Piecewise Analysis of Probabilistic Programs via k-Induction

ANONYMOUS AUTHOR(S)

In probabilistic program analysis, quantitative analysis aims at deriving tight numerical bounds for probabilistic properties such as expectation and assertion probability. Most previous works consider numerical bounds over the whole program state space monolithically and do not consider piecewise bounds. Not surprisingly, monolithic bounds are either conservative, or not expressive and succinct enough in general. To derive better bounds, we propose a novel approach for synthesizing piecewise bounds over probabilistic programs. First, we show how to extract useful piecewise information from latticed k-induction operators, and combine the piecewise information with Optional Stopping Theorem to obtain a general approach to derive piecewise bounds over probabilistic programs. Second, we develop algorithms to synthesize piecewise polynomial bounds, and show that the synthesis can be reduced to bilinear programming in the linear case, and soundly relaxed to semidefinite programming in the polynomial case. Experimental results show that our approach generates tight piecewise bounds for a wide range of benchmarks when compared with the state of the art.

1 INTRODUCTION

Probabilistic programming [30, 37, 52] is a programming paradigm that extends classical programming languages with probabilistic statements such as sampling and probabilistic branching, and provides a powerful modelling mechanism for randomized algorithms [6], machine learning [12], reliability engineering [14], etc. Therefore, analysis of probabilistic programs is becoming increasingly significant, and attracting more and more attention in recent years.

In this work, we consider the quantitative analysis problem that aims at automated approaches that derive quantitative bounds for probabilistic programs. Common quantitative properties include expected runtime [1, 28, 34, 35], expected resource consumption [45, 53, 56], sensitivity [2], assertion probabilities [19, 51, 55], and so forth. Most existing works focus on deriving numerical bounds instead of solving the semantic equations exactly, as the latter is impossible theoretically in general. In the literature, various approaches have been proposed to address the quantitative analysis problem, including template-based constraint solving [15, 16, 18, 31], trace abstraction [50], sampling [47], etc. Most of these approaches consider to synthesize a monolithic bound over the whole state space of a probabilistic program of interest, and have the following disadvantages: First, a monolithic bound is either too conservative (e.g., only very coarse bounds exist) or not succinct enough (e.g., although tight monolithic bounds exist, the tightness usually requires complicated polynomials with higher degree). Second, it may be even worse that no monolithic polynomial bounds exist.

It is straightforward to observe that piecewise bounds are more accurate than monolithic bounds. Moreover, a recent work [9] demonstrates that probabilistic program analysis requires piecewise feature. However, the synthesis of piecewise bounds for probabilistic programs is not well investigated in the literature. To our best knowledge, a handful relevant work is by [10]. They propose an approach for generating (piecewise) invariants to *verify* user-provided linear bounds for probabilistic programs with discrete probabilistic choices, which is based on Counterexample-Guided Inductive Synthesis (CEGIS) and template refinement. Another relevant work is [5] that proposes a data-driven approach that can synthesize piecewise (sub-)invariants over probabilistic programs with discrete probabilistic choices. Their approach prefers a suitable list of numerical program features (such as multiplication expressions over variables), which requires prior knowledge of the program or user's assistance. Both of these related works require a bound to be verified as an additional program input when synthesizing (super-/sub-) invariants.

In this work, we propose a novel automated approach that synthesizes piecewise polynomial bounds for probabilistic programs with discrete probability choices without user-provided bounds or piecewise features to assist the derivation of the piecewise bound. The challenges are that (a) We need to resolve a good criterion to partition the state space of a probabilistic program into multiple parts in order to derive the form of the target piecewise bound. (b) We need to devise efficient algorithms to synthesize piecewise bounds given the criterion. Our detailed contributions to address these challenges are as follows.

To address the first challenge, we consider latticed k-induction operators [11, 40]. k-induction is a powerful proof tactics in software and hardware verification that generalizes normal inductive reasoning [22, 23, 38, 49]. Latticed k-induction [11, 40] further adapts k-induction to lattices and has application in probabilistic program analysis [11]. We develop a novel combination between operators from latticed k-induction and Optional Stopping Theorem (see the classical Optional Stopping Theorem (OST) [58, Chapter 10]). Our combination allows to synthesize both upper and lower bounds for quantitative properties over probabilistic programs without requiring a global bound of program values (such as non-negativity in [10, 11, 40]). Moreover, the combination itself is non-trivial, since we observe that an extended version of OST from [57] is needed and the classical OST does not suffice. As a by-product, we slightly extend existing latticed k-induction operators.

To address the second challenge, we propose novel algorithms for synthesizing piecewise linear and polynomial bounds w.r.t our combination of latticed k-induction and OST. It is important to observe that the latticed k-induction involves *minimum/maximum* operation, and therefore increases the difficulty to synthesize a bound algorithmically. We first introduce a key improvement in time efficiency on the unrolling of the k-induction operators. Then, we show that the synthesis of piecewise linear bounds can be equivalently transformed into a bilinear programming problem. A bilinear programming problem is that the variables can be decomposed into two groups so that within each group of variables the constraints are linear, and is a special non-convex programming that admits efficient constraint solving [41]. Finally, since even on the linear benchmarks we require piecewise polynomials to upper/lower bound the quantitative properties, we show that the synthesis of the more general piecewise polynomial bounds can be soundly relaxed to semidefinite programming. Experimental results over an extensive set of benchmarks that includes various benchmarks from the literature show that our approach is capable of generating tight or even accurate piecewise bounds and can solve benchmarks that previous approaches could not handle.

Technical Contributions. Approaches with latticed k-induction has inherent combinatorial explosion [11, 40]. To address the difficulty, we propose two techniques. The first is a heuristic selection of a small part of the functions in the minimum operation of latticed k-induction. The second is the sound relaxation that over-approximates the minimum operation with convex combination.

2 PRELIMINARIES

In this section, we briefly review probability theory, define the k-induction operators, present the probabilistic loops under consideration, nd finally formulate the problem of interest.

2.1 Probability Theory and Martingales

Consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the sample space, $\mathcal{F}$ is a σ-algebra on Ω and $\mathbb{P} : \mathcal{F} \to [0,1]$ is a probability measure on the measurable space $(\Omega, \mathcal{F})$. A *random variable* is an $\mathcal{F}$-measurable function $X : \Omega \to \mathbb{R} \cup \{+\infty, -\infty\}$, i.e., a function satisfying that for all $d \in \mathbb{R} \cup \{+\infty, -\infty\}$, $\{\omega \in \Omega : X(\omega) \leq d\} \in \mathcal{F}$. The *expectation* of a random variable X, denoted by $\mathbb{E}(X)$, is the Lebesgue integral of X w.r.t. $\mathbb{P}$, i.e., $\mathbb{E}(X) = \int X d\mathbb{P}$. 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* w.r.t. $\{\mathcal{F}_n\}_{n=0}^{\infty}$ is a random variable $\tau : \Omega \to \mathbb{N} \cup \{0, \infty\}$ such that for every $n \geq 0$, the event $\{\tau \leq n\} \in \mathcal{F}_n$, i.e., $\{\omega \in \Omega : \tau(\omega) \leq n\} \in \mathcal{F}_n$. Intuitively, τ is

$$C ::= \text{skip} \mid x := e \mid x :\approx \mu \mid C;C \mid \{C\}\ [p]\ \{C\} \mid \text{if } (\varphi)\ \{C\} \text{ else } \{C\}$$
$$\varphi ::= e < e \mid \neg\varphi \mid \varphi \wedge \varphi \qquad e ::= c \mid x \mid e \cdot e \mid e + e \mid e - e$$

Fig. 1. Syntax of Loop Guard and Body in the form (1)

interpreted as the time at which the stochastic process shows a desired behavior. 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 $(\Omega, \mathcal{F}_n, \mathbb{P})$. 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 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 Williams [58] for more details about martingale theory. Applying martingales for probabilistic programs analysis is well-studied [15, 16, 19].

2.2 k-Induction Operators

To present k-induction operators, we briefly review lattice theory. Informally, a lattice is a partially ordered set $(E, \sqsubseteq)$ (where E is a set and $\sqsubseteq$ is a partial order on E) equipped with a *meet* operation $\sqcap$ and a *join* operation $\sqcup$. Given two elements $u, v \in E$, the meet $u \sqcap v$ is defined as the infimum of $\{u, v\}$ and dually the join $u \sqcup v$ is defined as the supremum of $\{u, v\}$. A partially ordered set $(E, \sqsubseteq)$ is a *lattice* if for any $u, v \in E$, we have that both $u \sqcap v$ and $u \sqcup v$ exist. Given a lattice $(E, \sqsubseteq)$, we say that an operator $\Phi : E \longrightarrow E$ is *monotone* if for all $u, v \in E$, $u \sqsubseteq v$ implies $\Phi(u) \sqsubseteq \Phi(v)$. Throughout this section, we fix a lattice $(E, \sqsubseteq)$ and a monotone operator $\Phi : E \longrightarrow E$.

We recall the k-induction operator given in [11] as follows, which we refer to as the *upper* k-induction operator.

Definition 2.1 (Upper k-Induction Operator [11]). Given any element $u \in E$, the upper k-induction operator Ψ_u w.r.t. u and the monotone operator Φ is defined by: $\Psi_u : E \longrightarrow E, v \mapsto \Phi(v) \sqcap u$.

Below we propose a dual version for the upper k-induction operator. The intuition is simply to replace the meet operation with join. We call this dual operator as the *lower* k-induction operator.

Definition 2.2 (Lower k-Induction Operator). Let $u \in E$. The *dual* k-induction operator Ψ'_u w.r.t. u and the aforementioned monotone operator Φ is defined by: $\Psi'_u : E \longrightarrow E, v \mapsto \Phi(v) \sqcup u$.

Remark 1. *Alterative formulation of the k-induction operators have also been proposed in [40]. In Appendix A, We show that these formulation are essential equivalent to the definitions adopted in this work. Therefore, in the rest of this paper, we focus exclusively on the upper and lower k-induction operators defined above.* □

2.3 Probabilistic Loops

In this work, we use simple probabilistic while loops of the form (1) for easing the explanation of our basic idea, and will discuss how to extend our approach to general probabilistic while loops like nested loops without substantial changes in Section 5.2. Below we define the class of single probabilistic loops.

Syntax. A probabilistic while loop takes the form

$$\textbf{while}\ (\varphi)\ \{C\} \tag{1}$$

where φ is the loop guard and C is the loop body without loops. Formally, the loop guard φ and loop body C are generated by the grammar in Figure 1, where x is a program variable taken from a countable set Vars of variables, $c \in \mathbb{R}$ is a real constant, e is an arithmetic expression that involves addition and multiplication, φ is a formula over program variables that is a Boolean combination of arithmetic inequalities, and μ is a predefined probability distribution. In this work, we consider μ to be a finite discrete probability distribution (i.e., distributions with a finite support) such as Bernoulli distribution and discrete uniform distribution. The semantics of skip, assignment, sequential composition, conditional, and while statement can be understood as their counterparts in imperative programs. The semantics of a probabilistic choice $\{C_1\}[p]\{C_2\}$ is that flips a coin with bias $p \in [0,1]$ and executes the statement C_1 if the coin yields head, and C_2 otherwise. The semantics of a sampling statement $x :\approx \mu$ samples a value according to the predefined distribution μ and assigns the value to the variable x.

Given a probabilistic while loop, a *program state* is a function that maps every program variable to a real number. We denote by S the set of program states. The initial state for a probabilistic while loop is denoted by s^*. The evaluation $\varphi(s)$ of a logical formula φ and the evaluation $e(s)$ of an arithmetic expression e over a program state s are defined in the standard way. $\varphi(s) = \texttt{true}$ is denoted by $s \models \varphi$.

Semantics. The semantics of a probabilistic loop of the form (1) can be interpreted as a discrete-time Markov chain, where the state space is the set of all program states S, and the transition probability function $\mathbf{P}$ is given by the loop body C and determines the probability $\mathbf{P}(s, s')$ for $s, s' \in S$, meaning the probability producing output state s' from input state s. If the loop guard $\varphi(s)$ evaluates to false, then we treat the program state s as a sink state, that is $\mathbf{P}(s, s) = 1$ and $\mathbf{P}(s, s') = 0$ for $s \neq s'$.

Given the Markov chain of a probabilistic while loop as described above, a *path* is an infinite sequence $\pi = s_0, s_1, \ldots, s_n, \ldots$ of program states such that $\mathbf{P}(s_n, s_{n+1}) > 0$ for all $n \geq 0$. Intuitively, each s_n corresponds to the state right before the $(n+1)$-th loop iteration. A program state s is *reachable* from an initial program state s^* if there exists a path $\pi = s_0, s_1, \ldots$ such that $s_0 = s^*$ and $s_n = s$ for some $n \geq 0$, and define $Reach(s^*)$ as the set of reachable states starting from the initial state s^*. By the standard cylinder construction (see e.g. [4, Chapter 10]), the Markov chain with a designated initial program state s^* for the probabilistic loop induces a probability space over paths and reachable states. We denote the probability measure in this probability space by $\mathbb{P}_{s^*}$ and its related expectation operator by $\mathbb{E}_{s^*}$.

Problem formulation. Given a probabilistic loop P in the form (1), assuming that P terminates with probability 1, a *return function* f is a function $f : S \longrightarrow \mathbb{R}$ that is used to specify the output of the loop P in the sense that when the loop P terminates at a program state s, then the return value is given as $f(s)$. A return function is *piecewise polynomial* if it can be expressed as a piecewise polynomial expression in program variables. We denote by X_f the random variable for the return value of the loop given a return function f. In this work, we consider the following problem: Given a probabilistic while loop P in the form (1) and a piecewise polynomial return function f, synthesize *piecewise upper and lower bounds* on the expected value of X_f.

3 AN OVERVIEW OF OUR APPROACH

Our approach falls in the background of (latticed) k-induction [11, 40]. k-induction is an induction principle that generalizes the standard induction by considering k consecutive transitions together in the inductive condition. Roughly speaking, given a predicate P to be proved via induction, the k-induction principle considers the inductive condition as $(P(x_1) \wedge \cdots \wedge P(x_k)) \longrightarrow P(x_{k+1})$, for which the premise $P(x_1) \wedge \cdots \wedge P(x_k)$ means that the predicate P holds for k consecutive transitions, and the whole condition states that if P holds for k consecutive transitions, then P holds after these consecutive transitions. In particular, 1-induction coincides with the usual inductive condition.

Latticed k-induction [11] adapts the idea of k-induction to lattices for deriving bounds of fixed points. It considers k consecutive applications of a monotone operator over a lattice and applies the *meet/join* operations iteratively in the k consecutive applications. The parameter k here does not matter in the monotone operator (see Definitions 2.1 and 2.2), but is the number of iterative applications (see Definition 4.5) when the operator is applied. In this work, we propose a novel combination of latticed k-induction operators and Optional Stopping Theorem (OST), and propose novel algorithms for deriving piecewise linear and polynomial bounds on probabilistic programs.

We illustrate the main idea of our approach via the following example, which is a discretized version of the Growing Walk in Beutner et al. [12]:

$$\text{Growing Walk}: \quad \texttt{while}\,(\,0 \leq x\,)\,\{\{x := x+1; y := y+x\}\ [0.5]\ \{x := -1\}\}$$

The example models a simple random walk where the step size x is increased by 1 with one half probability, and set to -1 with the other half probability. The program terminates when x becomes negative. The objective is to analyze the expected value of the return function $f(x,y) = y$, which corresponds to the total traveled distance y, after the program terminates. We take the synthesis of piecewise linear upper bound as an example.

Step 1: Establishing k-induction operators. Let $\overline{\Phi}_f$ be the operator

$$\overline{\Phi}_f(h(x,y)) := [x < 0] \cdot y + [x \geq 0](0.5 \cdot h(x+1, y+x+1) + 0.5 \cdot h(-1,y))$$

for function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and $[x \geq 0]$ denotes the Iverson-bracket of the predicate $x \geq 0$, which evaluates to 1 if $x \geq 0$ holds at state s and 0 otherwise. Intuitively, $\overline{\Phi}_f$ outputs y if the loop guard $x \geq 0$ is violated, and the expected value of $h(x,y)$ after the execution of the loop body $\{x := x+1; y := y+x\}$ [0.5] $\{x := -1\}$ otherwise. We introduce the k-induction operator Ψ_h (c.f. [11]), defined by $\Psi_h(g) := \min\{\overline{\Phi}_f(g), h\}$ for any fixed function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Informally, when applied to a function g, the operator $\Psi_h(g)$ pulls $\overline{\Phi}_f(g)$ down via the pointwise minimum operation with h.

Step 2: Applying k-induction condition. Let $k = 2$. We unroll the loop P $(k = 2)$ times and examine the $(k = 2)$-induction condition to upper-bound the expected value of X_f. The resultant inductive condition from our approach is as follows (here $\leq$ is taken pointwise), which is obtained by applying the operator Ψ_h to a candidate bound function h once (i.e., $k-1$ times):

$$\overline{\Phi}_f(\Psi_h(h)) \leq h \tag{2}$$

We show that under a mild assumption and by using OST, if we have a function h that fulfills this inductive condition, then $\Psi_h(h)$ is an upper bound for the expected value of X_f, for which the *pointwise minimum* in $\Psi_h(h) = \min\{\overline{\Phi}_f(h), h\}$ is the key to derive the piecewise partition of the bound apart from loop unrolling.

Step 3: Simplifying the k-induction condition. Our approach synthesizes a function h w.r.t the condition (2). To the end, we reduce the condition (2) to the form below with four functions h_i $(1 \leq i \leq 4)$ combined with a minimum operation:

$$\min\{h_1, h_2, h_3, h_4\} \leq h, \tag{3}$$

where $h_1 = [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot h(-1,y))$, $h_2 = [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot h(-1,y))$, $h_3 = [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot y)$ and $h_4 = [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot y)$. Using our algorithm, we employ a loop unrolling based approach to efficiently derive the simplified constraint (3) and we show that each h_i results from the unfolding of the loop up to depth $k = 2$ and corresponds to a loop-free program from the unfolding. See **Stage 2** in Section 5 for the details.

Step 4: Solving the simplified $(k = 2)$*-induction condition.* After *Step 3*, we obtain the constraint in (3) and further synthesize the function h in (3) by assuming a template for h and solving the template w.r.t. the constraint (3). Every synthesized function h leads to a piecewise upper bound $\Psi_h(h) = \min\{\overline{\Phi}_f(h), h\}$ for the expected value of X_f. Since this constraint includes a minimum operation, it is non-convex and non-trivial to solve. Our approach reduces the synthesis problem with a linear template to bilinear programming, and obtains a piecewise linear upper bound $[x < 0] \cdot y + [x \geq 0] \cdot (x + y + 2)$, which is actually the exact expected value of y. Similarly, our method can also obtain a piecewise linear lower bound $[x < 0] \cdot y + [x \geq 0] \cdot (x + y + 13/8)$.

4 PIECEWISE BOUNDS VIA LATTICED k-INDUCTION

In this section, we propose a novel combination of OST and latticed k-induction operators to derive bounds for the expected value of X_f. We first introduce expectation functions over which we construct concrete k-induction operators, then define potential functions, and finally show the soundness of potential functions to derive expectation bounds via OST. Throughout this section, we fix a probabilistic while loop $P = \textbf{while}(\varphi)\{C\}$ in the form of (1) and a return function f.

4.1 Expectation Functions

Definition 4.1 (Expectation Functions). An *expectation function* is a function $h : S \to \mathbb{R}$ that assigns to each program state a real value. The partial order $\preceq$ over expectation functions is defined in the pointwise fashion, i.e., $h_1 \preceq h_2 \iff \forall s \in S, h_1(s) \leq h_2(s)$. We denote the set of expectation functions by $\mathcal{E}$ and the lattice by $(\mathcal{E}, \preceq)$, for which the meet operation $\sqcap$ in the lattice is given by $h_1 \sqcap h_2 := \min\{h_1, h_2\}$,where min is the pointwise minimum on functions, i.e., $\forall s \in S, \min\{h_1, h_2\}(s) = \min\{h_1(s), h_2(s)\}$, and the join operation $\sqcup$ is given by $h_1 \sqcup h_2 := \max\{h_1, h_2\}$, where max is the pointwise maximum.

Informally, an expectation function h is that for each program state $s \in S$, the value $h(s)$ bounds the expected value of X_f after the execution of the while loop P when the loop starts with the program state s. Although one observes that the partially ordered set $(\mathcal{E}, \preceq)$ with the meet and join operations defined above is a lattice, we do not use lattice properties in our approach.

To instantiate the k-induction operators for expectation functions, we construct the monotone operator for the lattice $(\mathcal{E}, \preceq)$. To this end, we first define the notion of pre-expectation as follows, wherein $[\varphi]$ denotes the Iverson-bracket of φ. Notice that the random assignment command $x :\approx \mu$ (where μ is a discrete distribution of finite support) can be written in an iterative style of $\{C_1\}\ [p]\ \{C_2\}$, so that we define pre-expectation without random assignment commands.

Definition 4.2 (Pre-expectation [15, 56]). Given an expectation function $h : S \to \mathbb{R}$. We define its *pre-expectation* over a loop-free program Q, $pre_Q(h) : S \to \mathbb{R}$, recursively on the structure of Q:

- $pre_Q(h) := h$, if $Q \equiv \text{skip}$.
- $pre_Q(h) := h[x/e]$, if $Q \equiv x := e$, where $h[x/e]$ denotes $h[x/e](s) = h(s[x/e])$ with $s[x/e](x) = e(s)$ and $s[x/e](y) = s(y)$ for all $y \in \text{Vars} \backslash \{x\}$.
- $pre_Q(h) := pre_{Q_1}(pre_{Q_2}(h))$, if $Q \equiv Q_1; Q_2$.
- $pre_Q(h) := p \cdot pre_{Q_1}(h) + (1 - p) \cdot pre_{Q_2}(h)$, if $Q \equiv \{Q_1\}\ [p]\ \{Q_2\}$.
- $pre_Q(h) := [\phi] \cdot pre_{Q_1}(h) + [\neg\phi] \cdot pre_{Q_2}(h)$, if $Q \equiv \text{if } (\phi)\ \{Q_1\} \text{ else } \{Q_2\}$.

The intuition of pre-expectation is that given an expectation function h, the pre-expectation pre_Q computes the expected value $pre_Q(h)$ of h after the execution of the command Q. With pre-expectation, we then define the monotone operator to be the characteristic function $\overline{\Phi}_f$ of the probabilistic loop P with respect to the return function f as follows.

For the rest of this section, we fix an initial state s^* and override the set S of program states with $Reach(s^*)$ in Definition 4.1 so that we consider expectation functions restricted to $Reach(s^*)$.

Definition 4.3 (Characteristic Function [15, 34]). The *characteristic function* $\overline{\Phi}_f : \mathcal{E} \to \mathcal{E}$ is defined by $\overline{\Phi}_f(h) := [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(h)$. The monotone operator for the lattice $(\mathcal{E}, \preceq)$ is defined as $\overline{\Phi}_f$.

Informally, the characteristic function $\overline{\Phi}_f$ outputs f if the loop guard φ is violated and the loop terminates in the next step, and the pre-expectation of h w.r.t. the loop body C otherwise. It is straightforward to verify the monotonicity of $\overline{\Phi}_f$. In the following, we omit the subscript f in $\overline{\Phi}_f$ if it is clear from the context. Given the monotone operator, we establish the concrete k-induction operators as follows.

Definition 4.4 (k-Induction Operators for $(\mathcal{E}, \preceq)$). Given an expectation function h, the *upper (resp. lower) k-induction operator* $\overline{\Psi}_h : \mathcal{E} \to \mathcal{E}$ (resp. $\overline{\Psi}'_h : \mathcal{E} \to \mathcal{E}$) is defined by $\overline{\Psi}_h(g) = \min\{\overline{\Phi}_f(g), h\}$ (resp. $\overline{\Psi}'_h(g) = \max\{\overline{\Phi}_f(g), h\}$) for arbitrary expectation function $g \in \mathcal{E}$.

Note that k does not explicitly appear within the operators; rather, it denotes the number of times these operators are iteratively applied.

4.2 Potential Functions

We define potential functions as expectation functions satisfying the k-induction conditions. These potential functions serve as candidate bounds to be synthesized.

Definition 4.5 (Potential Functions). Let k be a positive integer. A *k-upper* (resp. *k-lower*) potential function is an expectation function h that satisfies the *upper* (resp. *lower*) k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ (resp. $\overline{\Phi}_f((\overline{\Psi}'_h)^{k-1}(h)) \succeq h$), respectively.

We apply Optional Stopping Theorem (OST) to address our soundness results. We find that the classical OST [24, 58] cannot handle our problem due to the requirement of bounded step-wise difference (see Appendix B.1), while the OST variant proposed in [57] can handle our problem.

Theorem 4.6 (Extended OST [57]). *Let $\{X_n\}_{n=0}^{\infty}$ be a supermartingale adapted to a filtration $\mathcal{F} = \{\mathcal{F}_n\}_{n=0}^{\infty}$ and τ be a stopping time w.r.t the filtration $\mathcal{F}$. Suppose there exist positive real numbers b_1, b_2, c_1, c_2, c_3 such that $c_2 > c_3$ and*

(a) For all sufficiently large natural numbers n, it holds that $\mathbb{P}(\tau > n) \leq c_1 \cdot e^{-c_2 \cdot n}$.
(b) For every natural number $n \geq 0$, it holds almost-surely that $|X_{n+1} - X_n| \leq b_1 \cdot n^{b_2} \cdot e^{c_3 \cdot n}$.

Then we have that $\mathbb{E}(|X_\tau|) < \infty$ and $\mathbb{E}(X_\tau) \leq \mathbb{E}(X_0)$.

Under certain side conditions that guarantee the validity of the extended OST, the potential functions provide upper and lower bounds on the expected value of X_f. Before presenting this result, we introduce some concepts that capture the magnitude of updates to program variables between two consecutive steps.

Definition 4.7 (Termination Time). The *termination time* T of the loop P is the random variable that for any path of the loop, measures the number of total loop iterations in the path.

Definition 4.8 (Uniform Amplifier). Suppose that the loop P is affine, i.e., all conditions and assignments within the loop are affine functions of the program variables. For each program variable x, let x_n denote the random variable representing the value of x at the n-th iteration of the loop. A *uniform amplifier* c is a constant $c > 0$ such that, for all $n \geq 0$, $|x_{n+1}| \leq c \cdot |x_n| + a$ holds for some fixed constant a.

Definition 4.9 (Bounded Update). The loop P has the *bounded-update* property if there exists a real constant $a > 0$ such that for each program variable x, $|x_{n+1} - x_n| \le a$ for every $n \ge 0$ (see Definition 4.8 for the meaning of x_n).

REMARK 2. *Note that any program satisfying the bounded update property also admits a uniform amplifier with $c = 0$.*

We now present the soundness theorem of k-upper (resp. lower) potential functions. We distinguish between *affine programs* and *polynomial programs*, as each requires different side conditions for potential functions to serve as upper or lower bounds. Notably, the side conditions for affine programs are weaker than those for polynomial programs.

