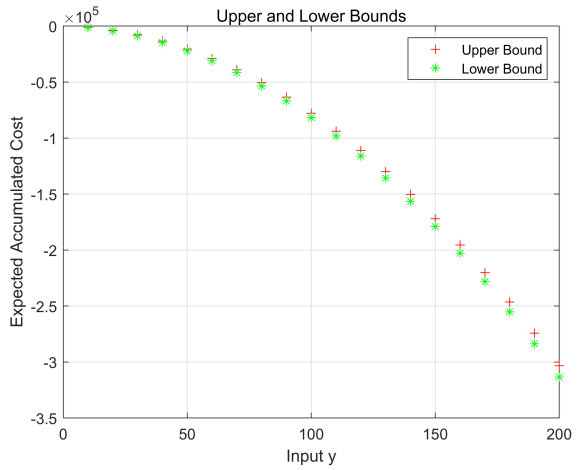

Figure 16. Bitcoin Mining Pool

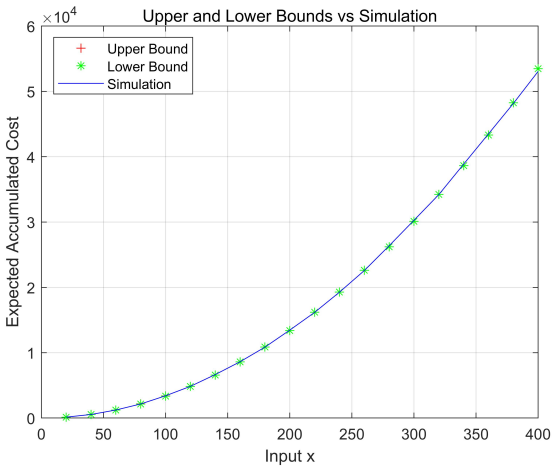

Figure 19. Simple Loop

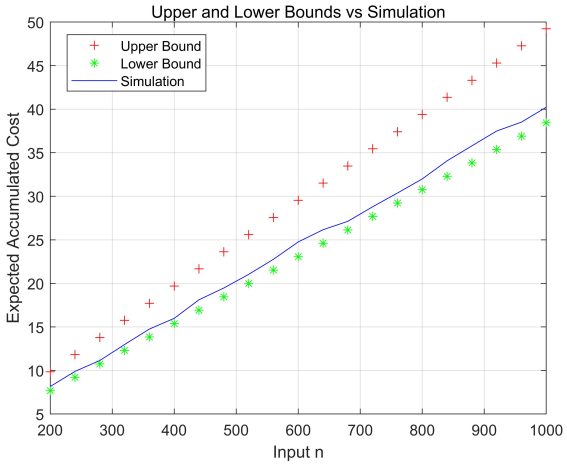

Figure 17. Queuing Network

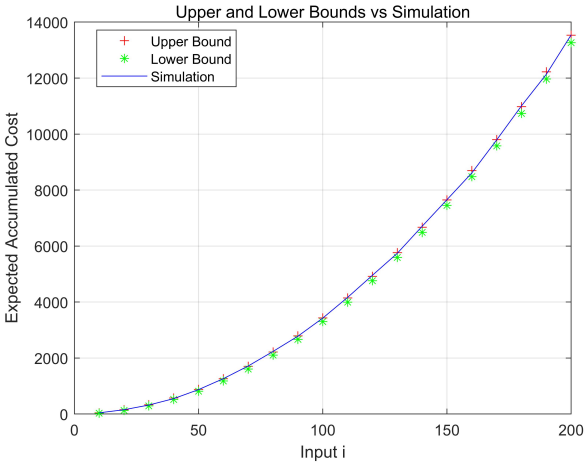

Figure 20. Nested Loop

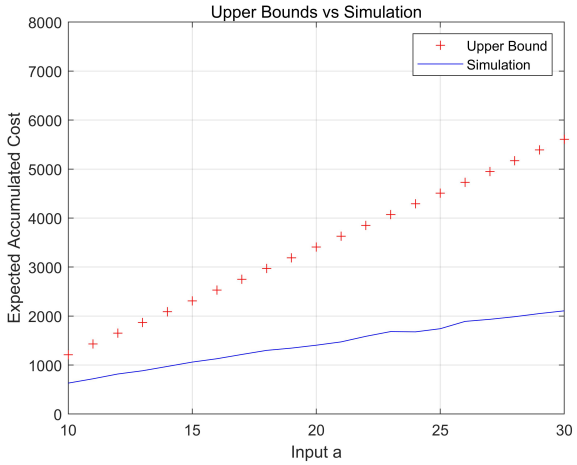

Figure 18. Species Fight

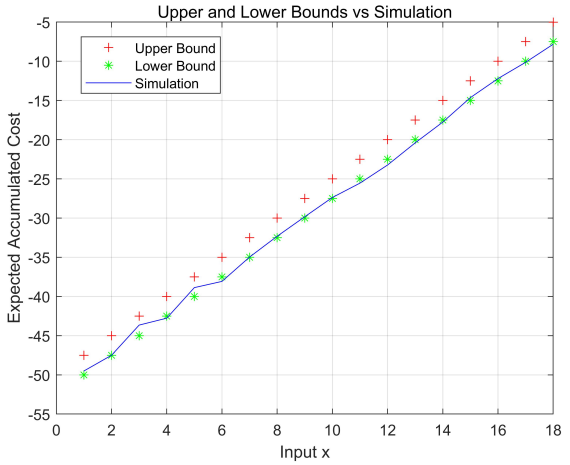

Figure 21. Random Walk

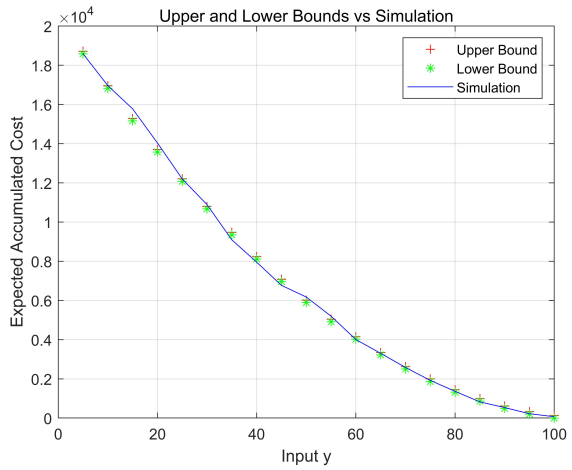

Figure 22. 2D Robot

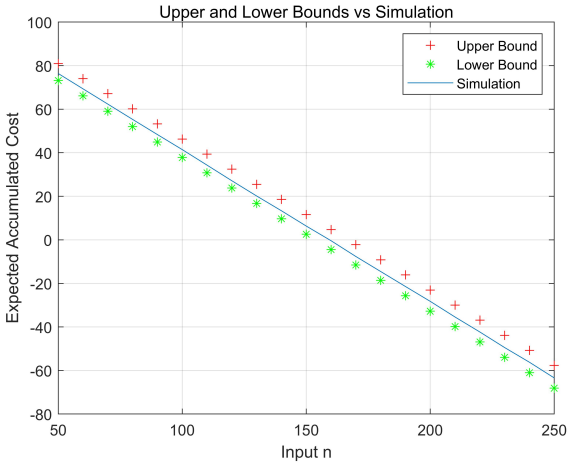

Figure 23. Goods Discount

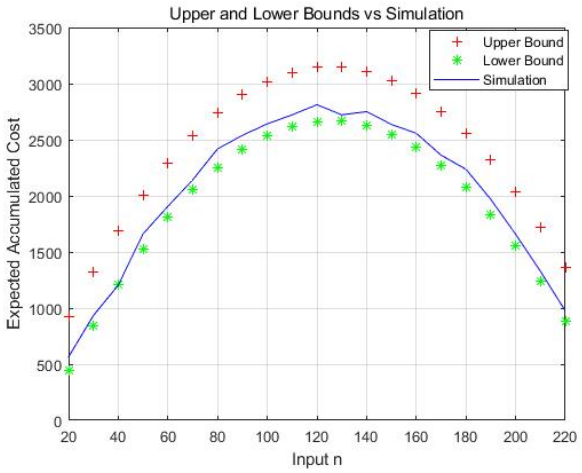

Figure 24. Pollutant Disposal