THEOREM 4.10. *Suppose the loop P is affine. Let k be a positive integer and h be a polynomial potential function in the program variables with degree d. If there exist real numbers $c_1 > 0$ and $c_2 > c_3 > 0$ such that*

(P1) there exists a uniform amplifier c satisfying $c \le e^{c_3/d}$, and
(P2) the termination time T of P has the concentration property, *i.e., $\mathbb{P}(T > n) \le c_1 \cdot e^{-c_2 \cdot n}$.*

hold, then for any initial program state s^*, we have:

- $\mathbb{E}_{s^*}(X_f) \le \overline{\Psi}_h^{k-1}(h)(s^*) \le h(s^*)$ *holds for any k-upper potential function h.*
- $\mathbb{E}_{s^*}(X_f) \ge (\overline{\Psi}'_h)^{k-1}(h)(s^*) \ge h(s^*)$ *holds for any k-lower potential function h.*

PROOF SKETCH. (See Appendix B.2 for the full proof) Let s_n be the random variable of the program state at the n-th iteration with $s_0 = s^*$, and let $H = \overline{\Psi}_h^{k-1}(h)$. A *key point* is that since H is piecewise polynomial (by the definition of $\overline{\Psi}_h$) and condition (P1) holds, condition (b) in Theorem 4.6 holds for process $\{H(s_n)\}_{n\in\mathbb{N}}$. Combining with the fact that h is a k-upper potential function, one can further deduce $\{H(s_n)\}_{n\in\mathbb{N}}$ is a supermartingale. By applying Theorem 4.6, we have $\mathbb{E}_{s^*}(X_T) \le \mathbb{E}_{s^*}(X_0)$ (T is a stopping time), thus $\mathbb{E}_{s^*}(X_f) \le \mathbb{E}_{s^*}(X_0) = H(s^*)$. The lower case is derived similarly. □

The side condition (P1) for affine programs requires that the loop P possesses a uniform amplifier constant. In contrast, for polynomial programs, a stronger property is needed: the program must satisfy the bounded update property, which imposes stricter constraints than (P1).

THEOREM 4.11. *Let k be a positive integer. Suppose there exist real numbers $c_1 > 0$ and $c_2 > 0$ such that condition (P1') loop P has the bounded update property; and condition (P2) in Theorem 4.10 holds, then for any initial program state s^*, we have*

- $\mathbb{E}_{s^*}(X_f) \le \overline{\Psi}_h^{k-1}(h)(s^*) \le h(s^*)$ *holds for any k-upper potential function h.*
- $\mathbb{E}_{s^*}(X_f) \ge (\overline{\Psi}'_h)^{k-1}(h)(s^*) \ge h(s^*)$ *holds for any k-lower potential function h.*

REMARK 3. *See Appendix B.3 for the proof of Theorem 4.11. The concentration condition (P2), which ensures exponentially decreasing nontermination probabilities as stated in Theorems 4.10 and 4.11, guarantees that loop P terminates almost surely. This condition has been extensively studied in the literature (see, e.g., [16, 17, 26]).* □

According to Theorems 4.10 and 4.11, synthesizing upper and lower bounds reduces to finding a potential function h that satisfies the conditions outlined in these theorems. However, solving the k-upper and k-lower potential conditions is challenging due to the intricate combination of minimum and indicator functions involved. In the following sections, we introduce algorithmic approaches to systematically synthesize these upper and lower bounds.

5 ALGORITHMS FOR BOUND SYNTHESIS

In this section, we first present algorithms for synthesizing upper and lower bounds for single-loop programs. We then demonstrate how our approach naturally extends to handle programs containing nested or sequential loops.

5.1 Algorithms for Probabilistic Single Loops

In this subsection, we present algorithms for synthesizing k-upper and lower potential functions that satisfy the conditions specified in Theorem 4.10 and Theorem 4.11, leading to piecewise bounds on the expected value of X_f. Below, we consider a fixed probabilistic loop P of the form (1) along with a return function f. Due to the space limit, we only illustrate the synthesis procedure for upper bounds. The case for lower bounds is nearly analogous, obtained by replacing minimum with maximum and substituting $\preceq$ by $\succeq$. The pseudocode for our algorithm is presented in Algorithm 1. Our approach consists of the following major steps:

Stage 1: Prerequisites Checking and External Inputs. Our algorithm first verifies the side conditions (P1) and (P2) (respectively, (P1') and (P2')) for affine (respectively, polynomial) programs, as specified by Theorems 4.10 and 4.11. The algorithm also accepts the hyperparameter k and a program invariant as input parameters.

Prerequisites checking. When P is affine, condition (P1) is verified by syntactically inspecting the loop body to identify a positive constant c_3, ensuring that each program variable is amplified by at most $e^{c_3/d}$, up to an additive constant, within a single loop iteration, where d denotes the degree of the polynomial template potential function h (c.f. Stage 2). Condition (P2) is guaranteed either by synthesizing a difference-bounded ranking supermartingale (dbRSM) that demonstrates the exponentially decreasing concentration property [16, 17], or by syntactically analyzing probabilistic branching within the loop to extract a suitable constant c_2 satisfying $c_2 > c_3 > 0$. For polynomial programs, condition (P1')—the bounded update property—is checked via an SMT solver (e.g., Z3 [21]), while condition (P2) is ensured analogously to the affine case.

External inputs. Our algorithm requires the following hyperparameters as input: (1) *Induction parameter k:* We specify a positive real number k as the parameter for k-induction, along with the initial program state s^*. (2) *Program invariant:* We assume the existence of an invariant I at the entry point of the loop, which over-approximates the set of reachable program states $Reach(s^*)$. That is, for every $s \in Reach(s^*)$, we have $s \models I$. The state space is thus restricted to program states satisfying I, and the relation $\preceq$ is interpreted over I, i.e., $h_1 \preceq h_2 \iff \forall s \models I,\ h_1(s) \le h_2(s)$. The rational of this restriction follows from the over-approximation property of I. Invariants can be obtained using external invariant generators, such as [48].

Example 5.1. We take the following example as a running example, which is a discretized version of the Growing Walk in [12]:

$$\texttt{while}\,(\,0 \le x\,)\,\{\{x := x+1; y := y+x\}\ [0.5]\ \{x := -1\}\}$$

In this example, our goal is to analyze the expected value of y upon program termination. We check the prerequisites and specify the external inputs as follows: (1) *Prerequisite Verification*: We find that $c = 1$ serves as a uniform amplifier, satisfying $c \le e^{c_3/d}$ with $c_3 = \ln 1.5$ and $d = 1$. The concentration condition (P2) is also met with $c_2 = \ln 2$. (2) *External Inputs*: We set $k = 2$, and choose the invariant $I = \{x \mid -1 \le x\}$ with initial state $s^* = (x, y) = (1, 1)$. □

Stage 2: Templates and Constraints. After verifying the prerequisites and identifying the external inputs as described in **Stage 1**, our algorithm predefines a d-degree polynomial template h as the candidate k-upper potential function for the loop P. This template consists of a linear combination of all monomials in the program variables of degree at most d, where each monomial is multiplied by an unknown coefficient.

Next, we apply the k-induction conditions from Definition 4.5, resulting in the constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$. The presence of min and indicator operators within this constraint complicates direct simplification. To address this, we reformulate the constraint into the form $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i is free of the minimum operator. Although a brute-force arithmetic expansion can achieve this transformation (see Appendix C.1 for details), our algorithm employs a more efficient unfolding strategy, which we outline below.

The unfolding process for constraint simplification: We symbolically unroll the probabilistic loop from the initial state up to k iterations, exploring all possible unfolding strategies. Here, "symbolic" means that program variables in each program state retain their original variable names and represent undetermined values. An *unfolding strategy* operates at each symbolic program state encountered during the unfolding process (excluding the initial state), and chooses one of three actions: (i) unfold the loop iteration once more, (ii) terminate the unfolding, or (iii) forced to stop when the total number of unfoldings reaches k. Each unfolding strategy, determined by the choices made at each unfolding step, yields a loop-free program. Let $C_1, \ldots, C_m$ denote all loop-free programs generated by applying the above decision process across all possible unfolding strategies. For each loop-free program C_d, we compute the *pre-expectation* $pre_{C_d}(h)$ of h with respect to C_d (see Definition 4.2), allowing us to equivalently rewrite the constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ as:

$$\min\{h_1, h_2, \ldots, h_m\} \preceq h, \tag{4}$$

where each h_i is given by $pre_{C_d}(h)$ for some C_d. According to the computation of pre-expectation (Definition 4.2), each h_i can be represented as $h_i = \sum_r [B_{ir}] \cdot e_{ir}$, where B_{ir} is a predicate independent of the template's unknown coefficients, and e_{ir} is a monolithic polynomial in the program variables, potentially containing unknown coefficients. Moreover, the B_{ir}'s are pairwise logically disjoint.

The following proposition formally establishes the relationship between the unfolding process and the k-induction condition. The proof is provided in Appendix C.2.

Proposition 5.2. *The upper k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ is equivalent to constraint $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i equals $pre_{C_d}(h)$ for some unique $C_d \in \{C_1, \ldots, C_m\}$ from the unfolding process above.*

By Proposition 5.2, the k-induction constraint can be simplified by computing the pre-expectations of all programs $\{C_1, \ldots, C_m\}$ generated by all possible unfolding strategy within k loop iterations. Since these programs are structurally similar, we can efficiently compute $pre_{C_d}(h)$ for all $C_d \in \{C_1, \ldots, C_m\}$ simultaneously by traversing the k-unfolding of the program loop once. This approach reduces runtime by eliminating excessive and repeated computations.

Illustrative Example of the Unfolding Process. We demonstrate our unfolding process via a simple but illustrative example as follows:

$$P := \texttt{while}\,(\,\varphi(x)\,)\,\{\{x := a_1 x + b_1\}\ [p]\ \{x := a_2 x + b_2\}\} \tag{5}$$

where x is a real-valued program variable, $a_i, b_i (i = 1, 2)$ are real constants, $p \in [0, 1]$ and $\varphi(x)$ is a guard condition. Let f be the return function, and let $\overline{\Phi}_f$ be the operator defined as

$$\overline{\Phi}_f(h)(x) := [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1 x + b_1) + (1 - p) \cdot h(a_2 x + b_2))$$

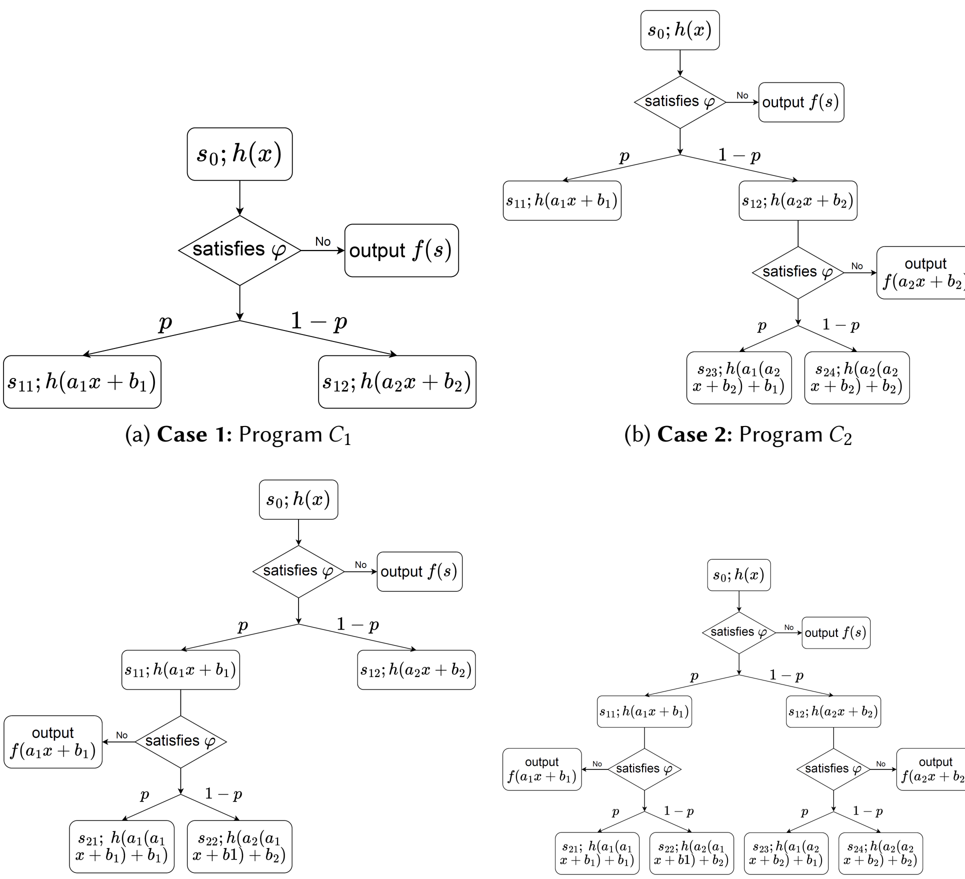
for any function $h : \mathbb{R} \to \mathbb{R}$ (with $S = \mathbb{R}$), where $[\varphi]$ denotes the Iverson bracket for the predicate φ. In this example, we consider the 2-induction operator $\overline{\Psi}_h$ for a fixed function $h : \mathbb{R} \to \mathbb{R}$, as defined in [11]. Specifically, $\overline{\Psi}_h(g)$ is given by $\overline{\Psi}_h(g) := \min\{\overline{\Phi}_f(g), h\}$, and the corresponding 2-upper induction condition is :

$$\overline{\Phi}_f(\overline{\Psi}_h(h)) \leq h\,. \tag{6}$$

According to Proposition 5.2, we simplify this constraint by transforming (6) into the following form, which expresses the minimum over four functions h_i $(1 \leq i \leq 4)$:

$$\min\{h_1, h_2, h_3, h_4\} \preceq h\,,$$

where each h_i corresponds to a loop-free program C_i generated during the unfolding process up to depth $k = 2$. All such unfolded programs are summarized in Fig. 2.



(a) **Case 1:** Program C_1 (b) **Case 2:** Program C_2

(c) **Case 3:** Program C_3 (d) **Case 4:** Program C_4

Fig. 2. Loop-free programs generated by $(k = 2)$-induction

We illustrate the unfolding process as follows. Starting from an initial value x, if $\varphi(x)$ is not satisfied, the loop terminates immediately and outputs $f(x)$. If $\varphi(x)$ holds, we proceed to unfold the loop, resulting in four distinct cases. Due to space constraints, we describe only the first case in detail here; the remaining three cases are depicted in Fig. 2, with further explanations provided in Appendix C.3. In Case 1, the loop executes once and transitions to two possible states, a_1x+b_1 and a_2x+b_2, after which it terminates. This corresponds to a single unrolling of the loop and terminating the unfolding at both resulting symbolic states, yielding the loop-free program C_1 as shown in Fig. 2a. The associated expression is $h_1 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1 x + b_1) + (1-p) \cdot h(a_2 x + b_2))$, which represents the expected value of $h(x)$ after executing program C_1. Cases 2, 3, and 4 are derived analogously by unrolling the loop up to two iterations.

Example 5.3. Returning to the running example in Example 5.1, we establish a 1-degree, i.e., linear template $h = a \cdot x + b \cdot y + c$, where a, b, c are unknown coefficients. We apply 2-induction condition to synthesize a piecewise linear upper bound. Starting from a symbolic initial program state $s^* = (x, y)$, we unroll the loop once and arrive at two new symbolic program states $(x+1, x+y+1)$ and $(-1, y)$. Over each new state, we take the decision separately and the unfolding strategy produces four loop-free programs. The $pre_{C_d}(h)$ w.r.t. these four programs are as follows:

$$\begin{aligned}
h_1 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot h(-1, y)) \\
h_2 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot h(-1, y)) \\
h_3 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.25 \cdot h(-1, y+x) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot y) \\
h_4 &= [x < 0] \cdot y + [x \geq 0] \cdot (0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot y)
\end{aligned} \tag{7}$$

Thus, we have the simplified constraint $\forall (x, y) \models I, \min\{h_1, h_2, h_3, h_4\} \preceq h$. □

Branch reduction. During the unfolding process used to simplify the latticed k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$, the number of resulting functions h_i in (4) grows rapidly with the number of probabilistic choices in the loop body. This combinatorial growth occurs because, when computing the pre-expectation for probabilistic branches, the sum of two minimum expressions results in a new minimum taken over the Cartesian product of the original function sets. To address this issue, we introduce a heuristic that selects only a small subset of "representative" functions from the complete set of h_i in (4). Importantly, this approach does not compromise soundness (see Theorems 4.10 and 4.11), as the minimum over any subset is always at least as the minimum over the full set.

Taking the case of $k = 2$ as an example, by definition of operator $\overline{\Psi}_h$, we have

$$\begin{aligned}
\overline{\Phi}_f(\overline{\Psi}_h(h)) &= \overline{\Phi}_f(\min\{\{\overline{\Phi}_f(h), h\}\} \\
&= [\neg G] \cdot f + [G] \cdot \sum_{i=1}^{n} p_i \cdot \min\{\{\overline{\Phi}_f(h(u_i(s))), h(u_i(s))\}
\end{aligned}$$

where each p_i denotes a probabilistic choice in the characteristic function $\overline{\Phi}_f$, and u_i represents the corresponding state update function under that choice. Instead of enumerating all possible 2^n combinations in choosing either $\overline{\Phi}_f(h(u_i(s)))$ or $h(u_i(s))$ for each p_i (to expand into the minimum form (4)), one could consider combinations that have at most one $\overline{\Phi}_f(h(u_i(s)))$ and at most one $h(u_i(s))$, so that only a linear number of combinations are considered while retaining soundness. For the case of $k > 2$, a possible way for relaxation is to recursively consider combinations that have at most one $\overline{\Phi}_f(\overline{\Psi}_h^{k-2}(h(u_i(s)))$ and at most one $h(u_i(s))$.

Stage 3: Transforming to Canonical Form. At this stage, our algorithm transforms the constraint of the form (4) from **Stage 2** into the following canonical form:

$$[B_1] \implies \min\{e_{11}, \ldots, e_{m1}\} \leq h, \ \ldots, \ [B_l] \implies \min\{e_{1l}, \ldots, e_{ml}\} \leq h \tag{8}$$

where h is the predefined polynomial template. Each $B_j (j \in \{1, ..., l\})$ is a conjunction of predicates over the program variables that does not involve the template's unknown coefficients, and each e_{ij} is a polynomial expression in these unknown coefficients. The transformation begins by rewriting the inequality (4) as

$$\min\{\textstyle\sum_r [B_{1r}] \cdot e_{1r}, \ \ldots, \ \sum_r [B_{mr}] \cdot e_{mr}\} \preceq h \tag{9}$$

where, as described previously, each h_i is expressed as $h_i = \sum_r [B_{ir}] \cdot e_{ir}$. Next, for each conjunction $B = \bigwedge_{i=1}^{m} B_{ir_i}$ – with each B_{ir_i} taken from the summation $\sum_r [B_{ir}] \cdot e_{ir}$ – we obtain the constraint $\Psi_B = [B] \implies \min_{i=1}^{m} e_{ir_i} \leq h$. The transformed system of inequalities (8) is thus precisely the set of all such Ψ_B constraints. Infeasible constraints (i.e., those with unsatisfiable B) are removed, whenever possible, using an SMT solver such as Z3 [21].

Example 5.4. Continuing from Example 5.3, we convert (7) into its canonical form by partitioning the state space S into two regions: $[x < 0]$ and $[x \geq 0]$, as indicated in (7). Applying **Stage 3** and eliminating unsatisfiable predicates yields the following canonical form:

$$[x < 0] \implies \min\{y\} \leq h$$

$$[x \geq 0] \implies \min \begin{Bmatrix} 0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot h(-1, y) \\ 0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot h(-1, y) \\ 0.25 \cdot h(-1, y+x+1) + 0.25 \cdot h(x+2, 2x+y+3) + 0.5 \cdot y \\ 0.5 \cdot h(x+1, x+y+1) + 0.5 \cdot y \end{Bmatrix} \leq h \tag{10}$$

□

Stage 4: Solving Constraints. Below, we describe our approach for solving the canonical constraints given in (8). It is important to note that the presence of the *minimum* operator in this canonical form makes the constraint *non-convex*. To address this, we develop distinct algorithms for the linear and polynomial cases. In the linear case, where the program is affine (i.e., all conditions and assignments are linear), we employ a linear template for the k-upper potential function h. In the polynomial case, where the program may be non-affine, we utilize a polynomial template.

Solving constraints (linear case). In our algorithm for the linear case, we require that the return function be piecewise linear and that the invariant be affine in the program variables. We first eliminate the minimum operator in (8) by considering its negation. This allows us to transform the constraint into a set of bilinear constraints using Motzkin's Transposition Theorem, which can then be solved with off-the-shelf bilinear programming solvers such as *Gurobi*.

Below, we present a variant of Motzkin's Transposition Theorem [43], which will be utilized in the subsequent analysis. The proof is provided in Appendix C.4.

Theorem 5.5 (Motzkin's Transposition Theorem [43]). *Let $S = (A_1 \cdot \mathbf{x} + \mathbf{b_1} \leq 0)$ and $T = (A_2 \cdot \mathbf{x} + \mathbf{b_2} < 0)$ be systems of linear inequalities, where $A_1 = (\alpha_{i,j}) \in \mathbb{R}^{m\times n}$ and $A_2 = (\alpha_{m+i,j}) \in \mathbb{R}^{k\times n}$ are real coefficient matrices, $\mathbf{b_1} = (\beta_1, \ldots \beta_m)^\top$ and $\mathbf{b_2} = (\beta_{m+1}, \ldots \beta_{m+k})^\top$ are real vectors, and $\mathbf{x} = (x_1, \ldots x_n)^\top$. If S is satisfiable, then $S \wedge T$ is unsatisfiable if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_{m+k}$, with at least one λ_i for $i \in \{m+1, \ldots, m+k\}$ being nonzero, such that:*

$$\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)} = 0, \ \ldots, \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)} = 0, \ (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 = 0. \tag{11}$$

Remark 4. *Note that, since $\lambda_i \geq 0$ for $0 \leq i \leq m+k$, the requirement that at least one λ_i for $i \in \{m+1, \ldots, m+k\}$ is nonzero can be equivalently encoded as the linear constraint $\sum_{i=m+1}^{m+k} \lambda_i > 0$.*

In what follows, we demonstrate how to apply Theorem 5.5 to solve the canonical constraints (8). We begin by conjunct the affine invariant I with the antecedent predicates in (8) and eliminating any constraints with unsatisfiable antecedents, resulting in

$$[I \wedge B_j] \implies \min\{e_{1j}, e_{2j}, \ldots, e_{mj}\} \leq h \quad \text{for } j \in \{1, 2, \ldots, l\}, \tag{12}$$

where we assume that each $I \wedge B_j$ is satisfiable. For each j, we have

$$\begin{aligned} & \big([I \wedge B_j] \implies \min\{e_{1j}, e_{2j}, ..., e_{mj}\} \le h\big) \text{ holds} \\ \iff & \big([I \wedge B_j] \wedge (\wedge_{i=1}^{m}(e_{ij} > h))\big) \text{ is not satisfiable} & \text{[Apply Thm 5.5]} \\ \iff & \text{exists nonnegative real vector } \boldsymbol{\lambda_j} = (\lambda_{0,j}, \dots, \lambda_{m_j+k_j,j}), \\ & \text{s.t. } (\lambda_{m_j+1,j}, \dots, \lambda_{m_j+k_j,j}) \ne \mathbf{0}, \text{ and eq. (11) holds.} \end{aligned}$$

The second equivalence follows from the Motzkin's Transposition Theorem by setting $S = I \wedge B_j$ and $T = \big(\wedge_{i=1}^{m}(e_{ij} > h)\big)$ for each $j \in \{1, 2, \dots, l\}$. Note that (11) constitutes a bilinear constraint problem, as its nonlinearity arises solely from the products of unknown template coefficients and the variables $\boldsymbol{\lambda_j}$. Our approach aggregates all such bilinear constraints and utilizes off-the-shelf bilinear solvers to obtain concrete solutions for the template h.

Example 5.6. Continuing from Example 5.4, recall that we choose $x \ge -1$ as the invariant. For the constraint (10), substituting $h(x, y)$ with the template $ax + by + c$ and considering its negation as previously illustrated, we obtain the following inequalities:

$$-x \le 0 \qquad 0.5(a-b)x - 0.5b < 0 \qquad 0.75(a-b)x - (b - 0.25a) < 0$$
$$0.75(a-b)x + 0.5(b-1)y + (0.5c - 0.25a - b) < 0 \qquad 0.5(a-b)x + 0.5(b-1)y + 0.5(c-a-b) < 0.$$

Then by Theorem 5.5, the constraint (10) is equivalent to solving the following set of bilinear constraints involving the unknown coefficients a, b, and c.

$$\begin{aligned} & \exists \lambda_0 \ge 0, \lambda_1 \ge 0, \cdots, \lambda_5 \ge 0 \quad \textbf{s.t.} \quad (\lambda_2 \ne 0 \vee \lambda_3 \ne 0 \vee \lambda_4 \ne 0 \vee \lambda_5 \ne 0) \wedge \\ & 0 = (-1) \cdot \lambda_1 + 0.5(a-b) \cdot \lambda_2 + 0.75(a-b) \cdot \lambda_3 + 0.75(a-b) \cdot \lambda_4 + 0.5(a-b) \cdot \lambda_5 \wedge \\ & 0 = 0.5(b-1) \cdot \lambda_4 + 0.5(b-1) \cdot \lambda_5 \wedge \\ & 0 = -0.5b \cdot \lambda_2 - (b - 0.25a) \cdot \lambda_3 + (0.5c - 0.25a - b) \cdot \lambda_4 + 0.5(c-a-b) \cdot \lambda_5 - \lambda_0. \quad \square \end{aligned}$$

Our algorithm utilizes bilinear solvers to address the derived bilinear constraints. Since these constraints define only a feasible region, we heuristically select an objective function to guide the solver toward solutions that yield tighter upper bounds. Specifically, we minimize $h(s^*)$, where s^* is a designated initial program state of interest. Once the template coefficients for h are determined (yielding a candidate h^*), we reconstruct the piecewise linear upper bound by applying the upper k-induction operator $\overline{\Psi}_{h^*}$ iteratively $k-1$ times, resulting in $\overline{\Psi}_{h^*}^{k-1}(h^*)$. We claim that our linear bound algorithm is complete in the sense that the reduction to bilinear programming preserves the original k-induction condition.

Example 5.7. Continuing with Example 5.6, we use the objective function $h = ax + by + c$ with the initial state $s^* = (x, y) = (1, 1)$. Solving the optimization yields the candidate $h^*(x, y) = x + y + 2$. We then reconstruct the piecewise upper bound by applying $\overline{\Psi}_{h^*}$ once, resulting in the upper bound $[x < 0] \cdot y + [x \ge 0] \cdot (x + y + 2)$.

Solving constraints (polynomial case). In our algorithm for the polynomial case, we assume that the return function is piecewise polynomial and that the invariant is a polynomial predicate over the program variables. We design a sound approach that relaxes the k-induction constraint and reduces the relaxed formulation to a semidefinite programming (SDP) problem using Putinar's Positivstellensatz [46]. This relaxation guarantees that the synthesized upper bound h satisfies the original k-induction condition (see Definition 4.5). The algorithm is described as follows.

First, for each constraint in the canonical form (8), namely $[B_j] \implies \min\{e_{1j}, \dots, e_{mj}\} \le h$ for $j \in \{1, \dots, l\}$, we relax the constraint by replacing the minimum operator with a convex combination of the terms $\{e_{ij}\}_{i=1}^{m}$. This results in the following relaxed form:

$$[B_j] \implies \textstyle\sum_{i=1}^{m} w_i \cdot e_{ij} \le h, \, , j \in \{1, \dots, l\} \tag{13}$$

where each weight $w_i \geq 0$ and the set of weights satisfies $\sum_{i=1}^{m} w_i = 1$. Various forms of weight combinations $\{w_i\}_{i=1}^{m}$ can be employed, such as uniform weights (where each $w_i = 1/m$) or randomly generated weights normalized to sum to one. This relaxation is sound: any function h and set $\{eij\}_{i=1}^{m}$ that satisfy the relaxed constraint (13) will also satisfy the original canonical form (8). This follows from the fact that $\sum_{i=1}^{m} w_i \cdot e_{ij} \leq h \implies \min_{i\in\{1,\ldots,m\}} \{e_{ij}\} \leq h$.

Next, we conjunct the invariant I with each constraint in (13), resulting in the following form:

$$\bigwedge_{j\in\{1,\ldots,l\}} [I \wedge B_j] \implies \sum_{i=1}^{m} w_i \cdot e_{ij} \leq h, \tag{14}$$

We then apply Putinar's Positivstellensatz [46], following previous work [16, 57], to generate constraints on the unknown coefficients, which are solved using off-the-shelf SDP solvers (see Appendix C.5 for details). As these constraints define only a feasible region, we employ a heuristic objective function to guide the solver towards tighter upper bounds. Specifically, we minimize $\sum_i h(s_i^*)$, where s_i^* are selected initial program states of interest. After obtaining the optimal solution h^* from the SDP solver, we reconstruct the piecewise polynomial upper bound $\overline{\Psi}_{h^*}^{k-1}(h^*)$ by iteratively applying the upper k-induction operator $\overline{\Psi}_{h^*}$ to h^* for a total of $k-1$ times.

Algorithm 1: Synthesizing Bounds

Input : Probabilistic loop P in the form of (1) and a return function f
Output : Piecewise bounds for the expected value of X_f upon termination of P

Prerequisites Checking and External Inputs:
(a) Prerequisites Checking: Verify the prerequisites in Theorem 4.10 (Theorem 4.11).
(b) External Inputs: Generate an invariant I, select parameter k and specify initial program state s^*.
Templates and Constraints:
(a) Predefining a (monolithic) polynomial template h.
(b) Unfolding the loop within k times and calculate $pre_{C_d}(h)$ for all $C_d \in \{C_1, \ldots, C_m\}$ (generated by our unfolding process) to obtain the constraint $\min\{h_1, h_2, \ldots, h_m\} \preceq h$.
Transforming to Canonical Form:
Transform the constraints (4) into the form of (8) through an iterative approach and obtain l canonical constraints;
Constraints Solving:
if the loop P is linear and the template h is linear **then**
  $Cons \leftarrow \emptyset$; ▷ Linear Case
  for $j \leftarrow 1$ **to** l **do**
    Extract the coefficients of the variables from canonical-formed constraints;
    Construct bilinear constraints K_j with auxiliary variables $\boldsymbol{\lambda}_j$;
    $Cons \leftarrow Cons \cup K_j$;
  end
  Call bilinear solver to solve $Cons$ and obtain the piecewise bound with the solution h^*
else
  (a) Soundly relax the original canonical constraints (8) into (14). ▷ Polynomial Case
  (b) Call SDP solver to solve and obtain the piecewise bound with h^*.
end

Correctness. Our algorithms are guaranteed to produce correct bounds by Theorems 4.10 and 4.11. The *Prerequisites Checking* stage ensures that all prerequisites in Theorem 4.10 and Theorem 4.11 are met, and the function h is determined according to the k-induction conditions (see Definition 4.5).

Additionally, the invariants we use over-approximate the set of reachable program states, thereby preserving the soundness of our approach. Specifically, our linear bound algorithm is both sound and complete in the sense that the reduction to bilinear programming exactly preserves the original k-induction condition. In the polynomial case, our algorithm employs a sound relaxation, which likewise guarantees the correctness of the synthesized bounds.

5.2 Extensions: Handling Probabilistic Programs with Multiple Loops

Below, we describe the extension of our approach to probabilistic programs with multiple loops, including both sequential compositions of probabilistic loops and nested loops. For brevity, we focus on the synthesis of upper bounds; the synthesis of lower bounds is entirely analogous.

Sequential Composition. For a sequential composition $P = P_1; \dots ; P_n$ of probabilistic loops $P_1, \dots , P_n$ with a return function f, our method analyzes each loop component in reverse order. To illustrate the approach, we focus on the case $P = P_1; P_2$. Given a k-induction parameter k, the procedure for synthesizing upper bounds proceeds as follows:

- Begin by computing a piecewise upper bound h_2 for the expected value of f after the execution of loop P_2.
- Then, treat h_2 as the return function for P_1 and compute its piecewise upper bound, resulting in the final bound h_1 for the entire composition.

This backward compositional reasoning can be systematically extended to compositions with more than two loops.

Nested Loops. To address nested loops, we incorporate our approach with the methods proposed in [26, 27], applying k-induction exclusively to the innermost loop and 1-induction to the outer loops. Since the innermost loop can be unfolded independently of the outer loops, we are able to derive tight piecewise bounds for the inner loop via k-induction and subsequently propagate these bounds to the outer loops. For clarity, we focus on the case where the program R contains a single inner loop and has the following structure:

$$R = \texttt{while}(\psi)\{P\} \text{ with } P = \texttt{while}(\varphi)\{Q\} \text{ and } Q \text{ loop-free.}$$

Our objective is to analyze the expected value of X_f upon termination of the loop. Let Φ_f^{out} denote the characteristic function (see Definition 4.3) with respect to the outer loop and return function f, and let Φ_g^{in} denote the characteristic function for the inner loop P and return function g. While Φ_g^{in} can be computed explicitly, Φ_f^{out} typically cannot. We therefore apply 1-induction to the outer loop and k-induction to the inner loop, as summarized below:

- Define a template h_{out} at the entry of the outer loop and a template h_{in} at the entry of the inner loop.
- For the outer loop, the 1-induction rule yields the constraint $\Phi_f^{out}(h_{out}) \preceq h_{out}$. Since $\Phi_f^{out}(h_{out})$ cannot generally be computed explicitly, we upper-approximate the expected value of h_{out} after executing the inner loop P by h_{in}, i.e., $\Phi_f^{out}(h_{out}) \preceq [\neg\psi] \cdot f + [\psi] \cdot h_{in}$, and the original constraint $\Phi_f^{out}(h_{out}) \preceq h_{out}$ can be strengthened into $[\neg\psi] \cdot f + [\psi] \cdot h_{in} \preceq h_{out}$.
- For the inner loop, we apply the k-induction condition (see Definition 4.4) to ensure that h_{in} upper-approximates the expected value of h_{out} after executing the inner loop. This leads to the constraint $\Phi_{h_{out}}^{in}((\Psi_{h_{in}}^{in})^{k-1}(h_{in})) \preceq h_{in}$, where $\Psi_{h_{in}}^{in}(g) = \min\{\Phi_{h_{out}}^{in}(g), h_{in}\}$ is the upper k-induction operator for the inner loop P (see Definition 2.1).
- Collect the resulting constraints and apply our synthesis algorithm as described in Section 5.

Through this process, we obtain h_{out} as a piecewise upper bound for the expected value of X_f with respect to the return function f upon termination of the entire while loop R.

6 EXPERIMENTAL RESULTS

We implement our algorithms[1] in Python 3.9.12 and Julia 1.9.4. We use Gurobi in Python for bilinear programming and Mosek in Julia for semi-definite programming. All experiments are conducted on a Windows 10 (64-bit) machine equipped with an Intel(R) Core(TM) i7-9750H CPU at 2.60GHz and 16GB of RAM. We evaluate our algorithms for synthesizing piecewise linear and polynomial upper bounds, as detailed in Section 6.1 and Section 6.2. Results for lower bound synthesis, which exhibit similar performance and comparative advantages, are provided in Appendix D.2 and Appendix D.3 due to space limitations.

Evaluation Goals. Our experiments are designed to address the following research questions:

- **RQ1.** How effective is our approach in generating piecewise bounds?
- **RQ2.** How does our approach compare to the most closely related methods?
- **RQ3.** How do our piecewise bounds compare to monolithic polynomial bounds?

Experimental Settings. We address the evaluation goals for our piecewise linear and polynomial algorithms separately. The experiments are conducted under the following settings:

Invariants. We employ invariants to over-approximate the set of reachable states, which is standard in various existing results [15, 16, 28]. Note that invariants do not provide information about the piecewise partitioning of the bounds to be computed. In our experiment, we minimize their impact by deliberately choosing trivial interval-bound invariants that can be directly derived as the union of loop guard and its post image under the increment/decrement operations within the loop body.

Prerequisites Checking. Our experiments cover both linear and polynomial probabilistic programs (see Appendix E for details). For linear programs with monolithic linear return functions, we use a linear template and apply our linear algorithm. For more general cases involving polynomial programs with piecewise polynomial return functions, we employ a higher-degree polynomial template and apply our polynomial algorithm. In our piecewise linear experiments, we ensure that the prerequisites (P1) and (P2) in Theorem 4.10 are satisfied as follows. For (P1), we verify syntactically that the uniform amplifier c can typically be set to 1 across most benchmarks, ensuring that (P1) holds for any positive c_2. For the remaining benchmarks, we take the maximum coefficient of the program variables in the loop body as c. For example, in the St-Petersburg benchmark, we set the uniform amplifier c to 2, choosing $c_3 = \ln 2$ (since $e^{c_3} = 2$) and $c_2 = \ln 4$ to meet the required conditions. For (P2), in benchmarks where each loop iteration terminates with probability p and continues with probability $1 - p$, we can syntactically extract p and verify that the concentration property holds, exhibiting exponential decay at a rate of $e^{\ln(1-p)}$. For the remaining benchmarks, we construct difference-bounded ranking supermartingales (dbRSMs) to ensure the concentration property. Such dbRSMs can be synthesized automatically using methods described in [16, 17]. In our piecewise polynomial experiments, we ensure that the prerequisites (P1') and (P2) in Theorem 4.11 are satisfied as follows. For (P1'), we verify the bounded-update property on each polynomial benchmark using an SMT solver [21]. For (P2), we apply the same approach as in the linear case to establish the concentration property for polynomial programs.

Bound Optimization. Recall that in our algorithms described on pages 14 and 15, we optimize the synthesized upper bounds by minimizing their values over the initial states of interest, which serve as the objective function. In the piecewise linear experiments, we typically set the default initial state s^* by assigning the value 1 to all program variables across most benchmarks. For specific cases, such as Fair Coin, we assign initial values $x = 0$ and $y = 0$ — since $(x, y) = (0, 0)$ is the only state from which the loop can be entered — and set the variable i to its default value of 1. In the piecewise polynomial experiments, for path probability estimation benchmarks selected from [13, 29, 47, 57],

[1] https://anonymous.4open.science/r/text1-B83C-popl/

we adopt the default initial state s^* used in previous work to ensure consistency. For the remaining benchmarks, we first define an interval-bound region, with real-valued variables ranging over $[0, 10]$ and Boolean variables over $[0, 1]$. We then select 10 initial states comprising the boundary points of the region, the midpoints of each boundary, the center point, and uniformly distributed integer points within the region.

Weights Selection. For the polynomial experiments, recall that our algorithm requires a predefined set of weight combinations (see Eq. (14)). We employ uniformly distributed weights (i.e., each weight is $\frac{1}{m}$) and additionally generate 10 sets of randomly selected weights, each normalized to sum to one. Independent computations are performed for each of these 11 weight combinations. From the resulting solutions, we select the function with the minimum objective value as the synthesized upper bound h^*. The total execution time is reported as the cumulative runtime for the 11 independent runs with different weight settings.

Numerical Repair. To address the inherent numerical issues associated with numerical solvers, we apply a post-processing step to repair the computed results. In the linear experiments, we approximate the output floating-point coefficients with rational numbers using continued fractions (see Appendix D.1, [33]), and validate these approximations by checking the constraints in (8). This numerical repair is applied to all benchmarks except Expected Time. For this particular benchmark, since suitable rational approximations could not be found, we truncate the floating-point results to a precision of 10^{-4} and verify their validity against the same constraints. In the polynomial experiments, we similarly truncate all floating-point coefficients to 10^{-4} precision, then substitute the results into the constraints in (14) to check feasibility. Of the 20 benchmarks evaluated, the results for 16 passed our validation procedure, while the remainder remain unknown.

6.1 Piecewise Linear Bound Synthesis

Benchmark Selection. We choose upper-bound benchmarks from existing works [5, 10–12, 20, 26, 27, 29] that fall into our scope and have the following adaptions. First, for those that do not have linear return functions, we add simple linear return functions. Second, for those whose upper bound that can be handled directly by 1-induction (except for several classical examples: k-Geo, RevBin, Fair Coin), we adapt them by reasonable perturbations (such as changing the assignment statement, changing the probability parameters, reducing the continuous distribution to discrete distribution, etc) so that they require $(k > 1)$-induction. Third, for those whose upper bound that cannot be handled by k-induction with small $k = 1, 2, 3$, we adapt them by reasonable perturbations as above so that they can be handled by $(k > 1)$-induction, while still cannot be handled by 1-induction.

In detail, we consider 7 original examples and 6 adapted examples from the literature. The examples Geo, k-Geo and Equal-Prob-Grid are taken from [10, 11], for which we replace the assertion probability with a linear return function *goal* in Equal-Prob-Grid. We consider the benchmark Zero-Conf-Variant adapted from [10, 26]. We revise the assignments and probabilistic parameters in the original program, and add a linear return function *curprobe*. The benchmark St-Petersburg variant is taken from [26] where we replace the probability parameter $\frac{1}{2}$ with $\frac{3}{4}$ since the original program does not satisfy the prerequisites in Theorem 4.10. From [5, 20, 27], we consider the benchmarks Coin, Mart, RevBin and Fair Coin, and revise the assignments, guards on the original benchmarks Bin series so that we obtain a more complex version Bin-Ran. The remaining three examples, Expected Time, Growing Walk and its variant, are all adapted from [12, 29] by reducing the continuous distributions to discrete distributions.

Answering RQ1. We present the experimental results on these 13 benchmarks in Table 1. As bilinear solving is an iterative search for optimal solutions, we set the maximum searching time for

Table 1. Experimental Results for **RQ1** and **RQ2**, Linear Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "T(s)" (of our approach) stands for the execution time of our approach (in seconds), including the parsing from the program input, transforming the k-induction constraint into the bilinear problems, bilinear solving time and verification time. "Conventional Approach ($k = 1$)" stands for the monolithic linear upper bound synthesized via 1-induction, "k" stands for the k-induction we apply, "Solution" stands for the linear candidate solved by Gurobi, and "Piecewise Linear Upper Bound" stands for our piecewise results. "Result" stands for the synthesized results by other tools and "T(s)" (of their approaches) stands for the execution time of their tools.

Benchmark	f	Conventional Approach ($k = 1$)	Our Approach				CEGISPRO2		EXIST	
			k	Solution	Piecewise Linear Upper Bound	T(s)	Result	T(s)	Result	T(s)
GEO	x	✗	3	$x+1$	$[c>0]\cdot x+[c\le 0]\cdot(x+1)$	1.92	$[c>0]\cdot x+$ $[c\le 0]\cdot(x+1)$	0.05	$x+[c=0]$	17.29
K-GEO	y	$-k+N+$ $x+y+1$	3	$-k+N$ $+x+y+1$	$[k>N]\cdot y+[k\le N-1]\cdot$ $(-k+N+x+y+1)+$ $[N-1<k\le N]\cdot$ $(-0.5k+0.5N+x+y+1)$	132.76	$[k>N]\cdot y+[k\le N]$ $\cdot(-k+N+x+y+1)$	0.38	$y+[k\le n]\cdot$ $(x-k+n+1)$	76.74
BIN-RAN	y	✗	2	$0.9x-21i$ $+y+233$	$[i>10]\cdot y+$ $[\frac{90}{11}<i\le 10]\cdot(0.9x-21i+y+233)$ $+[i\le\frac{90}{11}]\cdot(0.9x-18.8i+y+215)$	106.29	inconsistent results	-	inner error	-
COIN	i	✗	2	$i+\frac{8}{3}$	$[x\ne y]\cdot i+$ $[x=y]\cdot(i+\frac{8}{3})$	104.13	not terminate	-	fail	-
MART	i	✗	3	$i+2$	$[x\le 0]\cdot i+[x>0]\cdot(i+2)$	19.29	violation of non-negativity	-	$i+[x>0]*2$	37.23
GROWINGWALK	y	✗	3	$x+y+2$	$[x<0]\cdot y+[x\ge 0]\cdot(x+y+2)$	4.03	violation of non-negativity	-	$y+[x\ge 0]\cdot$ $(x+2)$	21.98
GROWINGWALK -VARIANT	y	$x+y+1$	3	$x+y+1$	$[x<0]\cdot y+$ $[0\le x<1]\cdot(0.5x+y+0.25)$ $+[x\ge 1]\cdot(x+y)$	125.19	violation of non-negativity	-	not terminate	-
EXPECTED TIME	t	✗	3	$4.4280x+t$ $+6.2461$	$[x<0]\cdot t+[0\le x<1]\cdot(t+1)+$ $[1\le x<3.258]\cdot(3.9852x+t+7.39)$ $+[3.258\le x<3.3772]\cdot$ $(4.4280x+t+6.2461)+$ $[3.3772\le x]\cdot(3.5867x+t+9.0874)$	109.35	violation of non-negativity	-	not terminate	-
ZERO-CONF -VARIANT	cur	✗	3	$\mathit{cur}+140$	$[\mathit{est}>0]\cdot\mathit{cur}+$ $[\mathit{start}=0\wedge\mathit{est}\le 0]\cdot(\mathit{cur}+140)$ $+[\mathit{start}\ge 1\wedge\mathit{est}\le 0]\cdot(\mathit{cur}+42)$	180.42	violation of non-negativity	-	$\mathit{cur}+[\mathit{est}=0]$ $\cdot(-49\cdot\mathit{start}^2-$ $49\cdot\mathit{start}+141)$	392.19
EQUAL-PROB-GRID	goal	✗	2	$\mathit{goal}+1.5$	$[a>10\vee b>10\vee\mathit{goal}\ne 0]\cdot\mathit{goal}$ $+[a\le 10\wedge b\le 10\wedge\mathit{goal}=0]\cdot 1.5$	142.68	$[a>10\vee b>10\vee$ $\mathit{goal}\ne 0]\cdot\mathit{goal}+$ $[a\le 10\wedge b\le 10$ $\wedge\mathit{goal}=0]\cdot 1.5$	0.11	fail	-
REVBIN	z	$2x+z$	3	$2x+z$	$[x<1]\cdot z+[1\le x<2]\cdot(z+x+1)$ $+[x\ge 2]\cdot(z+2x)$	70.30	$[x<1]\cdot z+$ $[x\ge 1]\cdot(z+2x)$	0.22	$z+[x>0]\cdot 2x$	151.26
FAIR COIN	i	$i-2y+2$	3	$i+\frac{4}{3}$	$[x>0\vee y>0]\cdot i+$ $[x\le 0\wedge y\le 0]\cdot(i+\frac{4}{3})$	129.34	$[x>0\vee y>0]\cdot i+$ $[x\le 0\wedge y\le 0]\cdot$ $(i+\frac{4}{3})$	0.06	$[x+y=0]\cdot$ $\frac{4}{3}+i$	17.95
ST-PETERSBURG VARIANT	y	✗	3	$\frac{3}{2}y$	$[x>0]\cdot y+[x\le 0]\cdot\frac{3}{2}y$	1.53	$[x>0]\cdot y+$ $[x\le 0]\cdot\frac{3}{2}y$	0.04	$y+[x=0]\cdot$ $0.5y$	13.39

Gurobi to 100s. On most benchmarks, we find that a monolithic linear bound with 1-induction does not exist but obtain a piecewise linear upper bound via ($k > 1$)-induction in a few minutes. Our approach derives the exact bound, i.e., the tightest upper bound, on the benchmarks GEO, COIN, K-GEO, MART, GROWING WALK, EQUAL-PROB-GRID, REVBIN, FAIR COIN, ST-PETERSBURG VARIANT. The exactness of these bounds is established by comparison with the exact invariants synthesized in [5] (see **RQ2**) and with the piecewise lower bounds presented in Appendix D.2. We also show that on a significant number of benchmarks (e.g., K-GEO, BIN-RAN, GROWING WALK-VARIANT, EXPECTED TIME, etc), the piecewise bounds we synthesize are non-trivial (i.e., the program state space S is partitioned into more than $[\varphi]$ and $[\neg\varphi]$).

Answering RQ2. We answer **RQ2** by comparing our approach with the most related approaches [5, 10]. We present our comparison results in Table 1. The main difference between CEGISPRO2 [10] and our approach is that CEGISPRO2 requires an upper bound to be verified as an additional program input and it will only return a super-invariant (i.e., a possibly piecewise upper-bound) that is sufficient to *verify* (i.e., smaller than) the input upper bound, while we intend to synthesize a tight piecewise upper bound directly. The benchmarks Geo, k-Geo are the common benchmarks in these two works and the direct comparisons are as follows: For the benchmark Geo, the piecewise upper bounds of the two methods are the same. For k-Geo, their piecewise result is consistent with our result over $\mathbb{Z}_{\geq 0}$. While in the scope of real numbers, our piecewise upper bound is tighter than theirs. To have a richer comparison with cegispro2, we give cegispro2 an advantage by feeding our benchmarks (including the above two benchmarks) in Table 1 to cegispro2 paired with the piecewise upper bounds synthesized by our approach. We find cegispro2 cannot adequately handle piecewise inputs. Additionally, it reports violation of non-negativity on 5 of our benchmarks (see Table 1). By feeding one segment from the piecewise bounds synthesized via our approaches for the remaining 8 benchmarks, we find on 6 benchmarks, cegispro2 produce the consistent results with our inputs on $\mathbb{Z}_{\geq 0}$, while some of them (e.g., k-Geo) are incorrect over $\mathbb{R}$. On Bin-Ran, the results they produce are impossible to compare since it produces sophisticated and different results when we feed different segments from our piecewise upper bound. On Coin, the execution using their tool does not terminate, which prevents the output of a result.

The work [5] considers the probabilistic invariant synthesis via data-driven approach. Note that the synthesis of upper bounds (i.e., super-invariants) is not considered in their work, and the only relevant work in [5] with our upper bound synthesis is the exact invariant synthesis. For a further comparison, We apply their tool exist on our benchmarks to try to generate exact invariants. On the benchmarks Geo, k-Geo, Mart, Growing Walk, RevBin, Fair Coin, St-Petersburg variant, exist can generate an exact invariant for each benchmark and we show that on these benchmarks, the piecewise upper bounds we synthesize are equal to their exact invariants so that the upper bounds we synthesize are actually the exact expected value of X_f . On the benchmark Zero-Conf-Variant, they spend about 400s while we obtain a respectable piecewise linear bound in around 180s. For the remaining benchmarks, their tool fails or the computation seems to be stuck.

In conclusion, our approach can handle many benchmarks that these two works [5, 10] cannot handle. When feeding our benchmarks with the bounds synthesized through our approach to cegispro2 and exist, they fail on about 40% of our benchmarks. Over most of the benchmarks that their and our approaches can handle, our bounds are comparable with theirs.

Answering RQ3. In addition **RQ2**, we compare our piecewise linear upper bounds with monolithic polynomial bounds via 1-induction in Table 2. Following [16, 57], we implement the polynomial synthesis with Putinar's Positivstellensatz [46] (see Appendix C.5). For a fair comparison, we generate the polynomial bounds with the same invariant and optimal objective function for each benchmark. All the numerical results in the polynomial bounds are cut to 10^{-4} precision. We compare two results by uniformly taking the grid points in the invariant and evaluate two results, and we compute the percentage of the points that our piecewise upper bound are larger (i.e., not better) than monolithic polynomial, which is shown in the last column "PCT" in Table 2. We show that on most of our benchmarks, our piecewise linear bounds are significantly tighter than monolithic polynomial bounds.

6.2 Piecewise Polynomial Bound Synthesis

Benchmark Selection. We select all remaining benchmarks from [5, 10] that are not used in the previous linear experiments, as well as path probability estimation benchmarks from [13, 29, 47, 57], including all unbounded loop benchmarks from [47] in particular. For the former 7 benchmarks from [5], we instantiate the probability parameters with commonly used values (such as 0.5). Note that among them, the benchmarks GeoAr, Bin0, Bin2, Sum0, Duel cannot be handled by our piecewise linear algorithm with k-induction when $k = 1, 2, 3$, even though both the program and the return function are linear. For the benchmarks from [10], the benchmarks chain, brp exhibit

Table 2. Experimental Results for **RQ3**, Linear Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "k" stands for the k-induction condition we apply in this comparison, "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, and "d" stands for the degree of polynomial template we use, "PCT" stands for the percentage of the points that our piecewise upper bound are larger (i.e., not better) than monolithic polynomial.

Benchmark	f	**Our Approach**		**Monolithic Polynomial via 1-Induction**		**PCT**
		k	**Piecewise Linear Upper Bound**	**d**	**Monolithic Polynomial Upper Bound**	
Geo	x	3	$[c > 0] \cdot x + [c \leq 0] \cdot (x+1)$	3	$1.0000 - 1.9996 * c + 1.0000 * x + 0.9996 * c^2 - 0.0002 * x * c + 0.0002 * x * c^2$	0.0%
k-Geo	y	3	$[k > N] \cdot y + [k \leq N-1] \cdot (-k+N+x+y+1) + [N-1 < k \leq N] \cdot (-0.5k+0.5N+x+y+1)$	2	$274.1142 - 53.62281 * N - 1.0000 * k + 1.0000 * y + 1.0000 * x + 2.7311 * N^2$	2.82%
Bin-ran	y	2	$[i > 10] \cdot y + [\frac{90}{11} < i \leq 10] \cdot (0.9x - 21i + y + 233) + [i \leq \frac{90}{11}] \cdot (0.9x - 18.8i + y + 215)$	3	$66.8036 + 21.0161 * i - 29.5267 * y - 17.6524 * x - 1.5735 * i^2 - 0.2059 * y * i - 0.0157 * y^2 - 0.4056 * x * i - 0.2380 * x * y - 1.7910 * x^2 - 0.0102 * i^3 + 0.2917 * y * i^2 + 0.0103 * y^2 * i - 0.0045 * y^3 + 0.4251 * x * i^2 - 0.0036 * x * y * i - 0.0095 * x * y^2 + 0.6938 * x^2 * i - 0.03827 * x^2 * y + 0.6886 * x^3$	49.59%
Coin	i	3	$[x \neq y] \cdot i + [x = y] \cdot (i + \frac{8}{3})$	2	$2.6667 + 1.0000 * i - 0.6381 * y + 4.2840 * x - 2.0286 * y^2 - 2.0067 * x * y + 0.3893 * x^2$	0.0%
Mart	i	3	$[x \leq 0] \cdot i + [x > 0] \cdot (i+2)$	2	$0.0248 + 1.0000 * i + 199999.6588 * x + 0.1643 * x^2$	0.0%
Growing Walk	y	3	$[x < 0] \cdot y + [x \geq 0] \cdot (x+y+2)$	3	$2.5000 + 1.0000 * y + 1.900 * x - 0.5000 * x^2 + 0.1000 * x^3$	0.0%
GrowingWalk variant	y	3	$[x < 0] \cdot y + [0 \leq x < 1] \cdot (0.5x + y + 0.25) + [x \geq 1] \cdot (x+y)$	3	$1.0000 * y - 0.2380 * x + 0.1041 * y^2 - 0.0686 * x * y + 0.0951 * x^2 + 0.03558 * x * y^2 + 0.0686 * x^2 * y + 0.1430 * x^3$	5.52%
Expected Time	t	3	$[x < 0] \cdot t + [0 \leq x < 1] \cdot (t+1) + [1 \leq x < 3.258] \cdot (3.9852x + t + 7.39) + [3.258 \leq x < 3.3772] \cdot (4.4280x + t + 6.2461) + [3.3772 \leq x] \cdot (3.5867x + t + 9.0874)$	3	$3.1203 + 0.9622 * t + 2.8278 * x + 0.0015 * t^2 - 0.01558 * x * t - 0.1397 * x^2 - 0.0003 * x * t^2 - 0.0002 * x^2 * t + 0.0025 * x^3$	50.0%
Zero-Conf -variant	cur	3	$[est > 0] \cdot cur + [start == 0 \land est \leq 0] \cdot (cur + 140) + [start \geq 1 \land est \leq 0] \cdot (cur + 42)$	2	$109.8660 - 0.1357 * cur + 293795.0410 * start + 209178.7117 * est + 0.0019 * cur^2 + 0.7202 * start * cur - 293865.0570 * start^2 + 1.0313 * est * cur + 274251.8886 * est * start - 209283.0750 * est^2$	0.5 %
Equal-Prob-Grid	$goal$	2	$[a > 10 \lor b > 10 \lor goal \neq 0] \cdot goal + [a \leq 10 \land b \leq 10 \land goal = 0] \cdot 1.5$	2	$1.6661 + 5.7396 * goal - 9.4857 * 10^{-5} * b + 1.5707 * 10^{-5} * a + 0.6003 * goal^2 - 0.6740 * b * goal + 1.5975^1 0{-}5 * b^2 + 2.2074 * 10^{-5} * a * goal$	0.0%
RevBin	z	3	$[x < 1] \cdot z + [1 \leq x < 2] \cdot (z + x + 1) + [x \geq 2] \cdot (z + 2x)$	2	$1.0000 * z + 2.0000 * x$	0.0%
Fair Coin	i	3	$[x > 0 \lor y > 0] \cdot i + [x \leq 0 \land y \leq 0] \cdot (i + \frac{4}{3})$	2	$1.3333 + 1.0000 * i - 0.4141 * y - 0.4141 * x + 1.1743 * i^2 - 2.3486 * y * i + 0.2551 * y^2 - 2.3486 * x * i + 3.6820 * x * y + 0.2551 * x^2$	0.0%
St-Petersburg variant	y	3	$[x > 0] \cdot y + [x \leq 0] \cdot \frac{3}{2} y$	3	$0.0197 + 1.5047 * y + 371727.7656 * x - 0.5028 * x * y - 371727.7734 * x^2$	0.0%

numerical pathologies due to extremely large constants, which can cause numerical instability and render our algorithms ineffective. To address this issue, we scale down these pathological values to more moderate magnitudes—for instance, replacing 1000000000000 with 100 in the chain benchmark and 8000000000 with 800 in the brp benchmark—so that our numerical algorithm can operate reliably. For the benchmarks from [13, 29, 47, 57], since 5 of 9 benchmarks contain continuous distributions originally, we make simple adaptions on these benchmarks by replacing each continuous distribution (e.g. uniform distribution over $[0, 1]$) with a uniform discrete choice of the same range (e.g. 0 with probability 0.5 and 1 also with 0.5), resulting in 5 adapted benchmarks. The benchmark inv-Pend in [47] does not pass our checking of prerequisite (P2). Therefore we make minor modifications to the coefficients in this benchmark so that we can synthesize a dbRSM to satisfy (P2), thereby obtaining the benchmark inv-Pend variant. We apply 2-induction on these 24 benchmarks.

Answering RQ1. Our algorithm successfully handles all of the aforementioned benchmarks except for four. The failures in these cases are attributed to excessive branching introduced by our algorithm based on Proposition 5.2 (see **Stage 2** in Section 5), and branch reduction techniques (see Page 12) have not yet been incorporated into our implementation. Nevertheless, our current implementation is capable of addressing a wide range of complex benchmarks. For example, the benchmark CAV5 comprises 35 lines of code (see Appendix E), the benchmark INV-PEND VARIANT benchmark features 4 variables with complex polynomial updates, posing significant challenges for analysis. We leave further optimization for future work. We present the experimental results for the synthesis of piecewise polynomial upper bounds on the remaining 20 benchmarks in Table 3. Our approach successfully derives piecewise polynomial upper bounds for 16 out of 20 benchmarks within seconds. Of the remaining four, two benchmarks (FIG-6 and FIG-7) are solved within tens of seconds, while only INV-PEND VARIANT and CAV-5 require more than five minutes to compute a result. Our algorithms obtain the exact bound (i.e., the tightest upper bound) on the benchmarks BIN0, BIN2, DEPRV, PRINSYS, SUM0. The exactness of these results is verified by comparison with the exact invariants synthesized in [5] (see **RQ2**) and with our corresponding lower bounds in Appendix D.3.

Answering RQ2. We answer **RQ2** by comparing our approach with the relevant work EXIST [5] in Table 3, whose illustration is the same to Table 1. It is worth noting that CEGISPRO2 only supports linear bounds and does not accept nonlinear expressions as additional program input. Therefore, we exclude it from our comparison. Note that the only relevant aspect of [5] with respect to upper bound synthesis (i.e., super-invariants) is their method for exact invariant synthesis. For comparison, we apply their tool EXIST to our benchmarks in an attempt to generate exact invariants. On the benchmarks BIN0, BIN2, PRINSYS, and SUM0, we show that the piecewise polynomial upper bounds we synthesize are actually the exact expected value of X_f, i.e., the tightest upper bounds, by comparing them with the exact invariants synthesized by EXIST. Among these benchmarks, EXIST spends about 80s on BIN0, about 250s on BIN2, and about 100s on SUM0, while we spend only several seconds to obtain the specific results. Thus, our algorithm is much more efficient. For the benchmarks DUEL, CHAIN, and CAV2, the tool EXIST is able to identify candidates for exact invariants but fails to verify them, and thus does not produce exact invariants. Additionally, EXIST does not support the benchmarks GRID SMALL and GRID BIG. For the remaining benchmarks, EXIST fails to generate results due to internal errors. Moreover, for the benchmark DEPRV, we demonstrate that the piecewise polynomial upper bound synthesized by our approach is exact, as it coincides with the corresponding lower bound (see Appendix D.3). Thus, our method yields the tightest upper bound for 5 out of the 20 benchmarks in this table. In summary, our approach successfully handles more benchmarks than [5], and for those benchmarks that both methods can process, our approach is more efficient and produces comparable bounds.

Answering RQ3. In addition to the comparisons in **RQ2**, we further evaluate our piecewise polynomial upper bounds (obtained via k-induction) against monolithic polynomial bounds of higher degree synthesized using simple induction (i.e., 1-induction). The synthesis of these monolithic polynomial bounds is implemented using Putinar's Positivstellensatz [46] (see Appendix C.5 for details). For a fair comparison, we use the same invariant and optimal objective function for each benchmark. We also verify the validity of the monolithic polynomial bounds (see *Numerical Repair*). In our experimental evaluation, we observe that for most benchmarks, when the degree of the polynomial template exceeds 5, numerical performance deteriorates and the synthesized monolithic bounds fail our validation process. Therefore, in this experiment, we restrict the degree of monolithic polynomial bounds to at most 5.

Table 3. Experimental Results for **RQ1** and **RQ2**, Polynomial Case (Upper Bounds). "f" stands for the return function considered in the benchmark, "T(s)" stands for the execution time of our approach (in seconds), including the parsing procedure from the program input, relaxing the k-induction constraint into the SDP problems, the SDP solving time and verification time. "d" stands for the degree of polynomial template we use and "Solution h^*" is the candidate polynomial solved directly by the solver. "Piecewise Polynomial upper Bound" stands for the piecewise bound we synthesize. "Exact" stands for the exact expected value synthesized by EXIST.

Benchmark	f	**Our Approach**				**EXIST**	
		d	**Solution h^***	**T(s)**	**Piecewise Polynomial Upper Bound**	**Exact**	**T(s)**
GeoAr	x	2	$0.0001 * x^2 - 0.0004 * x * y + 0.0005 * x * z + 0.0011 * y^2 + 0.0079 * z^2 + 0.9998 * x + 1.5398 * y - 0.0085 * z + 5.0078$	7.28	$\min\{[z > 0] \cdot (0.0001x^2 - 0.0003 * x * y + 0.0003 * x * z + 0.0010y^2 + 0.0003 * y * z + 0.0040z^2 + 0.9995x + 2.0416y - 0.004z + 7.0485) + [z \leq 0] \cdot x, h^*\}$	inner error	-
Bin0	x	2	$x + 0.5 * y * n$	10.31	$x + [n > 0] \cdot 0.5 * y * n$	$x + [n > 0] \cdot 0.5 * y * n$	79.04
Bin2	x	2	$0.25 * n + x + 0.25 * n^2 + 0.5 * y * n$	10.12	$x + [n > 0] \cdot (0.25 * n + x + 0.25 * n^2 + 0.5 * y * n)$	$x + [n > 0] \cdot (0.25 * n + x + 0.25 * n^2 + 0.5 * y * n)$	250.60
DepRV	$x * y$	2	$-0.25 * n + 0.25 * n^2 + 0.5 * y * n + 0.5 * x * n + x * y$	9.57	$[n > 0] \cdot (-0.25 * n + 0.25 * n^2 + 0.5 * y * n + 0.5 * x * n + x * y) + [n \leq 0] \cdot x * y$	inner error	-
Prinsys	$[x == 1]$	2	$0.5 + 0.5 * x$	2.35	$[x == 1] * 1 + [x == 0] * 0.5$	$[x == 1] * 1 + [x == 0] * 0.5$	3.02
Sum0	x	2	$0.25 * i^2 + 0.25 * i + x$	2.33	$[i > 0] * (0.25 * i^2 + 0.25 * i) + x$	$x + [i > 0] * (0.25i + 0.25i^2)$	105.01
Duel	t	2	$-20.267 * x^2 - 0.4198 * x * t - 2.5502 * t^2 + 20.6657 * x + 3.5505 * t + 0.0013$	6.9	$\min\{[t > 0 \wedge x \geq 1] \cdot (-10.1335x^2 - 2.5502t^2 + 0.2099 * x * t + 10.1230 * x + 2.5502 * t + 0.5015) + [t \leq 0 \wedge x \geq 1] \cdot (-5.0668 * x^2 + 0.1050 * x * t - 2.5502 * t^2 + 5.0615 * x + 3.0504 * t + 0.2514) + [x < 1] \cdot t, h^*\}$	fail	-
BRP	$[failed = 10]$	2	$38912.3699 * failed^2 + 0.7329 * sent^2 + 3.2173 * failed * sent + 1486.258 * failed - 573.6644 * sent - 2459.9909$	10.12	$\min\{[failed < 10 \wedge sent < 800] \cdot (0.7329sent^2 + 0.0322 * failed * sent + 389.1237 * failed^2 + 793.1100 * failed - 572.1811 * sent - 2623.2068) + [failed = 10], h^*\}$	not terminate	-
CHAIN	$[y = 1]$	2	$-0.006 * x * y + 0.4841 * y^2 - 0.0021 * x + 0.4477 * y + 0.1007$	4.79	$\min\{[y = 0 \wedge x < 100] \cdot (-0.0059 * x * y + 0.4793 * y^2 - 0.0022 * x + 0.4373 * y + 0.1079) + [y = 1], h^*\}$	fail	-
GRID SMALL	$[a < 10 \wedge b \geq 10]$	3	$0.0018 * a * b^2 - 0.0003 * a^3 - 0.0008 * a^2 * b - 0.0011 * b^3 + 0.0117 * a^2 - 0.0154 * a * b + 0.0136 * b^2 - 0.097 * a + 0.0239 * b + 0.5355$	6.71	$\min\{[a < 10 \wedge b < 10] \cdot (-0.0003 * a^3 - 0.0011 * b^3 - 0.0008 * a^2 * b + 0.0018 * a * b^2 + 0.0109 * a^2 \cdots + 0.0277 * b + 0.5109) + [a < 10 \wedge b \geq 10], h^*\}$	Not support	-
GRID BIG	$[a < 1000 \wedge b \geq 1000]$	2	$0.0159 * a^2 - 0.0319 * a * b + 0.0159 * b^2 + 0.2715 * a - 0.3086 * b - 0.437$	7.74	$\min\{[a < 1000 \wedge b < 1000] \cdot (0.0159 * a^2 - 0.0319 * a * b + 0.0159 * b^2 + 0.2714 * a - 0.3087 * b - 0.4397) + [a < 1000 \wedge b \geq 1000], h^*\}$	Not support	-
CAV-2	$[h > 1 + t]$	3	0.0	3.78	$[h > t + 1]$	fail	-
CAV-4	$[x \leq 10]$	2	1.0	2.75	1.0	inner error	-
FIG-6	$[y \leq 5]$	4	$0.0011 * x^3 * y - 0.0001 * x^4 - 0.0001 * y^4 + 0.0008 * x * y^3 - 0.001 * x^2 * y^2 + \cdots + 0.5712 * x - 0.281 * y + 0.6009$	109.03	$\min\{[x \leq 4] \cdot (-0.0001 * x^4 + 0.0011 * x^3 * y - 0.001 * x^2 * y^2 + 0.0008 * x * y^3 - 0.0001 * y^4 + 0.0023 * x^3 \cdots - 0.0094 * y^2 + 0.5530 * x - 0.2782 * y + 0.6027) + [x > 4 \wedge y \leq 5], h^*\}$	inner error	-
FIG-7	$[x \leq 1000]$	2	$0.0005 * y^2 - 0.0008 * y * i + 0.0002 * i^2 - 0.0001 * x + 0.001 * y - 0.0005 * i + 1.0003$	24.32	$\min\{[y \leq 0] \cdot (0.0002 * i^2 - 0.0002 * x - 0.0005 * i + 1.0004) + [y > 0 \wedge x \leq 1000], h^*\}$	inner error	-
INV-Pend VARIANT	$[pA \leq 1]$	3	$0.0058 * pAD^2 * pA + 0.0023 * pAD^2 * cV - 0.1313 * pAD^2 * cP - 0.6278 * pAD * pA^2 - 0.2352pAD * pA * cV \cdots - 5.002cV * cP - 44.9405 * cP^2 - 5.7109 * cV + 1.0$	412.20	$\min\{[cp > 0.5 \vee cp < -0.5 \vee pA > 0.1 \vee pA < -0.1] \cdot (0.0058pAD^2pA - 0.0011pAD^2cV - 0.1313pAD^2 * cP + \cdots + 0.0689 * cP + 0.3238) + [-0.5 \leq cp \leq 0.5 \wedge -0.1 \leq pA \leq 0.1], h^*\}$	inner error	-
CAV-7	$[x \leq 30]$	3	$-0.0001 * i^3 + 0.0002 * i^2 * x + 0.0011 * i^2 - 0.0012 * i * x - 0.0009 * i - 0.0001 * x + 0.9993$	5.26	$\min\{[i < 5] \cdot (0.0001 * i^2 * x + 0.0005 * i^2 - 0.0006 * i * x + 0.0004 * i - 0.0011 * x + 0.9983) + [i < 5 \wedge x \leq 30], h^*\}$	inner error	-
CAV-5	$[i \leq 10]$	3	$-0.0001 * i * money^2 - 0.0004 * i^2 - 0.0006 * i * money + 0.1029 * money^2 + 0.0037 * i + 1.0$	892.6	$\min\{[money \geq 10] \cdot (-0.0001 * i * money^2 - 0.0004 * i^2 - 0.0004 * i * money + 0.0015 * i + 0.1028 * money^2 - 0.2118 * money + 3.1283) + [money < 10 \wedge i \leq 10], h^*\}$	inner error	-
Add	$[x > 5]$	3	$0.0005 * x^3 - 0.0055 * x^2 * y + 0.0272 * x * y^2 - 0.0491 * y^3 - 0.0109 * x^2 + 0.0513 * x * y - 0.0224 * y^2 + 0.0819 * x - 0.2123 * y + 0.9308$	3.63	$\min\{[y \leq 1] \cdot (-0.0491 * x^3 + 0.0272 * x^2 * y - 0.0055 * x * y^2 + 0.0005 * y^3 - 0.1348 * x^2 + 0.0926 * x * y - 0.015 * y^2 - 0.3568 * x + 0.1406 * y + 0.7181) + [y > 1 \wedge x > 5], h^*\}$	inner error	-
GrowingWalk VARIANT2	y	2	$0.0622 * x^2 - 1.2722 * x * r + 6.5027 * r^2 + 0.6396 * x + y - 6.5379 * r + 1.6433$	5.33	$\min\{[r \leq 0] \cdot (0.0622 * x^2 + 0.6279 * x + y + 1.6914) + [r > 0] \cdot y, h^*\}$	inner error	-

We present the comparison results in Table 4, whose illustration is the same to Table 2. To compare the two synthesized bounds, we uniformly sample grid points from a region of interest (typically a subset of the invariant) and evaluate both results at these points. We then compute the percentage of points at which our piecewise polynomial upper bound is larger (i.e., not better) than the (higher degree) monolithic polynomial, which is shown in the last column "PCT" in Table 4. We show that on all the benchmarks except GRID SMALL, GRID BIG, FIG-6, ADD, our piecewise polynomial bounds are *significantly* tighter and simpler than monolithic polynomial bounds. Although our running time is a bit longer than that of monolithic polynomial experiments, our approach allows to synthesize lower-degree polynomials while achieving better precision against higher-degree polynomials. This advantage is critical as the synthesis of higher-degree polynomials suffers from a large amount of numerical errors as stated previously.

7 RELATED WORKS & CONCLUSION

In this work, we propose a novel approach to synthesize piecewise probabilistic bounds for probabilistic programs. Further improvements include optimization on the branch reduction and the constraint solving of latticed k-induction constraints with minimum. Below we compare our approach with most related approaches.

Compared with previous approaches (e.g. [15, 16, 18]) that mostly focus on synthesizing monolithic bounds over probabilistic programs, our approach targets piecewise bounds, and hence is orthogonal. The work [11] proposes latticed k-induction. We claim that their work differs significantly from ours. They do not synthesize bounds and only verify whether a given bound is an upper bound or not. The work [10] synthesize piecewise linear bounds to verify the input upper bound via counterexample-guided inductive synthesis (CEGIS), while we do not need this additional input bound and we solve the bounds by bilinear and semidefinite programming rather than CEGIS. For the verification of lower bounds, their work applies a proof rule in [28, 32] derived from the original OST, while our approach applies extended OST. The work [5] synthesizes (piecewise) exact invariants and sub-invariants (to verify the input lower bound) via data-driven learning. Their work additionally requires a list of features composed of numerical expressions, while our approach captures the piecewise feature via k-induction automatically and without such additional inputs. The works [13, 57, 59] focus on deriving bounds for the posterior distribution in Bayesian probabilistic programs, whereas our work aims at deriving piecewise bounds for the expected output of the probabilistic programs.

Other approaches [3, 7, 8, 36] focus on moment-based invariants generation and high-order moments derivation for probabilistic programs. These works can even handle the probabilistic program with non-polynomial expressions and continuous distributions, but they only consider the probabilistic while loop in a rather restricted form: **while** `true` $\{C\}$. The work [42] enlarges the theoretical foundation through the assumption that all variables appearing in if-conditions (loop guards) are finitely valued , and [44] further provides an algorithm about computing the strongest polynomial moment invariants for this kind of loops, but their works still cannot handle most of our benchmarks. Our approach can handle all the polynomial forms of loop guards and if-conditions. In a similar vein, the works [39, 53] bound higher central moments for running time and other monotonically increasing quantities, but are limited to programs with constant size increments.

Table 4. Experimental Results for **RQ3**, Polynomial Case (Upper Bounds). "f" stands for the return function considered in the benchmark. "Piecewise Polynomial Upper Bound" stands for the results synthesized by our algorithm. "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, "T(s)" stands for the total execution time. "PCT" stands for the percentage of the points that our piecewise polynomial upper bound are lower (i.e., not better) than (higher degree) monolithic polynomial.

Benchmark	f	**Our Approach**			**Monolithic Polynomial via 1-induction**			**PCT**
		d	**T(s)**	**Piecewise Polynomial Upper Bound**	**d**	**T(s)**	**Monolithic Polynomial Upper Bound**	
GeoAr	x	2	7.28	$\min\{[z > 0] \cdot (0.0001x^2 - 0.0003 * x * y + 0.0003 * x * z + 0.0010y^2 + 0.0003 * y * z + 0.0040z^2 + 0.9995x + 2.0416y - 0.004z + 7.0485) + [z \leq 0] \cdot x, h^*\}$	3	1.52	$-0.0001x^3 + 0.0001 * x^2 * y - 0.0011 * x^2 * z - 0.0004 * x * y^2 - 0.0112 * x * y * z + 0.164 * x * z^2 + 0.0012 * y^3 + \cdots - 0.0137 * y^2 + 2.7194 * y * z + 0.9993 * x + 0.0417 * y + 89867.2768 * z + 0.078$	5.0%
Bin0	x	2	10.31	$x + [n > 0] \cdot 0.5 * y * n$	3	1.0	$0.5 * y * n + x$	0.0%
Bin2	x	2	10.12	$x + [n > 0] \cdot (0.25 * n + x + 0.25 * n^2 + 0.5 * y * n)$	3	1.03	$-0.0001 * x * y^2 - 0.0002 * x * y * n - 0.0001 * x * n^2 + 0.0009 * y^3 - 0.0009 * y^2 * n - 0.0011 * y * n^2 - 0.0001 * n^3 + 0.0004 * x * y - 0.0093 * y^2 + 0.5117 * y * n + 0.2496 * n^2 + 0.9986 * x + 0.033 * y + 0.2641 * n + 0.051$	21.4%
DepRV	$x * y$	2	9.57	$[n > 0] \cdot (-0.25 * n + 0.25 * n^2 + 0.5 * y * n + 0.5 * x * n + x * y) + [n \leq 0] \cdot x * y$	3	1.02	$x * y + 0.5 * x * n + 0.5 * y * n + 0.25 * n^2 - 0.2499 * n + 0.0001$	0.0%
Prinsys	$[x == 1]$	2	2.35	$[x == 1] * 1 + [x == 0] * 0.5$	3	0.75	$0.2973 * x^3 + 0.2027 * x + 0.5$	0.0%
Sum0	x	2	2.33	$0.25 * i^2 + 0.25 * i + x$	4	0.7	$0.25 * i^2 + 0.25 * i + x$	0.0%
Duel	t	2	6.90	$\min\{[t > 0 \wedge x \geq 1] \cdot (-10.1335x^2 - 2.5502t^2 + 0.2099 * x * t + 10.1230 * x + 2.5502 * t + 0.5015) + [t \leq 0 \wedge x \geq 1] \cdot (-5.0668 * x^2 + 0.1050 * x * t - 2.5502 * t^2 + 5.0615 * x + 3.0504 * t + 0.2514) + [x < 1] \cdot t, h^*\}$	4	0.92	$-175.0474x^4 - 33.1201x^3t - 256.8154 * x^2 * t^2 + 74.5673 * x * t^3 + 81.1314 * t^4 - 115.4608 * x^3 + 153.7459 * x^2 * t - 125.7204 * x * t^2 - 104.9856t^3 + 78.3171 * x^2 + 186.7714 * x * t - 135.7646 * t^2 + 212.334 * x + 160.6187 * t$	0.02%
brp	$[failed = 10]$	2	10.12	$\min\{[failed < 10 \wedge sent < 800] \cdot (0.7329sent^2 + 0.0322 * failed * sent + 389.1237 * failed^2 + 793.1100 * failed - 572.1811 * sent - 2623.2068) + [failed = 10], h^*\}$	4	1.27	$5.6049failed^4 + 4.902failed^3sent + 3.2666failed^3 - 0.0035failed^2sent^2 - 7.0269 * failed^2 * sent + 0.0019 * failed * sent^2 + 2.9608 * failed * sent - 0.0001sent^3 + 5.1816 * failed^2 - 0.0288 * sent^2 + 2.4293 * failed - 7.3179 * sent - 0.9176$	28.8%
chain	$[y = 1]$	2	4.79	$\min\{[y = 0 \wedge x < 100] \cdot (-0.0059 * x * y + 0.4793 * y^2 - 0.0022 * x + 0.4373 * y + 0.1079) + [y = 1], h^*\}$	3	1.15	$-0.0449 * x^3 - 0.5045 * x^2 * y + 5.611 * x * y^2 - 155242.5616 * y^3 + 5.5921 * x^2 + 43.0661 * x * y - 668140.0947 * y^2 - 117.5705 * x + 823721.7882 * y + 160.1718$	0.85%
grid small	$[a < 10 \wedge b \geq 10]$	3	6.71	$\min\{[a < 10 \wedge b < 10] \cdot (-0.0003 * a^3 - 0.0011 * b^3 - 0.0008 * a^2 * b + 0.0018 * a * b^2 + 0.0109 * a^2 \cdots + 0.0277 * b + 0.5109) + [a < 10 \wedge b \geq 10], h^*\}$	4	1.16	$0.0001 * a^3 - 0.0003 * a^2 * b + 0.0002 * a * b^2 - 0.0001 * b^3 - 0.0002 * a^2 + 0.0001 * a * b + 0.0003 * b^2 - 0.0326 * a + 0.0322 * b + 0.4628$	43.54%
grid big	$[a < 1000 \wedge b \geq 1000]$	2	7.74	$\min\{[a < 1000 \wedge b < 1000] \cdot (0.0159 * a^2 - 0.0319 * a * b + 0.0159 * b^2 + 0.2714 * a - 0.3087 * b - 0.4397) + [a < 1000 \wedge b \geq 1000], h^*\}$	3	0.83	$0.0005 * a^3 - 0.0044 * a^2 * b + 0.0052 * a * b^2 - 0.0023b^3 + 2.3321 * a^2 - 4.6674 * a * b + 2.3399b^2 + 34.489 * a - 42.4502 * b - 83.5854$	45.78%
cav-2	$[h > t + 1]$	3	3.78	$[h > t + 1]$	4	0.75	$0.0008 * h^2 - 0.001 * h * t + 0.001 * t^2 - 0.0066 * h - 0.0073 * t + 0.0885$	0.0%
cav-4	$[x \leq 10]$	2	2.75	1.0	3	0.62	$0.0007 * x * y^2 - 20.236 * y^3 - 0.0007 * x * y + 13.2821 * y^2 + 6.9539 * y + 1.0$	0.0%
fig-6	$[y \leq 5]$	4	109.03	$\min\{[x \leq 4] \cdot (-0.0001 * x^4 + 0.0011 * x^3 * y - 0.001 * x^2 * y^2 + 0.0008 * x * y^3 - 0.0001 * y^4 + 0.0023 * x^3 \cdots - 0.0094 * y^2 + 0.5530 * x - 0.2782 * y + 0.6027) + [x > 4 \wedge y \leq 5], h^*\}$	5	1.12	$-0.0001 * x^5 - 0.0002 * x^4 * y - 0.0003 * x^2 * y^3 + 0.0001 * x * y^4 - 0.0002 * y^5 + 0.0011 * x^4 + 0.0037 * x^3 * y \cdots + 0.1432 * x * y + 0.0064 * y^2 + 0.9708 * x - 0.6526 * y + 0.575$	42.73%
fig-7	$[x \leq 1000]$	2	24.32	$\min\{[y \leq 0] \cdot (0.0002 * i^2 - 0.0002 * x - 0.0005 * i + 1.0004) + [y > 0 \wedge x \leq 1000], h^*\}$	3	2.65	$0.0003 * x^2 * i - 0.083 * x^2 * y + 48.5638 * x * y^2 + 0.5267 * x * y * i - 0.018 * x * i^2 + 2600.9691 * y^3 - 36.705 * y^2 * i - 2.646 * y * i^2 \cdots - 3.3923 * x + 56310.8279 * y - 0.0114 * i + 7.2868$	2.58%
inv-Pend variant	$[pA \leq 1]$	3	412.20	$\min\{[cp > 0.5 \vee cp < -0.5 \vee pA > 0.1 \vee pA < -0.1] \cdot (0.0058pAD^2pA - 0.0011pAD^2cV - 0.1313pAD^2 * cP + \cdots + 0.0689 * cP + 0.3238) + [-0.5 \leq cp \leq 0.5 \wedge -0.1 \leq pA \leq 0.1], h^*\}$	4	7.42	$0.2264 * pAD^4 + 1.1448 * pAD^3 * pA - 0.1026 * pAD^3 * cV - 0.1107 * pAD^3 * cP + 5.2869 * pAD^2 * pA^2 + \cdots + 10.6625 * cP^2 - 0.0001 * pA + 53.8573 * cV + 1.0$	4.04%
CAV-7	$[x \leq 30]$	3	5.26	$\min\{[i < 5] \cdot (0.0001 * i^2 * x + 0.0005 * i^2 - 0.0006 * i * x + 0.0004 * i - 0.0011 * x + 0.9983) + [i < 5 \wedge x \leq 30], h^*\}$	4	1.17	$0.0007 * i^4 - 0.0011 * i^3 * x + 0.0005 * i^2 * x^2 - 0.0001 * i * x^3 - 0.0045 * i^3 + 0.0052 * i^2 * x - 0.0012 * i * x^2 + 0.0134 * i^2 - 0.012 * i * x + 0.002 * x^2 - 0.0135 * i + 0.0046 * x + 1.0034$	37.37%
cav-5	$[i \geq 10]$	3	892.6	$\min\{[money \geq 10] \cdot (-0.0001 * i * money^2 - 0.0004 * i^2 - 0.0004 * i * money + 0.0015 * i + 0.1028 * money^2 - 0.2118 * money + 3.1283) + [money < 10 \wedge i \leq 10], h^*\}$	4	1.27	$0.0001 * i^2 * money^2 + 0.0002 * i * money^3 + 0.0001 * money^4 + 0.0184 * i^2 * money - 0.0396 * i * money^2 - 0.0168 * money^3 + 0.0009 * i^3 - 0.0291 * i^2 + 2.8701 * i + 0.2414 * i * money + 4.264 * money^2 + 1.0$	0.0%
add	$[x > 5]$	3	3.63	$\min\{[y \leq 1] \cdot (-0.0491 * x^3 + 0.0272 * x^2 * y - 0.0055 * x * y^2 + 0.0005 * y^3 - 0.1348 * x^2 + 0.0926 * x * y - 0.015 * y^2 - 0.3568 * x + 0.1406 * y + 0.7181) + [y > 1 \wedge x > 5], h^*\}$	4	0.81	$0.0637x^4 + 4.4802 * x^3 * y - 4.4386 * x^2 * y^2 + 3.8156 * x * y^3 - 2.5543 * y^4 - 4.6104 * x^3 + 4.8566 * x^2 * y - 7.0417 * x * y^2 + 6.8972 * y^3 - 0.4752 * x^2 - 1.6341 * x * y - 2.8078 * y^2 + 5.0331 * x - 1.5381 * y$	43.94%
GrowingWalk variant2	y	2	5.33	$\min\{[r \leq 0] \cdot (0.0622 * x^2 + 0.6279 * x + y + 1.6914) + [r > 0] \cdot y, h^*\}$	3	1.22	$0.999 * x * r^2 + 0.0008 * y * r^2 + 700.3292 * r^3 - 1.999 * x * r - 0.0008 * y * r - 1399.6591 * r^2 + x + y + 698.3298 * r + 1.0001$	5.0 %

REFERENCES

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

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

[3] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. 2022. Solving Invariant Generation for Unsolvable Loops. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 19–43. https://doi.org/10.1007/978-3-031-22308-2_3

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking.* MIT Press.

[5] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13371)*, Sharon Shoham and Yakir Vizel (Eds.). Springer, 33–54. https://doi.org/10.1007/978-3-031-13185-1_3

[6] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 749–758. https://doi.org/10.1145/2933575.2934554

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

[8] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020. Mora - Automatic Generation of Moment-Based Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12078)*, Armin Biere and David Parker (Eds.). Springer, 492–498. https://doi.org/10.1007/978-3-030-45190-5_28

[9] Kevin Batz, Tom Jannik Biskup, Joost-Pieter Katoen, and Tobias Winkler. 2024. Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2792–2820. https://doi.org/10.1145/3632935

[10] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2023. Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 410–429. https://doi.org/10.1007/978-3-031-30820-8_25

[11] Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. 2021. Latticed k-Induction with an Application to Probabilistic Programs. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 524–549. https://doi.org/10.1007/978-3-030-81688-9_25

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

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

[14] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 33–52. https://doi.org/10.1145/2509136.2509546

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

[16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

[17] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 327–342. https://doi.org/10.1145/2837614.2837639

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

[19] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 145–160. https://doi.org/10.1145/3009837.3009873

[20] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[22] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. 2003. Bounded Model Checking and Induction: From Refutation to Verification (Extended Abstract, Category A). In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 14–26. https://doi.org/10.1007/978-3-540-45069-6_2

[23] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software Verification Using k-Induction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[24] J. L. Doob. 1971. What is a Martingale? *The American Mathematical Monthly* 78, 5 (1971), 451–463. https://doi.org/10.1080/00029890.1971.11992788 arXiv:https://doi.org/10.1080/00029890.1971.11992788

[25] Gy Farkas. 1894. A Fourier-féle mechanikai elv alkalmazásai. *Mathematikaiés Természettudományi Értesitö* 12 (1894), 457–472.

[26] Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 696–726. https://doi.org/10.1145/3586051

[27] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10482)*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26

[28] Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

[29] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[30] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 167–181. https://doi.org/10.1145/2593882.2593900

[31] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2013. Prinsys - On a Quest for Probabilistic Loop Invariants. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8054)*, Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio (Eds.). Springer, 193–208. https://doi.org/10.1007/978-3-642-40196-1_17

[32] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 37:1–37:28. https://doi.org/10.1145/3371105

[33] William B. Jones and W. J. Thron. 1984. Continued Fractions: Analytic Theory and Applications. https://api.semanticscholar.org/CorpusID:118226015

[34] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 364–389. https://doi.org/10.1007/978-3-662-49498-1_15

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

[36] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. 2023. Exact and Approximate Moment Derivation for Probabilistic Loops With Non-Polynomial Assignments. *CoRR* abs/2306.07072 (2023). https://doi.org/10.48550/ARXIV.2306.07072 arXiv:2306.07072

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

[38] Hari Govind Vediramana Krishnan, Yakir Vizel, Vijay Ganesh, and Arie Gurfinkel. 2019. Interpolating Strong Induction. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 367–385. https://doi.org/10.1007/978-3-030-25543-5_21

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

[40] Jia Lu and Ming Xu. 2022. Bisection Value Iteration. In *29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022*. IEEE, 109–118. https://doi.org/10.1109/APSEC57359.2022.00023

[41] Garth P. McCormick. 1976. Computability of global solutions to factorable nonconvex programs: Part I - Convex underestimating problems. *Math. Program.* 10, 1 (1976), 147–175. https://doi.org/10.1007/BF01580665

[42] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. 2022. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1497–1525. https://doi.org/10.1145/3563341

[43] Theodore Samuel Motzkin. 1936. Beiträge zur Theorie der linearen Ungleichungen. *(No Title)* (1936).

[44] Julian Müllner, Marcel Moosbrugger, and Laura Kovács. 2024. Strong Invariants Are Hard: On the Hardness of Strongest Polynomial Invariants for (Probabilistic) Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 882–910. https://doi.org/10.1145/3632872

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

[46] Mihai Putinar. 1993. Positive Polynomials on Compact Semi-algebraic Sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984. http://www.jstor.org/stable/24897130

[47] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 447–458. https://doi.org/10.1145/2491956.2462179

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

[49] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D.

Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[50] Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace abstraction modulo probability. *Proc. ACM Program. Lang.* 3, POPL (2019), 39:1–39:31. https://doi.org/10.1145/3290352

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

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

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

[54] Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Expected-Cost Analysis for Probabilistic Programs and Semantics-Level Adaption of Optional Stopping Theorems. *CoRR* abs/2103.16105 (2021). arXiv:2103.16105 https://arxiv.org/abs/2103.16105

[55] Jinyi Wang, Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2021. Quantitative analysis of assertion violations in probabilistic programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1171–1186. https://doi.org/10.1145/3453483.3454102

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

[57] Peixin Wang, Tengshun Yang, Hongfei Fu, Guanyan Li, and C.-H. Luke Ong. 2024. Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving. *Proc. ACM Program. Lang.* 8, PLDI, Article 202 (jun 2024), 26 pages. https://doi.org/10.1145/3656432

[58] David Williams. 1991. *Probability with Martingales.* Cambridge University Press.

[59] Fabian Zaiser, Andrzej S. Murawski, and C.-H. Luke Ong. 2025. Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops. *Proc. ACM Program. Lang.* 9, POPL (2025), 1104–1135. https://doi.org/10.1145/3704874

A SUPPLEMENTARY MATERIAL FOR SECTION 2.2

In this section, we supplement the introduction of the variant of k-induction operators proposed in [40], some important properties of these two k-induction operators, the equivalence between them and all their proofs.

Recall that in Section 2.2, we fix a lattice $(E, \sqsubseteq)$ and a monotone operator $\Phi : E \longrightarrow E$.

A.1 Property of the Upper k-Induction Operator in [11]

We attach an important property of the upper k-induction operator Ψ_u in [11] here.

THEOREM A.1 (PARK INDUCTION FROM k-INDUCTION [11]). *For any $u \in E$ and $k \in \mathbb{N}$, we have that* $\Phi(\Psi_u^k(u)) \sqsubseteq u \iff \Phi(\Psi_u^k(u)) \sqsubseteq \Psi_u^k(u)$.

The proof is given in [11, Lemma 2].

A.2 Upper k-Induction Operator in [40]

First we recall the definition of the upper k-induction operator proposed in [40].

Definition A.2 (The k-Induction Operator in [40]). The upper k-induction operator Ψ is defined by: $\Psi : E \longrightarrow E, v \mapsto \Phi(v) \sqcap v$.

Intuitively, it can be seen as a natural tightening of the operator Ψ_u, which considers the meet with the input element v itself. Below we introduce some important properties of the operator Ψ.

LEMMA A.3. *Let Ψ be the k-induction operator in [40] w.r.t. Φ. Then*

(1) Ψ is monotonic, i.e., $\forall v_1, v_2 \in E, v_1 \sqsubseteq v_2$ implies $\Psi(v_1) \sqsubseteq \Psi(v_2)$.
(2) Iterations of Ψ starting from u are descending, i.e.,

$$\ldots \sqsubseteq \Psi^k(u) \sqsubseteq \Psi^{k-1}(u) \sqsubseteq \ldots \sqsubseteq \Psi(u) \sqsubseteq u$$

And thus we have for all $m < n \in \mathbb{N}, \Psi^n(u) \sqsubseteq \Psi^m(u)$.

PROOF. For item (1), observe that if we have $w_1 \sqsubseteq w_2$ and $v_1 \sqsubseteq v_2$, then we have $w_1 \sqcap v_1 \sqsubseteq w_2 \sqcap v_2$.

$$\begin{aligned}\Psi(v_1) &= \Phi(v_1) \sqcap v_1 && \text{(by definition of } \Psi) \\ &\sqsubseteq \Phi(v_2) \sqcap v_2 && \text{(by monotonicity of } \Phi \text{ and above property)} \\ &= \Psi(v_2) && \text{(by definition of } \Psi)\end{aligned}$$

For item (2), we can immediately derived from the definition of Ψ as

$$\begin{aligned}\Psi^k(u) &= \Psi(\Psi^{k-1}(u)) && \text{(by definition of } \Psi^k(u)) \\ &= \Phi(\Psi^{k-1}(u)) \sqcap \Psi^{k-1}(u) && \text{(by definition of } \Psi) \\ &\sqsubseteq \Psi^{k-1}(u) && \text{(by definition of } \sqcap)\end{aligned}$$

□

PROPOSITION A.4. *For any $u \in E$, $\Phi(\Psi^k(u)) \sqsubseteq u \iff \Phi(\Psi^k(u)) \sqsubseteq \Psi^k(u)$.*

PROOF. The if-direction is trivial as $\Psi^k(u) \sqsubseteq u$ (by Lemma A.3(2)). For the only-if direction:

$$
\begin{aligned}
\Psi^k(u) &\sqsupseteq \Psi^{k+1}(u) && \text{(by Lemma A.3(2))}\\
&= \Phi(\Psi^k(u)) \sqcap \Psi^k(u) && \text{(by definition of } \Psi)\\
&= \Phi(\Psi^k(u)) \sqcap \Psi(\Psi^{k-1}(u)) && \text{(by definition of } \Psi^k(u))\\
&= \Phi(\Psi^k(u)) \sqcap (\Phi(\Psi^{k-1}(u)) \sqcap \Psi^{k-1}(u)) && \text{(by definition of } \Psi)\\
&= (\Phi(\Psi^k(u)) \sqcap \Phi(\Psi^{k-1}(u))) \sqcap \Psi^{k-1}(u)) && \text{(by associative law)}\\
&= \Phi(\Psi^k(u)) \sqcap \Psi^{k-1}(u) && \text{(by monotonicity of } \Phi \text{ and Lemma A.3(2))}\\
&\vdots\\
&= (\Phi(\Psi^k(u)) \sqcap \Phi(u)) \sqcap u && \text{(by unfolding } \Psi^k \text{ until } k=1)\\
&= \Phi(\Psi^k(u)) \sqcap u && \text{(by monotonicity of } \Phi \text{ and Lemma A.3(2))}\\
&= \Phi(\Psi^k(u)) && \text{(by the premise)}
\end{aligned}
$$

□

A.3 Equivalence between Ψ_u and Ψ

Theorem A.5 (Equivalence between Ψ_u and Ψ). *For any element $u \in E$, the sequence $\{\Psi_u^k(u)\}_{k\geq 0}$ of elements in E coincides with the sequence $\{\Psi^k(u)\}_{k\geq 0}$. In other words, for any natural number $k \geq 0$, we have that $\Psi_u^k(u) = \Psi^k(u)$.*

Proof. Proof by mathematical induction. We denote $X_k = \Psi_u^k(u)$ and $Y_k = \Psi^k(u)$. when $k = 0$, $X_0 = u = Y_0$. When $k = 1$, $X_1 = \Phi(u) \sqcap u = Y_1$, by definition of two operators, respectively.

Now we suppose that $X_k = Y_k$, i.e., $\Psi_u^k(u) = \Psi^k(u)$, and we aim to prove that $\Psi_u^{k+1}(u) = \Psi^{k+1}(u)$.

$$
\begin{aligned}
X_{k+1} &= \Psi_u(\Psi_u^k(u)) && \text{(by definition of } \Psi_u^{k+1}(u))\\
&= \Phi(\Psi_u^k(u) \sqcap u && \text{(by definition of } \Psi_u)
\end{aligned}
$$

$$
\begin{aligned}
Y_{k+1} &= \Psi(\Psi^k(u)) && \text{(by definition of } \Psi^{k+1}(u))\\
&= \Phi(\Psi^k(u)) \sqcap \Psi^k(u) && \text{(by definition of } \Psi)\\
&= \Phi(\Psi^k(u)) \sqcap \Psi(\Psi^{k-1}(u)) && \text{(by definition of } \Psi^k(u))\\
&= \Phi(\Psi^k(u)) \sqcap (\Phi(\Psi^{k-1}(u)) \sqcap \Psi^{k-1}(u)) && \text{(by definition of } \Psi)\\
&= (\Phi(\Psi^k(u)) \sqcap \Phi(\Psi^{k-1}(u))) \sqcap \Psi^{k-1}(u)) && \text{(by associative law)}\\
&= \Phi(\Psi^k(u)) \sqcap \Psi^{k-1}(u) && \text{(by monotonicity of } \Phi \text{ and Lemma A.3(2))}\\
&\vdots\\
&= (\Phi(\Psi^k(u)) \sqcap \Phi(u)) \sqcap u && \text{(by unfolding } \Psi^k \text{ until } k=1)\\
&= \Phi(\Psi^k(u)) \sqcap u && \text{(by monotonicity of } \Phi \text{ and Lemma A.3(2))}
\end{aligned}
$$

Since we suppose that $\Psi_u^k(u) = \Psi^k(u)$, we obtain that $\Phi(\Psi_u^k(u) \sqcap u = \Phi(\Psi^k(u)) \sqcap u$, thus we have $\Psi_u^{k+1}(u) = \Psi^{k+1}(u)$, i.e., $X_{k+1} = Y_{k+1}$. □

A.4 Supplementary Materials for the Dual k-Induction Operators Ψ'_u and Ψ'

We first give the definition of the Dual k-Induction Operators Ψ', which has been examined in [40].

Definition A.6 (Dual k-Induction Operator in [40]). The lower k-induction operator Ψ' is given by: $\Psi' : E \longrightarrow E, v \mapsto \Phi(v) \sqcup v$.

Lemma A.7. *Fix a lattice $(E, \sqsubseteq)$ and a monotone operator Φ. For any element $u \in E$, both of these two dual k-induction operators Ψ'_u and Ψ' have the following properties:*

(1) Ψ'_u (resp. Ψ') is monotone.
(2) Iterations of Ψ'_u (resp. Ψ') starting from u are ascending, i.e.,

$$u \sqsubseteq \Psi'_u(u) \sqsubseteq \ldots (\Psi'_u)^{k-1}(u) \sqsubseteq (\Psi'_u)^k(u) \ldots$$

$$u \sqsubseteq \Psi'(u) \sqsubseteq \ldots (\Psi')^{k-1}(u) \sqsubseteq (\Psi')^k(u) \ldots$$

Thus we have for all $m < n \in \mathbb{N}$, $(\Psi'_u)^m(u) \sqsubseteq (\Psi'_u)^n(u)$ and $(\Psi')^m(u) \sqsubseteq (\Psi')^n(u)$.

Proof. We only prove the case of dual k-induction operator Ψ'_u, since the proof of the properties of the dual k-induction operator Ψ' is similar with that of Ψ'_u.

For item (1), observe that if we have $w_1 \sqsubseteq w_2$, then we have $w_1 \sqcup u \sqsubseteq w_2 \sqcup u$. Assume that $v_1 \sqsubseteq v_2$

$$\begin{aligned} \Psi'_u(v_1) &= \Phi(v_1) \sqcup u && \text{(by definition of } \Psi'_h\text{)} \\ &\sqsubseteq \Phi(v_2) \sqcup u && \text{(by monotonicity of } \Phi \text{ and above property)} \\ &= \Psi'_u(v_2) && \text{(by definition of } \Psi'_h\text{)} \end{aligned}$$

For item (2), we prove it by mathematical induction. We have $u \sqsubseteq \Psi'_u(u)$ as $\Psi'_u(u) = \Phi(u) \sqcup u$. We then assume that $(\Psi'_u)^k(u) \sqsupseteq (\Psi'_h)^{k-1}(u)$, and we prove that

$$\begin{aligned} (\Psi'_u)^{k+1}(u) &= \Psi'_u((\Psi'_u)^k(u)) && \text{(by definition of } (\Psi'_u)^{k+1}(u)\text{)} \\ &\sqsupseteq \Psi'_u((\Psi'_u)^{k-1}(u)) && \text{(by monotonicity of } \Psi'_u \text{ and assumption)} \\ &= (\Psi'_u)^k(u) && \text{(by definition of } (\Psi'_u)^k(u)\text{)} \end{aligned}$$

Thus the value sequence is an ascending chain. □

Proposition A.8. *For any element $u \in E$, the lower k-induction operators Ψ'_u and Ψ' have the following properties:*

$$\Phi((\Psi'_u)^k(u)) \sqsupseteq u \iff \Phi((\Psi'_u)^k(u)) \sqsupseteq (\Psi'_u)^k(u)$$
$$\Phi((\Psi')^k(u)) \sqsupseteq u \iff \Phi((\Psi')^k(u)) \sqsupseteq (\Psi')^k(u)$$

Proof. For the case of the dual k-induction operator Ψ'_u:

The if-direction is trivial as $(\Psi'_u)^k(u) \sqsupseteq u$ (by Lemma A.7(2)). For the only-if direction:

$$\begin{aligned} (\Psi'_u)^k(u) &\sqsubseteq (\Psi'_u)^{k+1}(u) && \text{(by Lemma A.7(2))} \\ &= \Psi'_u((\Psi'_u)^k(u)) && \text{(by the definition of } (\Psi'_u)^{k+1}(u)\text{)} \\ &= \Phi((\Psi'_u)^k(u)) \sqcup u && \text{(by the definition of } \Psi'_u\text{)} \\ &= \Psi((\Psi'_u)^k(u)) && \text{(by the premise)} \end{aligned}$$

For the case of the dual k-induction operator Ψ':

The if-direction is trivial as $(\Psi')^k(u) \sqsupseteq u$ (by Lemma A.7(2)). For the only-if direction:

$$
\begin{aligned}
(\Psi')^k(u) \sqsubseteq (\Psi')^{k+1}(u) & & \text{(by Lemma A.7(2))}\\
= \Psi'((\Psi')^k(u)) & & \text{(by the definition of } (\Psi')^{k+1}(u))\\
= \Phi((\Psi')^k(u)) \sqcup (\Psi')^k(u) & & \text{(by the definition of } \Psi')\\
= \Phi((\Psi')^k(u)) \sqcup \Psi'((\Psi')^{k-1}(u)) & & \text{(by the definition of } (\Psi')^k(u))\\
= \Phi((\Psi')^k(u)) \sqcup \Phi((\Psi')^{k-1}(u)) \sqcup (\Psi')^{k-1}(u) & & \text{(by the definition of } \Psi')\\
= (\Phi((\Psi')^k(u)) \sqcup \Phi((\Psi')^{k-1}(u))) \sqcup (\Psi')^{k-1}(u) & & \text{(by associate law)}\\
= \Phi((\Psi')^k(u)) \sqcup (\Psi')^{k-1}(u) & & \text{(by monotonicity of } \Phi \text{ and Lemma A.7(2))}\\
\vdots & & \\
= \Phi((\Psi')^k(u)) \sqcup \Psi'(u) & & \text{(by unfolding } (\Psi')^k(u) \text{ until } k=1)\\
= \Phi((\Psi')^k(u)) \sqcup \Phi(u) \sqcup u & & \text{(by definition of } \Psi')\\
= \Phi((\Psi')^k(u)) \sqcup u & & \text{(by monotonicity of } \Phi \text{ and Lemma A.7(2))}\\
= \Phi((\Psi')^k(u)) & & \text{(by the premise)}
\end{aligned}
$$

□

A.5 Equivalence between Ψ'_u and Ψ'

THEOREM A.9 (EQUIVALENCE BETWEEN Ψ'_u AND Ψ'). *For any element $u \in E$, we have that the sequence $\{(\Psi'_u)^k(u)\}_{k\geq 0}$ of elements in E coincides with the sequence $\{(\Psi')^k(u)\}_{k\geq 0}$. In other words, for any natural number $k \geq 0$, we have that $(\Psi'_u)^k(u) = (\Psi')^k(u)$.*

PROOF. Analogously, we proof it by mathematical induction. $X_k = (\Psi'_u)^k(u)$ and $Y_k = (\Psi')^k(u)$. when $k = 0$, $X_0 = u = Y_0$. When $k = 1$, $X_1 = \Phi(u) \sqcup u = Y_1$, by definition of two dual operators, respectively.

Now we suppose that $X_k = Y_k$, i.e., $(\Psi'_u)^k(u) = (\Psi')^k(u)$, and we aim to prove that $(\Psi'_u)^{k+1}(u) = (\Psi')^{k+1}(u)$.

$$
\begin{aligned}
X_{k+1} = \Psi'_u((\Psi'_u)^k(u)) & & \text{(by definition of } (\Psi'_u)^{k+1}(u))\\
= \Phi((\Psi'_u)^k(u)) \sqcup u & & \text{(by definition of } \Psi'_u)
\end{aligned}
$$

$$
\begin{aligned}
Y_{k+1} = \Psi'((\Psi')^k(u)) & & \text{(by definition of } (\Psi')^{k+1}(u))\\
= \Phi((\Psi')^k(u)) \sqcup (\Psi')^k(u) & & \text{(by definition of } \Psi')\\
= \Phi((\Psi')^k(u)) \sqcup \Psi'((\Psi')^{k-1}(u)) & & \text{(by definition of } (\Psi')^k(u))\\
= \Phi((\Psi')^k(u)) \sqcup (\Phi((\Psi')^{k-1}(u)) \sqcup \Psi'^{k-1}(u)) & & \text{(by definition of } \Psi')\\
= (\Phi((\Psi')^k(u)) \sqcup \Phi((\Psi')^{k-1}(u))) \sqcup \Psi'^{k-1}(u)) & & \text{(by associative law)}\\
= \Phi((\Psi')^k(u)) \sqcup (\Psi')^{k-1}(u) & & \text{(by monotonicity of } \Phi \text{ and Lemma A.7(2))}\\
\vdots & & \\
= (\Phi((\Psi')^k(u)) \sqcup \Phi(u)) \sqcup u & & \text{(by unfolding } (\Psi')^k \text{ until } k=1)\\
= \Phi((\Psi')^k(u)) \sqcup u & & \text{(by monotonicity of } \Phi \text{ and Lemma A.7(2))}
\end{aligned}
$$

Since we suppose that $(\Psi'_u)^k(u) = (\Psi')^k(u)$, we obtain that $\Phi((\Psi'_u)^k(u) \sqcup u = \Phi((\Psi')^k(u)) \sqcup u$, thus we have $(\Psi'_u)^{k+1}(u) = (\Psi')^{k+1}(u)$, i.e., $X_{k+1} = Y_{k+1}$. □

B SUPPLEMENTARY MATERIAL FOR SECTION 4

B.1 Classical OST

Optional Stopping Theorem (OST) is a classical theorem in martingale theory that characterizes the relationship between the expected values initially and at a stopping time in a supermartingale. Below we present the classical form of OST.

Theorem B.1 (Optional Stopping Theorem (OST) [58, Chapter 10]). *Let $\{X_n\}_{n=0}^{\infty}$ be a martingale (resp. supermartingale) adapted to a filtration $\mathcal{F} = \{\mathcal{F}_n\}_{n=0}^{\infty}$ and τ be a stopping time w.r.t the filtration $\mathcal{F}$. If we have that:*

- *$\mathbb{E}(\tau) < \infty$;*
- *exists an $M \in [0, \infty)$ such that $|X_{n+1} - X_n| \le M$ holds almost surely for every $n \ge 0$,*

then it follows that $(|X_\tau|) < \infty$ and $\mathbb{E}(X_\tau) = \mathbb{E}(X_0)$ (resp. $\mathbb{E}(X_\tau) \le \mathbb{E}(X_0)$).

Since the classical Optional Stopping Theorem [24, 58] requires bounded step-wise difference $|X_{n+1} - X_n|$ in a stochastic process $\{X_n\}_{n\ge 0}$, which cannot handle our problem due to the assignment commands in the loop body. To address this difficulty, We have sought several extended versions of OST, as proposed in [54, 56, 57], etc. Among which we find the OST variant proposed in [57] can handle our problem.

B.2 Proof of Theorem 4.10

Theorem 4.10. Suppose the loop P is affine. Let k be a positive integer and h be a polynomial potential function in the program variables with degree d. If there exist real numbers $c_1 > 0$ and $c_2 > c_3 > 0$ such that

(P1) there exists a uniform amplifier c satisfying $c \le e^{c_3/d}$, and
(P2) the termination time T of P has the *concentration property*, i.e., $\mathbb{P}(T > n) \le c_1 \cdot e^{-c_2 \cdot n}$.

hold, then for any initial program state s^*, we have:

- $\mathbb{E}_{s^*}(X_f) \le \overline{\Psi}_h^{k-1}(h)(s^*) \le h(s^*)$ holds for any k-upper potential function h.
- $\mathbb{E}_{s^*}(X_f) \ge (\overline{\Psi}'_h)^{k-1}(h)(s^*) \ge h(s^*)$ holds for any k-lower potential function h.

Proof. We first proof the soundness of upper potential functions. Let s_n be the random vector (random variable) of the program state at the n-th iteration of the probabilistic while loop P, where $s_0 = s^*$, and let $\{\mathcal{F}_n\}_{n\ge 0}$ be the filtration such that each $\mathcal{F}_n$ is the σ-algebra that describes the first n iterations of the loop, i.e., the smallest σ-algebra that makes the random values during the first n executions measurable. This choice of $\mathcal{F}_n$ is standard in previous martingale-based results [17–19, 56].

We also define $H = \overline{\Psi}_h^{k-1}(h)$. Note that H is piecewise linear or polynomial (by the definition of $\overline{\Psi}_h$ in Definition 4.4) . By Definition 4.5 and the property that $\overline{\Phi}(\overline{\Psi}_h^{k-1}(h)) \preceq h \iff \overline{\Phi}(\overline{\Psi}_h^{k-1}(h)) \preceq \overline{\Psi}_h^{k-1}(h)$ (Theorem A.1), we obtain that $\forall s \in \mathit{Reach}(s^*)$, $\overline{\Phi}(H)(s) \le H(s)$. We define the stochastic process $\{X_n\}_{n=0}^{\infty}$ by

$$X_n := H(s_n).$$

We first prove that the stochastic process $\{X_n\}$ is a supermartingale. We discuss this in the following two scenarios:

- if $s_n \not\models \varphi$, by the semantics of probabilistic while loop (see Section 2.3), $s_{n+1} = s_n$, and thus $X_{n+1} = X_n$, which satisfies the conditions of supermartingale;
- if $s_n \models \varphi$, we have

$$\begin{aligned}
\mathbb{E}_{s^*}[X_{n+1}|\mathcal{F}_n] &= \mathbb{E}_{s^*}[H(s_{n+1})|\mathcal{F}_n] \\
&= \mathbb{E}_{s_n}[H(s_{n+1})|\mathcal{F}_n] && \text{(by definition of conditional expectation)} \\
&= pre_C(H)(s_n) && \text{(by definition of pre-expectation)} \\
&= \overline{\Phi}(H)(s_n) && \text{(by definition of characteristic function)} \\
&\leq H(s_n) && \text{(by property of } H\text{)} \\
&= X_n
\end{aligned}$$

where the property of conditional expectation is the "take out what is known" property of conditional expectation (see [58]). From (P1) and the definition of uniform amplifier (see Definition 4.8), for each program variable x, the value of x_n is bounded by $|X_n| \leq c^n \cdot |x_0| + a \cdot (c^0 + \cdots c^{n-1}) \leq K_n \cdot c^n \leq K_n \cdot e^{c_3 * n/d}$ for some positive constant K_n. From that H is piecewise linear (resp. polynomial with degree d), i.e., H is linear (resp. polynomial with degree d) on each segment, we can obtain $\mathbb{E}_{s^*}[X_n] = \mathbb{E}_{s^*}[H(s_n)] = \mathbb{E}_{s^*}[M_n \cdot c^n] < \infty$ for some positive constant $M_n > 0$ by the definition of X_n. Thus $\{X_n\}$ is a supermartingale.

The condition (a) in Theorem 4.6 follows from the assumption that (P2) P has the concentration property.

Then we prove the condition (b) in Theorem 4.6. From (P1), we have that for each program variable x, the value of x_n at n-th iteration, i.e., at the program state s_n, is bounded by $K_n \cdot c^n$. When H is piecewise linear, i.e., $d = 1$, we have that $H(s_n) \leq M_n \cdot c^n$ for $M_n > 0$.

$$\begin{aligned}
|X_{n+1} - X_n| &= |H(s_{n+1}) - H(s_n)| \\
&\leq |H(s_{n+1})| + |H(s_n)| \\
&\leq M_n \cdot |c|^n + M_{n+1} \cdot |c|^{n+1} \\
&\leq (M_n + |c| \cdot M_{n+1}) \cdot |c|^n \\
&\leq b_1 \cdot e^{c_3 n}
\end{aligned}$$

When H is piecewise polynomial with degree d, we have that $H(s_n) \leq M_n \cdot c^{nd}$ for $M_n > 0$.

$$\begin{aligned}
|X_{n+1} - X_n| &= |H(s_{n+1}) - H(s_n)| \\
&\leq |H(s_{n+1})| + |H(s_n)| \\
&\leq M_n \cdot |c|^{nd} + M_{n+1} \cdot |c|^{(n+1)d} \\
&\leq (M_n + |c^d| \cdot M_{n+1}) \cdot |c|^{nd} \\
&\leq b_1 \cdot (e^{c_3/d})^{nd} \\
&\leq b_1 \cdot e^{c_3 n}
\end{aligned}$$

Especially, if the uniform amplifier c is chosen as 1, then c_3 can be chosen arbitrarily small, the prerequisites of this theorem always holds regardless of the values taken by c_2 and d.

By applying Theorem 4.6, we have that $\mathbb{E}_{s^*}(X_T) \leq \mathbb{E}_{s^*}(X_0)$. Since the termination time T is a stopping time w.r.t. the filtration $\{\mathcal{F}_n\}_{n\geq 0}$, and there will be $s_T \not\models \varphi$, thus $X_T = f(s_T) = X_f$. We have $\mathbb{E}_{s^*}(X_f) \leq \mathbb{E}_{s^*}(X_0) = H(s^*)$. The second inequality, i.e., $\overline{\Psi}_h^{k-1}(h)(s^*) \leq h(s^*)(\forall s^*)$ can be derived directly from the property that $\overline{\Psi}_h^{k-1}(h) \preceq h$ holds (see Appendix A.2 and [11]). The case of lower potential functions is completely dual to the case of upper potential functions since we can consider the stochastic process $\{-X_n\}$, that is, define the stochastic process by $Y_n := -H(s_n)$. The remaining proof is essentially the same.

□

B.3 Proof of Theorem 4.11

Theorem 4.11. Let k be a positive integer. Suppose there exist real numbers $c_1 > 0$ and $c_2 > 0$ such that condition (P1') loop P has the bounded update property; and condition (P2) in Theorem 4.10 holds, then for any initial program state s^*, we have

- $\mathbb{E}_{s^*}(X_f) \leq \overline{\Psi}_h^{k-1}(h)(s^*) \leq h(s^*)$ holds for any k-upper potential function h.
- $\mathbb{E}_{s^*}(X_f) \geq (\overline{\Psi}'_h)^{k-1}(h)(s^*) \geq h(s^*)$ holds for any k-lower potential function h.

Proof. We first proof the soundness of upper potential functions. Let s_n be the random vector (random variable) of the program state at the n-th iteration of the probabilistic while loop P, where $s_0 = s^*$, and let $\{\mathcal{F}_n\}_{n\geq 0}$ be the filtration such that each $\mathcal{F}_n$ is the σ-algebra that describes the first n iterations of the loop, i.e., the smallest σ-algebra that makes the random values during the first n executions measurable. This choice of $\mathcal{F}_n$ is standard in previous martingale-based results [17–19, 56].

We also define $H = \overline{\Psi}_h^{k-1}(h)$. Note that H is piecewise linear or polynomial (by the definition of $\overline{\Psi}_h$ in Definition 4.4) . By Definition 4.5 and the property that $\overline{\Phi}(\overline{\Psi}_h^{k-1}(h)) \preceq h \iff \overline{\Phi}(\overline{\Psi}_h^{k-1}(h)) \preceq \overline{\Psi}_h^{k-1}(h)$ (Theorem A.1), we obtain that $\forall s \in \mathit{Reach}(s^*), \overline{\Phi}(H)(s) \leq H(s)$. We define the stochastic process $\{X_n\}_{n=0}^{\infty}$ by

$$X_n := H(s_n).$$

We first prove that the stochastic process $\{X_n\}$ is a supermartingale. We discuss this in the following two scenarios:

- if $s_n \not\models \varphi$, by the semantics of probabilistic while loop (see Section 2.3), $s_{n+1} = s_n$, and thus $X_{n+1} = X_n$, which satisfies the conditions of supermartingale;
- if $s_n \models \varphi$, we have

$$\begin{aligned}
\mathbb{E}_{s^*}[X_{n+1}|\mathcal{F}_n] &= \mathbb{E}_{s^*}[H(s_{n+1})|\mathcal{F}_n] \\
&= \mathbb{E}_{s_n}[H(s_{n+1})|\mathcal{F}_n] && \text{(by definition of conditional expectation)} \\
&= \mathit{pre}_C(H)(s_n) && \text{(by definition of pre-expectation)} \\
&= \overline{\Phi}(H)(s_n) && \text{(by definition of characteristic function)} \\
&\leq H(s_n) && \text{(by property of } H) \\
&= X_n
\end{aligned}$$

where the property of conditional expectation is the "take out what is known" property of conditional expectation (see [58]). From (P1') that P has the bounded update property and H is a piecewise polynomial with degree d, i.e., H is a polynomial with degree d on each segment, we can obtain $\mathbb{E}_{s^*}[X_n] = \mathbb{E}_{s^*}[H(s_n)] \leq \zeta \cdot n^d$ for a positive constant $\zeta > 0$, thus $\{X_n\}$ is a supermartingale.

The condition (a) in Theorem 4.6 follows from the assumption that (P2) P has the concentration property.

Then we prove the condition (b) in Theorem 4.6. From that P has the bounded update property and H is a piecewise polynomial with degree d, we also have that $|X_n| \leq \zeta \cdot n^d$ for a positive constant $\zeta > 0$, thus we have

$$\begin{aligned} |X_{n+1} - X_n| &\leq |X_{n+1}| + |X_n| \\ &\leq \zeta \cdot n^d + \zeta \cdot (n+1)^d \\ &\leq b_1 \cdot n^d \end{aligned}$$

Note that in this theorem, c_3 in Theorem 4.6(b) is chosen arbitrarily small, therefore the prerequisites of Theorem 4.6 always holds regardless of the values taken by c_2.

By applying Theorem 4.6, we have that $\mathbb{E}_{s^*}(X_T) \leq \mathbb{E}_{s^*}(X_0)$. Since the termination time T is a stopping time w.r.t. the filtration $\{\mathcal{F}_n\}_{n\geq 0}$, and there will be $s_T \not\models \varphi$, thus $X_T = f(s_T) = X_f$. We have $\mathbb{E}_{s^*}(X_f) \leq \mathbb{E}_{s^*}(X_0) = H(s^*)$. The second inequality, i.e., $\overline{\Psi}_h^{k-1}(h)(s^*) \leq h(s^*)(\forall s^*)$, can be derived directly from the property that $\overline{\Psi}_h^{k-1}(h) \preceq h$ holds (see Appendix A.2 and [11]). The case of lower potential functions is completely dual to the case of upper potential functions since we can consider the stochastic process $\{-X_n\}$, that is, define the stochastic process by $Y_n := -H(s_n)$. The remaining proof is essentially the same.

□

C SUPPLEMENTARY MATERIAL FOR SECTION 5

C.1 Supplementary Material for Brute-Force Arithmetic Expansion in Stage 2

In this section, we supplement the brute-force arithmetic expansion that can simplify the k-induction constraint. To transform the k-induction constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ into a simpler form, our algorithm further unrolls this k-induction conditions so that the minimum operations appear at the outermost of the left-hand-side of the inequality. In detail, from the definition of the operator $\overline{\Psi}_h$ (Definition 4.4), the unrolling is reduced to the recursive computation of *pre-expectation* and the pointwise minimum operation. Following the definition of pre-expectation (Definition 4.2), the unrolling can be done by the following reduction rules for functions $f_1, \ldots, f_m, g_1, \ldots, g_n$:

(R1) $\min\{f_1, \ldots, f_m\} + \min\{g_1, \ldots, g_n\} = \min_{1\leq i\leq m, 1\leq j\leq n}\{f_i + g_j\}$;
(R2) $c \cdot \min\{f_1, \ldots, f_m\} = \min\{c \cdot f_1, \ldots, f_m\}$ for constant $c \geq 0$;
(R3) $[B] \cdot \min\{f_1, \ldots, f_m\} = \min\{[B] \cdot f_1, \ldots, [B] \cdot f_m\}$ for predicate B.

By iterative applications of the reduction rules, the constraint $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ can be transformed into a succinct form with only one minimum operation:

$$\min\{h_1, h_2, \ldots, h_m\} \preceq h$$

where h is the predefined polynomial template and each h_i $(i = 1, \ldots, m)$ is a piecewise expression derived from the unrolling that does not contain the minimum operation.

C.2 Proof of Proposition 5.2

We give a proof for Proposition 5.2 in this section.

Proposition 5.2. The upper k-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$ is equivalent to constraint $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i equals $pre_{C_d}(h)$ for some unique $C_d \in \{C_1, \ldots, C_m\}$ from the unfolding process above.

Proof. We concentrate on the left side of the constraint: $\overline{\Phi}_f(\overline{\Psi}_h^{k-1}(h)) \preceq h$.

We first proof the case of $k = 2$, i.e., $\overline{\Phi}_f(\overline{\Psi}_h^{1}(h)) \preceq h$. Since our syntax of the probabilistic programs is defined in a compositional style (see Fig. 1 in Section 2.3 for more details), we proof by induction on the structure of programs. For simplicity, we denote $pre_C([\Phi])$ by $[\Phi(C)]$, which represent the evaluation of $[\Phi]$ after the execution of C.

- Case $C \equiv \text{skip}$.

$$
\begin{aligned}
& \overline{\Phi}_f(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \overline{\Psi}_h(h) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \min\{\overline{\Phi}_f(h), h\} \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \min\{[\neg\varphi] \cdot f + [\varphi] \cdot h, h\} \\
= \; & [\neg\varphi] \cdot f + \min\{[\varphi] \cdot h, [\varphi] \cdot h\} \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot h \\
= \; & \overline{\Phi}_f(h)
\end{aligned}
$$

  It corresponds to pre-expectation of the loop-free program unfolded with twice (only one program).

- Case $C \equiv x := e$.

$$
\begin{aligned}
& \overline{\Phi}_f(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \overline{\Psi}_h(h)([x/e]) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \min\{[\neg\varphi] \cdot f + [\varphi] \cdot h([x/e]), h\}([x/e]) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \min\{[\neg\varphi([x/e])] \cdot f([x/e]) + \\
& [\varphi([x/e])] \cdot h([x/e])([x/e]), h([x/e])\} \\
= \; & \min\{[\neg\varphi] \cdot f + [\varphi \wedge \neg\varphi([x/e])] \cdot f([x/e]) + \\
& [\varphi \wedge \varphi([x/e])] \cdot h([x/e])([x/e]), [\neg\varphi] \cdot f + [\varphi] \cdot h([x/e])\} \\
= \; & \min\{[\neg\varphi] \cdot f + [\varphi \wedge \neg\varphi([x/e])] \cdot f([x/e]) + \\
& [\varphi \wedge \varphi([x/e])] \cdot pre_{C;C}(h), [\neg\varphi] \cdot f + [\varphi] \cdot h([x/e])\}
\end{aligned}
$$

  the expressions in the minimize operator correspond to pre-expectation of the two loop-free programs unfolded within twice (one for once, and another for twice).

- Case $C \equiv C_1; C_2$.

$$
\begin{aligned}
& \overline{\Phi}_f(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] pre_{C_1}(pre_{C_2}(\min\{[\neg\varphi] \cdot f + [\varphi] \cdot pre_{C_1}(pre_{C_2}(h)), h\})) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot \min\{[\neg\varphi(C_1; C_2)] \cdot pre_{C_1;C_2}(f) + \\
& [\varphi(C_1; C_2)] \cdot pre_{C_1;C_2}(h), pre_{C_1;C_2}(h)\} \\
= \; & \min\{[\neg\varphi] \cdot f + [\varphi \wedge \varphi(C_1; C_2)] \cdot pre_{C_1;C_2}(f) + \\
& [\varphi \wedge \neg\varphi(C_1; C_2)] \cdot pre_{C_1;C_2}(h), \\
& [\neg\varphi] \cdot f + [\varphi] \cdot pre_{C_1;C_2}(h)\}
\end{aligned}
$$

  the expressions in the minimize operator correspond to pre-expectation of the two loop-free programs unfolded within twice (one for once, and another for twice)

- case $C \equiv \{C_1\}[p]\{C_2\}$.

$$
\begin{aligned}
& \overline{\Phi}_f(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot p \cdot pre_{C_1}(\overline{\Psi}_h(h)) + [\varphi] \cdot (1-p) \cdot pre_{C_2}(\overline{\Psi}_h(h))
\end{aligned}
$$

wherein

$$
\begin{aligned}
pre_{C_1}(\overline{\Psi}_h(h)) \; = \; & pre_{C_1}(\min\{[\neg\varphi] \cdot f + [\varphi] \cdot (p \cdot pre_{C_1}(h) + (1-p) \cdot pre_{C_2}(h)), h\} \\
= \; & \min\{[\neg\varphi(C_1)] \cdot pre_{C_1}(f) + [\varphi(C_1)] \cdot \\
& (p \cdot pre_{C_1;C_1}(h) + (1-p) \cdot pre_{C_1;C_2}(h)), pre_{C_1}(h)\}
\end{aligned}
$$

and

$$
\begin{aligned}
pre_{C_2}(\overline{\Psi}_h(h)) \; = \; & pre_{C_2}(\min\{[\neg\varphi] \cdot f + [\varphi] \cdot (p \cdot pre_{C_1}(h) + (1-p) \cdot pre_{C_2}(h)), h\} \\
= \; & \min\{[\neg\varphi(C_2)] \cdot pre_{C_2}(f) + [\varphi(C_2)] \cdot \\
& (p \cdot pre_{C_2;C_1}(h) + (1-p) \cdot pre_{C_2;C_2}(h)), pre_{C_2}(h)\}
\end{aligned}
$$

Thus we have

$$
\begin{aligned}
& \overline{\Phi}_f(\overline{\Psi}_h(h)) \\
= \; & [\neg\varphi] \cdot f + [\varphi] \cdot p \cdot \min\{[\neg\varphi(C_1)] \cdot pre_{C_1}(f) \\
& +[\varphi(C_1)] \cdot (p \cdot pre_{C_1;C_1}(h) + (1-p) \cdot pre_{C_1;C_2}(h)), pre_{C_1}(h)\} + \\
& [\varphi] \cdot (1-p) \cdot \min\{[\neg\varphi(C_2)] \cdot pre_{C_2}(f) \\
& +[\varphi(C_2)] \cdot (p \cdot pre_{C_2;C_1}(h) + (1-p) \cdot pre_{C_2;C_2}(h)), pre_{C_2}(h)\} \\
= \; & \min\{[\neg\varphi] \cdot f + [\varphi \wedge \neg\varphi(C_1)] \cdot p \cdot pre_{C_1}(f) + [\varphi \wedge \neg\varphi(C_2)] \cdot (1-p) \cdot pre_{C_2}(f) \\
& +[\varphi \wedge \varphi(C_1)] \cdot (p^2 \cdot pre_{C_1;C_1}(h) + p(1-p) \cdot pre_{C_1;C_2}(h)) \\
& +[\varphi \wedge \varphi(C_2)] \cdot ((1-p)p \cdot pre_{C_2;C_1}(h) + (1-p)^2 \cdot pre_{C_2;C_2}(h)), \\
& [\neg\varphi] \cdot f + [\varphi \wedge \neg\varphi(C_1)] \cdot p \cdot pre_{C_1}(f) + \\
& [\varphi \wedge \varphi(C_1)] \cdot (p^2 \cdot pre_{C_1;C_1}(h) + p(1-p) \cdot pre_{C_1;C_2}(h)) + \\
& [\varphi] \cdot (1-p) \cdot pre_{C_2}(h), \\
& [\neg\varphi] \cdot f + [\varphi \wedge \neg\varphi(C_2)] \cdot (1-p) \cdot pre_{C_2}(f) + \\
& [\varphi \wedge \varphi(C_2)] \cdot ((1-p)p \cdot pre_{C_2;C_1}(h) + (1-p)^2 \cdot pre_{C_2;C_2}(h)) + \\
& [\varphi] \cdot p \cdot pre_{C_1}(h), \\
& [\neg\varphi] \cdot f + [\varphi] \cdot p \cdot pre_{C_1}(h) + [\varphi] \cdot (1-p) \cdot pre_{C_2}(h)
\end{aligned}
$$

The first expression corresponds to the case that we unfold for twice at each state we reach (after the execution of C_1 and C_2), and the second (resp. third) expression corresponds to the case that we unfold for twice at the state that we reach after the execution of C_1 (resp. C_2) and unfold for once at the state that we reach after the execution of C_2 (resp. C_1). The fourth expression corresponds to the case that we unfold for once at both states, i.e., 1-induction principle.

- case $C \equiv \text{if } (\phi)\ \{C_1\} \text{ else } \{C_2\}$.

$$\begin{aligned} & \overline{\Phi}_f(\overline{\Psi}_h(h)) \\ = \quad & [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\overline{\Psi}_h(h)) \\ = \quad & [\neg\varphi] \cdot f + [\varphi \wedge \phi] \cdot pre_{C_1}(\overline{\Psi}_h(h)) + [\varphi \wedge \neg\phi] \cdot pre_{C_2}(\overline{\Psi}_h(h)) \end{aligned}$$

wherein

$$\begin{aligned} pre_{C_1}(\overline{\Psi}_h(h)) \quad = \quad & pre_{C_1}(\min\{[\neg\varphi] \cdot f + [\varphi] \cdot ([\phi] \cdot pre_{C_1}(h) + [\neg\phi] \cdot pre_{C_2}(h)), h\} \\ = \quad & \min\{[\neg\varphi(C_1)] \cdot pre_{C_1}(f) + [\varphi(C_1)] \cdot \\ & ([\phi(C_1)] \cdot pre_{C_1;C_1}(h) + [\neg\phi(C_1)] \cdot pre_{C_1;C_2}(h)), pre_{C_1}(h)\} \end{aligned}$$

and

$$\begin{aligned} pre_{C_2}(\overline{\Psi}_h(h)) \quad = \quad & pre_{C_2}(\min\{[\neg\varphi] \cdot f + [\varphi] \cdot ([\phi] \cdot pre_{C_1}(h) + [\neg\phi] \cdot pre_{C_2}(h)), h\} \\ = \quad & \min\{[\neg\varphi(C_2)] \cdot pre_{C_2}(f) + [\varphi(C_2)] \cdot \\ & ([\phi(C_2)] \cdot pre_{C_2;C_1}(h) + [\neg\phi(C_2)] \cdot pre_{C_2;C_2}(h)), pre_{C_2}(h)\} \end{aligned}$$

Thus we have

$$\begin{aligned} & \overline{\Phi}_f(\overline{\Psi}_h(h)) \\ = \quad & [\neg\varphi] \cdot f + [\varphi \wedge \phi] \cdot \min\{[\neg\varphi(C_1)] \cdot pre_{C_1}(f) \\ & +[\varphi(C_1)] \cdot ([\phi(C_1)] \cdot pre_{C_1;C_1}(h) + [\neg\phi(C_1)] \cdot pre_{C_1;C_2}(h)), pre_{C_1}(h)\} + \\ & [\varphi \wedge \neg\phi] \cdot \min\{[\neg\varphi(C_2)] \cdot pre_{C_2}(f) \\ & +[\varphi(C_2)] \cdot ([\phi(C_2)] \cdot pre_{C_2;C_1}(h) + [\neg\phi(C_2)] \cdot pre_{C_2;C_2}(h)), pre_{C_2}(h)\} \\ = \quad & \min\{[\neg\varphi] \cdot f + [\varphi \wedge \phi \wedge \neg\varphi(C_1)] \cdot pre_{C_1}(f) + \\ & [\varphi \wedge \neg\phi \wedge \neg\varphi(C_2)] \cdot pre_{C_2}(f) + \\ & [\varphi \wedge \phi \wedge \varphi(C_1) \wedge \phi(C_1)] \cdot pre_{C_1;C_1}(h) + [\varphi \wedge \phi \wedge \varphi(C_1) \wedge \neg\phi(C_1)] \cdot pre_{C_1;C_2}(h)) + \\ & [\varphi \wedge \neg\phi \wedge \varphi(C_2) \wedge \phi(C_2)] \cdot pre_{C_2;C_1}(h) + [\varphi \wedge \neg\phi \wedge \varphi(C_2) \wedge \neg\phi(C_2)] \cdot pre_{C_2;C_2}(h)), \\ & [\neg\varphi] \cdot f + [\varphi \wedge \phi \wedge \neg\varphi(C_1)] \cdot pre_{C_1}(f) + \\ & [\varphi \wedge \phi \wedge \varphi(C_1) \wedge \phi(C_1)] \cdot pre_{C_1;C_1}(h) + [\varphi \wedge \phi \wedge \varphi(C_1) \wedge \neg\phi(C_1)] \cdot pre_{C_1;C_2}(h)) + \\ & [\varphi \wedge \neg\phi] \cdot pre_{C_2}(h), \\ & [\neg\varphi] \cdot f + [\varphi \wedge \neg\phi \wedge \neg\varphi(C_2)] \cdot pre_{C_2}(f) + \\ & [\varphi \wedge \neg\phi \wedge \varphi(C_2) \wedge \phi(C_2)] \cdot pre_{C_2;C_1}(h) + [\varphi \wedge \neg\phi \wedge \varphi(C_2) \wedge \neg\phi(C_2)] \cdot pre_{C_2;C_2}(h)) + \\ & [\varphi \wedge \phi] \cdot pre_{C_1}(h), \\ & [\neg\varphi] \cdot f + [\varphi \wedge \phi] \cdot pre_{C_1}(h) + [\varphi \wedge \neg\phi] \cdot pre_{C_2}(h) \end{aligned}$$

The one-to-one relation is the same as that in the former case (probabilistic choice case).

Then we proof the case of $k > 2$ by mathematical induction. Suppose that the proposition holds when $k = n$, i.e., the upper n-induction condition $\overline{\Phi}_f(\overline{\Psi}_h^{n-1}(h)) \preceq h$ is equivalent with $\min\{h_1, h_2, \ldots, h_m\} \preceq h$, where each h_i uniquely corresponds to one $C_d \in \{C_1, \ldots, C_m\}$ and is equal to $pre_{C_d}(h)$, where $\{C_1, \ldots, C_m\}$ are all the loop-free programs generated by following the decision process in **Stage 2** in Section 5 within m unfolding.

Then we proof the case of $n + 1$.

$$\begin{aligned}\overline{\Phi}_f(\overline{\Psi}_h^n(h)) &= \overline{\Phi}_f(\overline{\Psi}_h(\overline{\Psi}_h^{n-1}(h))) \\ &= \overline{\Phi}_f(\min\{\overline{\Phi}_f(\overline{\Psi}_h^{n-1}(h)), h\}) \\ &= \overline{\Phi}_f(\min\{\min\{h_1, h_2, \dots, h_m\}, h\}) \\ &= \overline{\Phi}_f(\min\{h_1, h_2, \dots, h_m, h\}) \\ &= [\neg\varphi] \cdot f + [\varphi] \cdot pre_C(\min\{h_1, h_2, \dots, h_m, h\})\end{aligned}$$

Through the same inference on the structure C as above, we show it is equivalent to $\min\{g_1, g_2, \dots, g_M\}$, where $M \geq m + 1$ and each g_i uniquely corresponds to one $C_d \in \{C_1, \dots, C_M\}$ and is equal to $pre_{C_d}(h)$, where $\{C_1, \dots, C_M\}$ are all the loop-free programs generated by following the decision process in **Stage 2** in Section 5 within $n + 1$ unfolding. Thus the proposition holds when $k = n + 1$. Notice that the operators $\overline{\Phi}_f$ and pointwise min are noncommutative.

By mathematical induction, the proposition holds for $k \geq 2$. □

Remark 5. *In Proposition 5.2, We only propose the case of upper k-induction condition, and the case of lower k-induction condition is completely dual.*

C.3 Supplementary Material for the Pedagogical Explanation in Stage 2

We now present a detailed mathematical analysis of the program in (5).

Recall that we denote f as the return function, and denote $\overline{\Phi}_f$ as the function given by

$$\overline{\Phi}_f(h)(x) := [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1x + b_1) + (1 - p) \cdot h(a_2x + b_2))$$

for every function $h : \mathbb{R} \to \mathbb{R}$. We use the k-induction operator Ψ_h from [11] (k is dummy here) which is given by $\Psi_h(g) := \min\{\overline{\Phi}_f(g), h\}$. We apply the $k = 2$-induction condition to upper-bound the expected value of X_f and perform a key simplification for this condition via loop unfolding as follows. For the ease of understanding, we let $H_1 = [\neg\varphi(a_1x + b_1)] \cdot f(a_1x + b_1) + [\varphi(a_1x + b_1)] \cdot (p \cdot h(a_1(a_1x + b_1) + b_1) + (1 - p) \cdot h(a_2(a_1x + b_1) + b_2))$, which intuitively represents that we unfold the loop once at the state of $a_1x + b_1$, and $H_2 = [\neg\varphi(a_2x + b_2)] \cdot f(a_2x + b_2) + [\varphi(a_2x + b_2)] \cdot (p \cdot h(a_1(a_2x + b_2) + b_1) + (1 - p) \cdot h(a_2(a_2x + b_2) + b_2))$, which intuitively represents that we unfold the loop once at the state of $a_2x + b_2$.

- **Case 1**: In this case, the loop is executed once, reaching two states $a_1x + b_1$ and $a_2x + b_2$, and does not continue. In other words, we unfold the loop only once and obtain the loop-free program C_1 as in Fig. 2a. This amounts to $h_1 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)](p \cdot h(a_1x + b_1) + (1 - p) \cdot h(a_2x + b_2))$, which is the expected value of $h(x)$ after the execution of the program C_1.
- **Case 2**: In this case, the loop is first executed once, reaching two states $a_1x + b_1$ and $a_2x + b_2$. Then, we clarify two cases below.
  - At the state $a_1x + b_1$, we stop the execution of the loop and have the value $h(a_1x + b_1)$.
  - At the state $a_2x + b_2$, we continue the execution of the loop and obtain two branches: (i) if φ is not satisfied, we directly have the return function $f(a_2x + b_2)$; (ii) if φ is satisfied, we arrive at the states $a_1(a_2x + b_2) + b_1$ and $a_2(a_2x + b_2) + b_2$.

  The unfolding process above generates a loop-free program C_2 (see Fig. 2b), and h_2 is derived from the program C_2 in a way similar to h_1. We have that $h_2 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)] \cdot (p \cdot h(a_1x + b_1) + (1 - p) \cdot H_2)$, which is the expected value of $h(x)$ after the execution of the program C_2.

- **Case 3**: This case is similar to **Case 2**, with the only difference that we choose to continue the execution of the loop at the state $a_1x + b_1$ and do not unfold the loop at $a_2x + b_2$. Then, we clarify two cases below.
  - At the state $a_1x + b_1$, we continue the execution of the loop and we will attain two branches: (i) if φ is not satisfied, output the return function $f(a_1x + b_1)$; (ii) if φ is satisfied, we will arrive at the states $a_1(a_1x + b_1) + b_1$ and $a_2(a_1x + b_1) + b_2$.
  - At the state of $a_2x + b_2$, we stop the execution of the loop and have the value $h(a_2x + b_2)$.

  This generates a loop-free program C_3 (see Fig. 2c), from which h_3 is derived similar to h_1, h_2. We have that $h_3 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)] \cdot (p \cdot H_1 + (1 - p) \cdot h(a_2x + b_2))$, which is the expected value of $h(x)$ after the execution of the program C_3.
- **Case 4**: In this case, at both the states a_1x+b_1 and a_2x+b_2, we choose to execute the loop once more. This generates a loop-free program C_4 (see Fig. 2d). h_4 is derived from the program C_4 similar to the previous cases. We have that $h_4 = [\neg\varphi(x)] \cdot f(x) + [\varphi(x)] \cdot (p \cdot H_1 + (1-p) \cdot H_2)$, which is the expected value of $h(x)$ after the execution of the program C_4.

C.4 Supplementary Material for Stage 4

Motzkin's Transposition Theorem is a classical theorem that provides a dual characterization for the satisfiability of a system of strict and non-strict inequalities. Below we present the original Motzkin's Transposition Theorem.

Theorem C.1 (Motzkin's Transposition Theorem [43]). *Given the set of linear, and strict linear, inequalities over real-valued variables $x_1, x_2, \ldots, x_n$,*

$$S = \begin{bmatrix} \sum_{i=1}^{n} \alpha_{(1,i)} \cdot x_i + \beta_1 \leq 0 \\ \vdots \\ \sum_{i=1}^{n} \alpha_{(m,i)} \cdot x_i + \beta_m \leq 0 \end{bmatrix} \text{ and } \quad T = \begin{bmatrix} \sum_{i=1}^{n} \alpha_{(m+1,i)} \cdot x_i + \beta_{m+1} < 0 \\ \vdots \\ \sum_{i=1}^{n} \alpha_{(m+k,i)} \cdot x_i + \beta_{m+k} < 0 \end{bmatrix}$$

in which $\alpha_{(1,1)}, \ldots, \alpha_{(m+k,n)}$ and $\beta_1, \ldots, \beta_{m+k}$ are real-valued, we have that S and T simultaneously are not satisfiable (i.e., they have no solution in x) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_{m+k}$ such that either the condition (A_1):

$$0 = \textstyle\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \ldots, 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, 1 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0,$$

or condition (A_2): at least one coefficient λ_i for i in the range $\{m+1, \ldots, m+k\}$ is non-zero and

$$0 = \textstyle\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \ldots, 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, 0 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0.$$

In our work, we consider the variant form of Motzkin's Transposition Theorem (see Theorem 5.5). Theorem 5.5 is first proposed in [18, Theorem 4.5 and Remark 4.6] without proof. We give a complete proof here.

Theorem 5.5. [Corollary of Motzkin's Transposition Theorem] Let S and T be the same systems of linear inequalities as that in Theorem C.1. If S is satisfiable, then $S \wedge T$ is unsatisfiable iff there exist non-negative reals $\lambda_0, \lambda_1, \ldots, \lambda_{m+k}$ and at least one coefficient λ_i for $i \in \{m+1, \ldots, m+k\}$ is non-zero, such that:

$$0 = \textstyle\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \ldots, 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, 0 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0.$$

i.e., the condition (A_2) in Theorem C.1.

Before we proof the theorem, we introduce the desired theorem: Farkas's Lemma:

Lemma C.2 (Farkas's lemma [25]). *Consider the following system of linear inequalities over real-valued variables* $x_1, x_2, ..., x_n$,

$$S = \begin{bmatrix} \alpha_{(1,1)}x_1 & +\cdots+ & \alpha_{(1,n)}x_n & +\beta_1 & \leq 0 \\ \vdots & \vdots & \vdots & \vdots & \\ \alpha_{(m,1)}x_1 & +\cdots+ & \alpha_{(m,n)}x_n & +\beta_m & \leq 0 \end{bmatrix}$$

When S is satisfiable, it entails a given linear inequality

$$\phi : c_1x_1 + ... + c_nx_n + d \leq 0$$

if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, ..., \lambda_m$, such that

$$c_1 = \sum_{i=1}^{m} \lambda_i \alpha_{(i,1)}, ..., c_n = \sum_{i=1}^{m} \lambda_i \alpha_{(i,n)}, d = (\sum_{i=1}^{m} \lambda_i \beta_i) - \lambda_0$$

Furthermore, S is unsatisfiable if and only if the inequality $1 \leq 0$ can be derived as shown above.

Now we proof the corollary (Theorem 5.5).

Proof. Proof by contradiction. According to Motzkin's Transposition Theorem, S and T have no solution in x if and only if there exists non-negative real numbers $\lambda_0, \lambda_1, ..., \lambda_{m+k}$ such that either condition (A_1) or (A_2) is satisfied. We first proof $(\lambda_{m+1} \neq 0) \vee (\lambda_{m+2} \neq 0) \vee ... \vee (\lambda_{m+k} \neq 0)$.

If it is not satisfied, we assume that $\lambda_{m+1} = ... = \lambda_{m+k} = 0$. Then we know the condition (A_1) must be satisfied and we have (By applying the assumption $\lambda_{m+1} = ... = \lambda_{m+k} = 0$):

$$0 = \sum_{i=1}^{m} \lambda_i \alpha_{(i,1)}, ..., 0 = \sum_{i=1}^{m} \lambda_i \alpha_{(i,n)}, \sum_{i=1}^{m} \lambda_i \beta_i = \lambda_0 + 1 \geq 1,$$

By applying Farkas's Lemma, we have:

$$c_1 = \sum_{i=1}^{m} \lambda_i \alpha_{(i,1)} = 0, ..., c_n = \sum_{i=1}^{m} \lambda_i \alpha_{(i,n)} = 0, d = (\sum_{i=1}^{m} \lambda_i \beta_i) - \lambda_0 = \lambda_0 + 1 - \lambda_0 = 1,$$

Thus we have:

$$\phi = c_1x_1 + ... + c_nx_n + d = d = 1 \leq 0$$

if and only if S is not satisfiable, which contradicts the assumption, so the assumption does not hold. We have proved $(\lambda_{m+1} \neq 0) \vee (\lambda_{m+2} \neq 0) \vee ... \vee (\lambda_{m+k} \neq 0)$.

If condition (A_1) is satisfied, then exists non-negative real numbers $\lambda_0, \lambda_1, ..., \lambda_{m+k}$ and $(\lambda_{m+1} \neq 0) \vee (\lambda_{m+2} \neq 0) \vee ... \vee (\lambda_{m+k} \neq 0)$(what we just prove) such that

$$0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, ..., 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, 1 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0,$$

let $\lambda_0' = \lambda_0 + 1 \geq 0$ and we can find that it also satisfies the condition (A_2), that is $A_1 \implies A_2$. Thus, Motzkin's Transposition Theorem can be simplified as: If S is satisfiable, then S and T have no solution in x if and only if there exists non-negative real numbers $\lambda_0, \lambda_1, ..., \lambda_{m+k}$, such that:

$$((A_1 \vee A_2) \wedge (A_1 \implies A_2)) \iff A_2$$

Thus we prove Theorem 5.5. □

C.5 Application of Putinar's Positivstellensatz [46]

We recall Putinar's Positivstellensatz below.

Theorem C.3 (Putinar's Positivstellensatz [46]). *Let V be a finite set of real-valued variables and $g, g_1, \ldots, g_m \in \mathbb{R}[V]$ be polynomials over V with real coefficients. Consider the set $\mathcal{S} := \{\mathbf{x} \in \mathbb{R}^V \mid g_i(\mathbf{x}) \geq 0$ for all $1 \leq i \leq m\}$ which is the set of all real vectors at which every g_i is non-negative. If (i) there exists some g_k such that the set $\{\mathbf{x} \in \mathbb{R}^V \mid g_k(\mathbf{x}) \geq 0\}$ is compact and (ii) $g(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathcal{S}$, then we have that*

$$g = f_0 + \textstyle\sum_{i=1}^{m} f_i \cdot g_i \tag{15}$$

for some polynomials $f_0, f_1 \ldots, f_m \in \mathbb{R}[V]$ such that each polynomial f_i is the a sum of squares (of polynomials in $\mathbb{R}[V]$), i.e. $f_i = \sum_{j=0}^{k} q_{i,j}^2$ for polynomials $q_{i,j}$'s in $\mathbb{R}[V]$.

In our comparison, we utilize the sound form in (15) for witnessing a polynomial g to be non-negative over a semi-algebraic set P for each inductive constraint $\forall x \in P, g(x) \geq 0$.

In our experiments, the maximum degree of unknown SOS polynomials is set to the degree of the polynomial template plus 2.

D SUPPLEMENTARY MATERIAL FOR SECTION 6

D.1 Continued Fraction

Continued fraction can represent a real number r by an expression as follows:

$$r = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

and r is abbreviated as $[a_0, a_1, a_2, \ldots]$. In our implementation, we first transform each output float coefficient into its continued fraction form $[a_0, a_1, a_2, \ldots]$. Then we perform the truncation operation that we find the first $a_i (i \geq 1)$ that is greater than a large threshold, for which we choose 100, and truncate from there (including this number). We keep only the previous parts, as our rational approximation results.

D.2 Experimental Results of Piecewise Linear Lower Bounds

We present the experimental results of piecewise linear lower bounds in this section. For the linear lower bounds, we consider the same benchmarks and return functions f as in Section 6.1, and use the same invariant from the *External Inputs* for each benchmark.

Answering RQ1. We present the experimental results for the synthesis of piecewise linear lower bounds on the 13 benchmarks in Table 5. In this table, we only show the piecewise results with $(k \leq 3)$-induction. We observe that on most of the benchmarks, we can obtain a linear lower bound via the conventional approach, i.e., 1-induction, while the piecewise linear lower bounds we synthesize are better (tighter) with $(k > 1)$-induction. Only on the benchmark Growing Walk-variant, we require $(k > 1)$-induction to synthesize a lower bound. Moreover, our k-induction-based approach can produce results within a few minutes.

Answering RQ2. We answer RQ2 by comparing our approach with the most related approaches [5, 10] in Table 5. The relevant explanations for **RQ2** in Table 5 are totally the same to Table 1. These two relevant works require a (possibly piecewise) lower bound to be verified as an additional program input and return a sub-invariant that is sufficient to *verify* the input lower bound, which is the most different aspect from our work. cegispro2 produce the results by a proof rule derived from the original OST (see Section 6 in [10] and Appendix B.1), while we apply an extended OST (see Theorem 4.6). To have a richer comparison, we also feed our benchmarks paired with

Table 5. Experimental Results for **RQ1** and **RQ2**, Linear Case (Lower Bounds). "f" stands for the return function considered in the benchmark, "T(s)" (of our approach) stands for the execution time of our approach (in seconds), including the parsing from the program input, transforming the k-induction constraint into the bilinear problems, bilinear solving time and verification time. "Conventional Approach ($k = 1$)" stands for the monolithic linear upper bound synthesized via 1-induction, "k" stands for the k-induction we apply, "Solution" stands for the linear candidate solved by Gurobi, and "Piecewise Linear Upper Bound" stands for our piecewise results. "Result" stands for the synthesized results by other approaches and "T(s)" (of their approaches) stands for the execution time of their tools.

Benchmark	f	**Conventional Approach** ($k = 1$)		**Our Approach**				CEGISPRO2		EXIST	
		Result	**T(s)**	k	**Solution**	**Piecewise Linear Lower Bound**	**T(s)**	**Result**	**T(s)**	**Result**	**T(s)**
Geo	x	x	0.33	3	x	$[c > 0] \cdot x + [c \leq 0] \cdot (x + \frac{3}{4})$	2.19	$[c > 0] \cdot x +$ $[c \leq 0] \cdot (x + \frac{3}{4})$	0.06	$x + [c = 0]$	83.01
k-Geo	y	y	100.18	3	y	$[k > N] \cdot y + [k \leq N] \cdot$ $(0.75x + y + 0.25)$	133.81	$[k > N] \cdot y +$ $[k \leq N] \cdot$ $(-k + N + x + y + 1)$	0.2	$y + [k \leq n] \cdot$ $(0.8x - 0.3k$ $+0.3n + 0.5)$	239.95
Bin-ran	y	$-0.5i + y + 5$	1.18	2	$-\frac{21}{29} * i + y$ $+\frac{210}{29}$	$[i > 10] \cdot y + [1 < i \leq 10] \cdot$ $(-\frac{21}{29} i + y + \frac{9}{20} x + \frac{1068}{145})$	106.59	$[i > 10] \cdot y + [i \leq 10] \cdot$ $(\frac{9}{19} * x + y -$ $\frac{53059}{112955} * i + \frac{154900}{22591})$	0.26	fail	-
Coin	i	i	100.51	2	i	$[y \neq x] \cdot i + [y = x] \cdot (i + \frac{13}{8})$	5.99	$[y \neq x] \cdot i +$ $[y = x] \cdot (i + \frac{13}{8})$	0.07	$i + [x = y] \cdot 2.2$	116.67
Mart	i	i	0.37	3	i	$[x \leq 0] \cdot i + [x > 0] \cdot (i + 1.5)$	2.44	violation of non-negativity	-	$i + [x > 0] * 2$	122.93
GrowingWalk	y	$x + y$	100.16	3	$x + y$	$[x < 0] \cdot y + [x \geq 0] \cdot (x + y + \frac{5}{4})$	101.80	violation of non-negativity	-	fail	-
GrowingWalk -variant	y	-	-	3	$y - 1$	$[x < 0] \cdot y +$ $[0 \leq x < 1] \cdot (y + 0.5x - 1) +$ $[1 \leq x < 2] \cdot (y + 0.5x - 1.5)$ $+[2 \leq x] \cdot (y + 0.75x - 2)$	125.53	violation of non-negativity	-	fail	-
Expected Time	t	$1.1111x + t$	0.25	3	$1.240x + t$	$[x < 0] \cdot t +$ $[0 \leq x < 1] \cdot (0.124x + t + 0.9)$ $[1 \leq x \leq 10] \cdot (1.1284x + t + 1.9116)$	125.54	violation of non-negativity	-	fail	-
Zero-Conf -variant	cur	cur	100.32	3	cur	$[est > 0] \cdot cur +$ $[start == 0 \wedge est \leq 0] \cdot (cur + 1.9502)$ $+[start \geq 1 \wedge est \leq 0] \cdot (cur + 0.287)$	183.63	violation of non-negativity	-	inner error	-
Equal-Prob-Grid	$goal$	$goal$	100.38	2	$goal$	$[a > 10 \vee b > 10 \vee goal \neq 0] \cdot goal$ $[a \leq 10 \wedge b \leq 10 \wedge goal = 0] \cdot 1.5$	139.80	$[a > 10 \vee b > 10 \vee$ $goal \neq 0] \cdot goal +$ $[a \leq 10 \wedge b \leq 10$ $\wedge goal = 0] \cdot 1.5$	0.1	inner error	-
RevBin	z	$z + 2x - 2$	100.14	3	$z + 2x - 2$	$[x < 1] \cdot z +$ $[1 \leq x < 2] \cdot (z + x)$ $+[x \geq 2] \cdot (z + 2x - 2)$	129.46	$[x < 1] \cdot z +$ $[x \geq 1] \cdot (z + 2x - 2)$	0.11	$z + [x > 0]$ $\cdot 2x$	122.85
Fair Coin	i	i	100.34	3	i	$[x > 0 \vee y > 0] \cdot i +$ $[x = 0 \wedge y = 0] \cdot (i + \frac{5}{4})$	43.84	$[x > 0 \vee y > 0] \cdot i +$ $[x = 0 \wedge y = 0]$ $\cdot (i + \frac{5}{4})$	0.06	$i + [x + y = 0]$ $\cdot 1.3$	82.67
St-Petersburg variant	y	y	0.32	3	y	$[x > 0] \cdot y + [x \leq 0] \cdot \frac{11}{8} y$	2.28	$[x > 0] \cdot y +$ $[x \leq 0] \cdot \frac{11}{8} y$	0.21	$y + [x = 0]$ $\cdot 0.4y$	98.05

the piecewise lower bounds synthesized by our approach to CEGISPRO2. On 5 of our benchmarks (e.g., Growing Walk-variant, Zero-Conf-variant, etc), it reports failure (violation of non-negativity). On 6 of our benchmarks, CEGISPRO2 produce the same results with our inputs. Only on two benchmarks (k-Geo, Bin-Ran) CEGISPRO2 produce a different result to verify our inputs.

For the comparison with EXIST, we note that EXIST synthesizes sub-invariants without the application of OST, which might be unsound for proving the input lower bounds (see also Section 7 in [10]). We compare with their tool on our benchmarks by assuming the soundness of their lower bounds and feed them our piecewise lower bounds as an additional program input. On benchmarks Geo, k-Geo, Coin, RevBin, Mart, Fair Coin, St-Petersburg variant, their tool can generate a tighter sub-invariant to *verify* our piecewise lower bound. On these benchmarks, due to the existence of exact invariants, they are usually able to find a tighter sub-invariant by a heuristic search based on sampling and machine learning at the cost of the long time (usually about or even more than 100s) . For the remaining benchmarks, either they cannot generate sub-invariants or there are internal errors within their tool.

In conclusion, our approaches can handle many benchmarks that these two works [5, 10] cannot handle. When feeding our benchmarks with the bounds synthesized through our approach to CEGISPRO2 and EXIST, they fail on about 40% of our benchmarks. Over most of the benchmarks that CEGISPRO2 and our approach can handle, our bounds are comparable with theirs. Over most of the benchmarks that EXIST and our approach can handle, they spend much more time to generate a slightly tighter bound.

Answering RQ3. Similarly to the upper case, we compare our piecewise linear lower bounds with monolithic polynomial lower bounds synthesized via 1-induction, as shown in Table 6. From the comparison result "PCT" in Table 6, we observe that on most of our benchmarks , our piecewise linear lower bounds are significantly tighter (i.e., greater) than monolithic polynomial lower bounds.

D.3 Experimental Results of Piecewise Polynomial Lower Bound

In this section, we present the experimental results of piecewise polynomial lower bounds. For the piecewise polynomial lower bounds, we consider the same benchmarks and return functions f as in Section 6.2, and use the same invariant as the *External Inputs* for each benchmark.

Answering RQ1. We present the experimental results for the synthesis of piecewise polynomial lower bounds on the 20 benchmarks in Table 7. The experimental results show that our approach can compute piecewise polynomial lower bounds for most of the benchmarks within around 10 seconds. Especially, on the benchmarks BIN0, BIN2, DEPRV, SUM0, PRINSYS, the lower bounds we obtain are the same with the upper bounds we obtain in Section 6.2 (see Table 3 for more details), which shows that we obtain the exact expected value of X_f after the execution of the loop, i.e., the tightest lower bounds, on these 5 benchmarks.

Answering RQ2. We answer RQ2 by comparing our approach with the relevant work EXIST in Table 7. Since their tool requires a lower bound to be verified as an extra program input, we feed them our lower bounds (the column "Solution h^*" in Table 7) synthesized by our approach. Over these benchmarks, they only successfully synthesize a sub-invariant to verify our lower bounds on PRINSYS and the sub-invariant they generate is the same as our piecewise lower bound. For the benchmarks BIN0, BIN2,SUM0, they can learn some candidates for sub-invariants but they are not able to verify them so that they fail to generate a sub-invariant. For the other 16 benchmarks, they fail to generate due to some inner errors within their tool.

Answering RQ3. Similarly to the upper case, we compare our piecewise polynomial lower bounds with higher degree monolithic polynomial lower bounds synthesized via 1-induction, as shown in Table 8. For a fair comparison, we generate the polynomial bounds with the same invariant and optimal objective function for each benchmark. The degree of monolithic polynomial bounds is also set to be not greater than 5 in this experiment.

From the comparison results "PCT", We show that on all the benchmarks except BRP, FIG-6, CAV-5, our piecewise polynomial bounds are significantly tighter than monolithic polynomial bounds. Although our running time is also a bit longer than that of monolithic polynomial experiments, our approach allows to synthesize lower-degree polynomials while achieving better precision against higher-degree polynomials. This advantage is critical as the synthesis of higher-degree polynomials suffers from a large amount of numerical errors as stated previously. Thus our approach has a value to use lower-degree piecewise polynomials to surpass the numerical problem of higher-degree polynomials.

Table 6. Experimental Results for **RQ3**, Linear Case (Lower Bounds). "f" stands for the return function considered in the benchmark, "k" stands for the k-induction condition we apply in this comparison, "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, and "d" stands for the degree of polynomial template we use, "PCT" stands for the percentage of the points that our piecewise lower bound are lower (i.e., not better) than monolithic polynomial.

Benchmark	f	**Our Approach**		**Monolithic Polynomial via 1-induction**		**PCT**
		k	**Piecewise Linear Lower Bound**	**d**	**Monolithic Polynomial Lower Bound**	
Geo	x	4	$[c > 0] \cdot x +$ $[c \le 0] \cdot (x + \frac{7}{8})$	3	$-0.0313 - 0.1902 * c + 1.0478 * x -$ $0.3980 * c^2 + 0.0695 * x * c - 0.0019 * x^2 -$ $0.1595 * x * c^2 + 0.07227 * x^2 * c - 0.0147 * x^3$	0.0%
k-Geo	y	3	$[k > N] \cdot y +$ $[k \le N] \cdot (0.75x + y + 0.25)$	2	$44.6223 * N - 221.2813 - 0.7791 * k + 1.0000 * y$ $+0.9281 * x - 2.1922 * N^2 - 0.1043 * x^2$	4.19%
Bin-ran	y	2	$[i > 10] \cdot y +$ $[1 < i \le 10] \cdot (-\frac{21}{29} i + y + \frac{9}{20} x + \frac{1068}{145})$	3	$-22.0746 - 24.4593 * i + 33.7063 * y +$ $20.7709 * x + 1.4945 * i^2 + 0.2057 * y * i +$ $0.0232 * y^2 + 0.4741 * x * i + 0.2689 * x * y +$ $1.9807 * x^2 + 0.0006 * i^3 - 0.3133 * y * i^2 -$ $0.0111 * y^2 * i + 0.0049 * y^3 - 0.4668 * x * i^2 +$ $0.0036 * x * y * i + 0.0105 * x * y^2 - 0.7437 * x^3$ $+0.04213 * x^2 * y - 0.7531 * x^2 * i$	33.39%
Coin	i	4	$[y \ne x] \cdot i + [y = x] \cdot (i + \frac{129}{64})$	2	$2.6655 + 1.0002 * i - 3622.3830 * y -$ $5419.0667 * x - 0.0001 * i^2 + 0.0007 * y * i +$ $3619.71553 * y^2 - 0.0008 * x * i +$ $1827.4383 * x * y + 3594.2952 * x^2$	2.0%
Mart	i	4	$[x \le 0] \cdot i +$ $[x > 0] \cdot (i + \frac{7}{4})$	2	$1.0000 * i + 39.9996 * x - 199.9958 * x^2$	1.0%
Growing Walk	y	4	$[x < 0] \cdot y +$ $[x \ge 0] \cdot (x + y + \frac{13}{8})$	3	$-0.0004 + 1.0003 * y + 1.3463 * x -$ $0.0001 * y^2 - 0.0010 * x * y - 0.0590 * x^2$ $+0.0007 * x^2 * y - 0.0022 * x^3$	0.0%
Growing Walk variant	y	3	$[x < 0] \cdot y +$ $[0 \le x < 1] \cdot (0.5x + y - 1)$ $+[1 \le x < 2] \cdot (0.5x + y - 1.5)$ $+[2 \le x] \cdot (0.75x + y - 2)$	3	$-1.0000 + 1.0000 * y - 0.3903 * x -$ $0.0734 * y^2 + 0.0484 * x * y + 0.4758 * x^2 -$ $0.0250 * x * y^2 - 0.0484 * x^2 * y - 0.0855 * x^3$	0.01%
Expected Time	t	3	$[x < 0] \cdot t +$ $[0 \le x < 1] \cdot (0.124x + t + 0.9) +$ $[1 \le x \le 10] \cdot (1.1284x + t + 1.9116)$	3	$-0.0784 + 1.0093 * t + 3.1426 * x -$ $0.0010 * t^2 + 0.0083 * x * t - 0.1576 * x^2 +$ $0.0002 * x * t^2 + 0.0002 * x^2 * t + 0.0043 * x^3$	64.6 %
Zero-Conf -variant	cur	3	$[est > 0] \cdot cur +$ $[start == 0 \land est \le 0] \cdot (cur + 1.9502)$ $+[start \ge 1 \land est \le 0] \cdot (cur + 0.287)$	2	$140.2458 + 1.0098 * cur - 424365.5964 * start -$ $587675.0179 * est - 0.0066 * start * cur +$ $424267.3602 * start^2 - 0.0095 * est * cur -$ $504437.5495 * est * start + 587534.7143 * est^2$	0.64%
Equal-Prob-Grid	$goal$	2	$[a > 10 \lor b > 10 \lor goal \ne 0] \cdot goal$ $+[a \le 10 \land b \le 10 \land goal = 0] \cdot 1.5$	2	$0.4950 * goal - 0.2020 * goal^2 +$ $0.0053 * b * goal - 0.0011 * a * goal$	0.0%
RevBin	z	3	$[x < 1] \cdot z +$ $[1 \le x < 2] \cdot (z + x)$ $+[x \ge 2] \cdot (z + 2x - 2)$	2	$-2.0000 + 1.0000 * z + 2.0000 * x$	0.0%
Fair Coin	i	4	$[x > 0 \lor y > 0] \cdot i +$ $[x \le 0 \land y \le 0] \cdot (i + \frac{21}{16})$	2	$1.0000 * i - 0.3932 * y - 0.39325 * x$ $-0.3153 * i^2 + 0.6305 * y * i - 0.7242 * y^2$ $+0.6305 * x * i - 0.1796 * x * y - 0.7242 * x^2$	0.0%
St-Petersburg variant	y	3	$[x > 0] \cdot y + [x \le 0] \cdot \frac{11}{8} y$	3	$-0.0017 + 1.0023 * y - 121479.0179 * x$ $-0.0550 * x * y + 121479.0185 * x^2$	0.0%

D.4 Full Expressions for Experimental Results

For readability and conciseness, some of the experimental results in the main text were partially omitted and denoted with $\cdots$. In this appendix, we provide the complete expressions corresponding to those abbreviated entries.

Piecewise polynomial upper bound of grid-small: $\min\{[a < 10 \land b < 10] \cdot (-0.0003 * a^3 - 0.0011 * b^3 - 0.0008 * a^2 * b + 0.0018 * a * b^2 + 0.0109 * a^2 - 0.0144 * a * b + 0.0129 * b^2 - 0.0926 * a + 0.0277 * b + 0.5109) + [a < 10 \land b \ge 10], h^*\}$.

Table 7. Experimental Results for **RQ1** and **RQ2**, Polynomial Case (Lower Bounds). "f" stands for the return function considered in the benchmark, "T(s)" stands for the execution time of our approach (in seconds), including the parsing procedure from the program input, relaxing the k-induction constraint into the SDP problems, the SDP solving time and verification time. "d" stands for the degree of polynomial template we use and "Solution h^*" is the candidate polynomial solved directly by the solver. "Piecewise Polynomial upper Bound" stands for the piecewise bound we synthesize, where h^* is the column "Solution h^*". "Sub-invariant" stands for the sub-invariant synthesized by EXIST, and "T(s)" stands for the execution time of their tool.

Benchmark	f	**Our Approach**				**EXIST**	
		d	**Solution h^***	**T(s)**	**Piecewise Polynomial lower Bound**	**Sub-invariant**	**T(s)**
GeoAr	x	2	$-0.0467*y^2+0.8036*y*z-7.1202*z^2+x+0.6668*y+10.2222*z-2.3795$	7.22	$\max\{[z>0]\cdot(-0.0467y^2+0.4018*y*z-3.5601*z^2+x+1.0734*y+5.5129*z-1.2594)+[z\leq 0]\cdot x, h^*\}$	inner error	-
Bin0	x	2	$x+0.5*y*n$	10.04	$x+[n>0]\cdot 0.5*y*n$	fail	-
Bin2	x	2	$0.25*n+x+0.25*n^2+0.5*y*n$	10.25	$x+[n>0]\cdot(0.25*n+x+0.25*n^2+0.5*y*n)$	fail	-
DepRV	$x*y$	2	$-0.25*n+0.25*n^2+0.5*y*n+0.5*x*n+x*y$	9.08	$[n>0]\cdot(-0.25*n+0.25*n^2+0.5*y*n+0.5*x*n+x*y)+[n\leq 0]\cdot x*y$	inner error	-
Prinsys	$[x==1]$	2	0	2.10	$[x==1]*1+[x==0]*0.5$	$[x==1]*1+[x==0]*0.5$	7.29
Sum0	x	2	$0.25*i^2+0.25*i+x$	1.98	$[i>0]*(0.25*i^2+0.25*i)+x$	fail	-
Duel	t	2	$21.7319*x^2-0.4706*x*t+1.3703*t^2-21.7099*x-0.3707*t-0.0011$	6.66	$\max\{[t>0\wedge x\geq 1]\cdot(10.8660x^2+0.2353*x*t+1.3703*t^2-11.0903*x-1.3703*t+0.4987)+[t\leq 0\wedge x\geq 1]\cdot(5.4330*x^2+0.1177*x*t+1.3703*t^2-5.5451*x-0.8705*t+0.2488)+[x<1]\cdot t, h^*\}$	inner error	-
BRP	$[failed=10]$	2	$-41834.4189*failed^2-6.0771*failed*sent-0.8349*sent^2-1710.0678*failed+655.2652*sent+2695.5257$	9.85	$\max\{[failed<10\wedge sent<800]\cdot(-418.3442*failed^2-0.0608*failed*sent-0.8349*sent^2-853.7891failed+653.5513sent+2907.9668)+[failed=10], h^*\}$	inner error	-
CHAIN	$[y=1]$	2	$-0.0001*x*y-0.0052*y^2+0.0032*x+0.0173*y-0.0347$	4.09	$\max\{[y=0\wedge x<100]\cdot(-0.0001*x*y-0.0051*y^2+0.0032*x+0.0170*y-0.0314)+[y=1], h^*\}$	inner error	-
GRID SMALL	$[a<10\wedge b\geq 10]$	3	$0.0006*a^3-0.0012*a^2*b+0.0008*a*b^2-0.0071*a^2+0.008*a*b-0.0056*b^2-0.046*a+0.0822*b+0.4185$	6.75	$\max\{[a<10\wedge b<10]\cdot(0.0006*a^3-0.0012*a^2*b+0.0008*a*b^2-0.0068*a^2+0.0076*a*b-0.0052*b^2-0.0478*a+0.0800*b+0.4306)+[a<10\wedge b\geq 10], h^*\}$	inner error	-
GRID BIG	$[a<1000\wedge b\geq 1000]$	2	$-0.0231*a^2+0.0462*a*b-0.0231*b^2-0.1895*a+0.2425*b+0.9503$	7.21	$\max\{[a<1000\wedge b<1000]\cdot(-0.0231*a^2+0.0462*a*b-0.0231*b^2-0.1895*a+0.2425b+0.9537)+[a<1000\wedge b\geq 1000], h^*\}$	inner error	-
CAV-2	$[h>1+t]$	3	$0.0001*h^3-0.0003*h^2*t+0.0003*h*t^2-0.0001*t^3+0.0018*h^2-0.0057*h*t+0.0032*t^2-0.002*h+0.054*t-0.6863$	3.45	$\max\{[t\geq h]\cdot(0.0001*h^3-0.0003*h^2*t+0.0003*h*t^2-0.0001*t^3+0.0023*h^2-0.0066*h*t+0.0037*t^2+0.0076*h+0.0399*t-0.5852)+[h>1+t], h^*\}$	inner error	-
CAV-4	$[x\leq 10]$	2	$-0.0148*x^2-0.0597*x*y+0.3443*y^2+0.0523*x-0.3282*y+0.9537$	2.47	$\max\{[y\geq 1]\cdot(-0.0148*x^2-0.0072*x+0.9694)+[y<1\wedge x\leq 10], h^*\}$	inner error	-
FIG-6	$[y\leq 5]$	4	$0.0001*x^4-0.0007*x^3*y+0.0009*x^2*y^2-0.0006*x*y^3-0.0011*x^3+0.0143*x^2*y-0.0035*x*y^2+0.0032*y^3+0.0556*x^2-0.1077*x*y+0.0085*y^2-0.3753*x+0.1362*y+0.5438$	109.28	$\max\{[x\leq 4]\cdot(0.0001*x^4-0.0007*x^3*y+0.0009*x^2*y^2-0.0006*x*y^3-0.0014*x^3+0.0140*x^2*y-0.0035*x*y^2+0.0026*y^3+0.0690*x^2-0.0960*x*y+0.0173*y^2-0.3696*x+0.1229*y+0.5508)+[x>4\wedge y\leq 5], h^*\}$	inner error	-
FIG-7	$[x\leq 1000]$	2	$-0.0002*x*y-0.0029*y^2+0.0038*y*i-0.0009*i^2+0.0002*x-0.0037*y+0.002*i+0.9978$	21.38	$\max\{[y\leq 0]\cdot(-0.0009*i^2+0.0002*x+0.0021*i+0.997)+[y>0\wedge x\leq 1000], h^*\}$	inner error	-
INV-Pend VARIANT	$[pA\leq 1]$	3	$0.0008*pAD^2*pA-0.0023*pAD^2*cV+0.0991*pAD^2*cP+0.4931*pAD*pA^2+0.1464*pAD*pA*cV\cdots-5.002*cV*cP-44.9405*cP^2-5.7109*cV+1.0$	436.04	$\max\{[cP>0.5\vee pA<-0.1\vee cP<-0.5\vee pA>0.1]\cdot(0.0011*pAD^2*pA+0.0011*pAD^2*cV+\cdots+0.999*cV-0.0688*cP+1.6061)+[cP\leq 0.5\wedge pA\leq 0.1\wedge cP\geq -0.5\wedge cP\leq 0.5], h^*\}$	inner error	-
CAV-7	$[x\leq 30]$	3	$0.0001*i^3-0.0002*i^2*x+0.0001*i*x^2-0.0006*i^2+0.001*i*x-0.0002*x^2+0.0007*i-0.0005*x+0.9981$	5.17	$\max\{[i<5]\cdot(-0.0001*i^2*x-0.0001*i^2+0.0006*i*x-0.0001*x^2+0.0003*i+0.0001*x+0.9985)+[i\geq 5\wedge x\leq 30], h^*\}$	inner error	-
CAV-5	$[i\geq 10]$	3	$0.0009*i^2*money+0.0043*i*money^2+0.0013*money^3-0.9614*i^2-17.8117*i*money-66.2212*money^2-29.2611*i+1.0$	897.32	$\max\{[money\geq 10]\cdot(0.0009*i^2*money+0.0043*i*money^2+0.0013*money^3-0.9624*i^2-17.8205*i*money-66.2275*money^2-12.8062*i+118.2861*money-1379.4033)+[money<10\wedge i\leq 10], h^*\}$	inner error	-
Add	$[x>5]$	3	$-0.0002*x^3+0.002*x^2*y-0.0092*x*y^2+0.0088*y^3+0.0049*x^2-0.0267*x*y+0.0425*y^2+0.0167*x-0.1369*y+0.0314$	3.74	$\max\{[y\leq 1]\cdot(0.0088*x^3-0.0092*x^2*y+0.002*x*y^2-0.0002*y^3+0.0618*x^2-0.0406*x*y+0.0064*y^2-0.0527*x-0.0102*y-0.0328)+[y>1\wedge x>5], h^*\}$	inner error	-
GrowingWalk Variant2	y	2	$-0.0055*x^2-0.0013*x*y-0.0132*x*r-0.0027*y^2+0.0123*y*r-0.0261*r^2+0.0288*x+1.0125*y+0.0111*r-0.0454$	4.83	$\max\{[r\leq 0]\cdot(-0.0075*x^2-0.004*x*y-0.0027*y^2+0.5230*x+1.0174*y-0.0362)+[r>0]\cdot y, h^*\}$	inner error	-

Table 8. Experimental Results for **RQ3**, Polynomial Case (Lower Bounds). "f" stands for the return function considered in the benchmark, "Piecewise Polynomial Lower Bound" stands for the results synthesized by our algorithm. "Monolithic Polynomial via 1-Induction" stands for the monolithic polynomial bounds synthesized via 1-induction, "T(s)" stands for the total execution time. "PCT" stands for the percentage of the points that our piecewise polynomial lower bound are larger (i.e., not better) than (higher degree) monolithic polynomial.

Benchmark	f	**Our Approach**			**Monolithic Polynomial via 1-induction**			**PCT**
		d	**T(s)**	**Piecewise Polynomial lower Bound**	**d**	**T(s)**	**Monolithic Polynomial lower Bound**	
GeoAr	x	2	7.22	$\max\{[z>0]\cdot(-0.0467y^2+0.4018*y*z$ $-3.5601*z^2+x+1.0734*y+5.5129*z$ $-1.2594)+[z\le 0]\cdot x, h^*$	3	1.25	$0.0021*x^2*z+0.0003*x*y^2+0.0128*x*y*z$ $-0.0745*x*z^2-0.0013*y^3+0.0006*y^2*z$ $-0.9011*y*z^2-34359.8787*z^3-0.0001*x^2$ $-0.0028*x*y+0.0294*x*z+0.0154*y^2$ $+1.9735*y*z+68717.1029*z^2+1.0025*x$ $-0.0476*y-34355.4581*z-0.0973$	5.0%
Bin0	x	2	10.04	$x+[n>0]\cdot 0.5*y*n$	3	0.81	$0.5*y*n+x$	0.0%
Bin2	x	2	10.25	$x+[n>0]\cdot(0.25*n+$ $x+0.25*n^2+0.5*y*n)$	3	1.14	$-0.0001*y^3+0.0001*y^2*n+0.0001*y*n^2$ $+0.0006*y^2+0.4992*y*n+0.25*n^2+x$ $-0.0021*y+0.249*n-0.0028$	16.2%
DepRV	$x*y$	2	9.08	$[n>0]\cdot(-0.25*n+0.25*n^2+0.5*y*n$ $+0.5*x*n+x*y)+[n\le 0]\cdot x*y$	3	0.83	$x*y+0.5*x*n+0.5*y*n+$ $0.25*n^2-0.2501*n-0.0001$	5.4%
Prinsys	$[x==1]$	2	2.10	$[x==1]*1+[x==0]*0.5$	3	0.45	0.0	0.0%
Sum0	x	2	1.98	$[i>0]*(0.25*i^2+0.25*i)+x$	4	0.50	$0.25*i^2+0.25*i+x$	0.0%
Duel	t	2	7.24	$\max\{[t>0\wedge x\ge 1]\cdot(10.8660x^2+$ $0.2353*x*t+1.3703*t^2-11.0903*x$ $-1.3703*t+0.4987)+[t\le 0\wedge x\ge 1]\cdot$ $(5.4330*x^2+0.1177*x*t+1.3703*t^2$ $-5.5451*x-0.8705*t+0.2488)$ $+[x<1]\cdot t, h^*\}$	4	0.58	$57.6107*x^4-0.3086*x^3*t+32.5537*x^2*t^2$ $-0.9734*x*t^3-8.9958*t^4+31.3993*x^3$ $-17.8531*x^2*t+10.7254*x*t^2+26.343*t^3$ $-27.2812*x^2-24.7154*x*t$ $+13.3805*t^2-61.5859*x-29.7278*t$	0.02%
BRP	$[failed=10]$	2	9.85	$\max\{[failed<10\wedge sent<800]\cdot$ $(-418.3442*failed^2-0.0608*failed*sent-$ $0.8349*sent^2-853.7891failed+653.5513sent$ $+2907.9668)+[failed=10], h^*\}$	4	1.24	$-5.1928*failed^4-0.992*failed^3*sent-$ $0.0002*failed^2*sent^2-1.6946*failed^3+$ $2.1022*failed^2*sent+0.0001*failed*sent^2-$ $3.3782failed^2-1.0916failed*sent-0.0057sent^2$ $-2.09*failed+1.1127*sent+0.7991$	53.54%
chain	$[y=1]$	2	4.09	$\max\{[y=0\wedge x<100]\cdot(-0.0001*x*y$ $-0.0051*y^2+0.0032*x+$ $0.0170*y-0.0314)+[y=1], h^*\}$	3	0.74	$0.0429*x^3+0.6155*x^2*y+12.0075*x*y^2$ $+124904.4081*y^3-5.4506*x^2$ $-67.0765*x*y+869301.3767*y^2+$ $119.3344*x-994786.2786*y+4.4144$	1.00%
grid small	$[y=1]$	3	6.75	$\max\{[a<10\wedge b<10]\cdot(0.0006*a^3-$ $0.0012*a^2*b+0.0008*a*b^2-0.0068*a^2$ $+0.0076*a*b-0.0052*b^2-0.0478*a+$ $0.0800*b+0.4306)+[a<10\wedge b\ge 10], h^*\}$	4	0.90	$-0.0002*a^2*b+0.0001*a*b^2+$ $0.0002*a^2-0.0007*a*b+0.0004*b^2$ $-0.0371*a+0.0364*b+0.4437$	0.62%
grid big	$[a<1000\wedge$ $b\ge 1000]$	2	7.24	$\max\{[a<1000\wedge b<1000]\cdot(-0.0231*a^2+$ $0.0462*a*b-0.0231*b^2-0.1895*a+$ $0.2425b+0.9537)+[a<1000\wedge b\ge 1000], h^*\}$	3	0.56	$0.001*a^3-0.0005*a^2*b-0.0018*a*b^2$ $+0.0008*b^3-2.9594*a^2+5.9103*a*b-$ $2.9631b^2-499.9807a+511.5109b+253.0223$	4.73%
cav-2	$[h>t+1]$	3	3.45	$\max\{[t\ge h]\cdot(0.0001*h^3-0.0003*h^2*t+$ $0.0003*h*t^2-0.0001*t^3+0.0023*h^2$ $-0.0066*h*t+0.0037*t^2+0.0076*h+$ $0.0399*t-0.5852)+[h>1+t], h^*\}$	4	0.47	$0.0001*h^4+0.0001*h^3*t-0.0009*h^2*t^2$ $+0.0013*h*t^3-0.0007*t^4-0.0062*h^3$ $+0.0306*h^2*t-0.0831*h*t^2+0.0777*t^3$ $-0.1378*h^2+1.2065*h*t-1.9084*t^2$ $-6.4628*h+21.3167*t-92.5531$	33.33%
cav-4	$[x\le 10]$	2	2.47	$\max\{[y\ge 1]\cdot(-0.0148*x^2-0.0072*x+$ $0.9694)+[y<1\wedge x\le 10], h^*\}$	3	0.34	$-0.0017*x^3-0.0105*x^2*y-0.0514*x*y^2+$ $11.376*y^3+0.0085*x^2+0.0539*x*y-$ $5.0983*y^2-0.0103*x-6.2928*y+1.0$	0.76%
fig-6	$[y\le 5]$	4	109.28	$\max\{[x\le 4]\cdot(0.0001*x^4-0.0007*x^3*y+$ $0.0009*x^2*y^2-0.0006*x*y^3-0.0014*x^3$ $+0.0140*x^2*y-0.0035*x*y^2+0.0026*y^3$ $+0.0690*x^2-0.0960*x*y+0.0173*y^2$ $-0.3696*x+0.1229*y+0.5508)$ $+[x>4\wedge y\le 5], h^*\}$	5	0.94	$0.0002*x^5+0.0001*x^4*y+0.0001*x^2*y^3$ $-0.0001*x*y^4-0.0021*x^4-0.0033*x^3*y$ $+0.001*x^2*y^2-0.0016*x*y^3$ $-0.0001*y^4+0.0072*x^3+0.033*x^2*y$ $-0.008*x*y^2+0.0017*y^3+0.0817*x^2$ $-0.2069*x*y+0.0347*y^2-0.8681*x$ $+0.5271*y+0.5958$	40.77%
fig-7	$[x\le 1000]$	2	21.38	$\max\{[y\le 0]\cdot(-0.0009*i^2+0.0002*x$ $+0.0021*i+0.997)+[y>0\wedge x\le 1000], h^*\}$	3	2.40	$0.0616*x^2*y-0.0002*x^2*i-47.1183*x*y^2$ $-0.4059*x*y*i+0.014*x*i^2-3529.0989*y^3$ $+23.9641*y^2*i+2.4655*y*i^2-0.38*i^3$ $-0.0401*x^2+43.7495*x*y+0.318*x*i$ $+86697.7958*y^2-25.0167*y*i-0.5461*i^2$ $+3.2993*x-83167.42*y-0.0624*i-5.0013$	2.37%
inv-Pend	$[pA\le 1]$	3	436.04	$\max\{[cP>0.5\vee pA<-0.1\vee cP<-0.5\vee$ $pA>0.1]\cdot(0.0011*pAD^2*pA+$ $0.0011*pAD^2*cV+\cdots+0.999*cV$ $-0.0688*cP+1.6061)+[cP\le 0.5$ $\wedge pA\le 0.1\wedge cP\ge -0.5\wedge cP\le 0.5], h^*\}$	4	6.71	$-0.2235*pAD^4-1.1293*pAD^3*pA+$ $0.1015*pAD^3*cV+0.1091*pAD^3*cP-$ $5.2183*pAD^2*pA^2+\cdots-10.4965*cP^2+$ $0.0001*pA-53.2106*cV+1.0$	1.18%
CAV-7	$[x\le 30]$	3	5.17	$\max\{[i<5]\cdot(-0.0001*i^2*x-0.0001*i^2$ $+0.0006*i*x-0.0001*x^2+0.0003*i+$ $0.0001*x+0.9985)+[i\ge 5\wedge x\le 30], h^*\}$	4	0.78	$-0.0007*i^4+0.001*i^3*x-0.0005*i^2*x^2+$ $0.0001*i*x^3+0.0044*i^3-0.0052*i^2*x$ $+0.0011*i*x^2-0.0134*i^2+0.0121*i*x$ $-0.0019*x^2+0.0128*i-0.004*x+0.9966$	25.83%

Benchmark	f	Our Approach			Monolithic Polynomial via 1-induction			PCT
		d	T(s)	Piecewise Polynomial lower Bound	d	T(s)	Monolithic Polynomial lower Bound	
CAV-5	$[i \geq 10]$	3	897.32	$\max\{[money \geq 10] \cdot (0.0009 * i^2 * money + 0.0043 * i * money^2 + 0.0013 * money^3 - 0.9624 * i^2 - 17.8205 * i * money - 66.2275 * money^2 - 12.8062 * i + 118.2861 * money - 1379.4033) + [money < 10 \wedge i \leq 10], h^*\}$	4	1.08	$-0.0001 * i^2 * money^2 - 0.0004 * i * money^3 - 0.0002 * money^4 - 0.001 * i^3 + 0.0222 * i^2 - 0.0257 * i^2 * money + 0.0526 * i * money^2 + 0.0298 * money^3 - 0.4528 * i * money - 4.1462 * money^2 - 3.6304 * i + 1.0$	50.0%
ADD	$[x > 5]$	3	3.74	$\max\{[y \leq 1] \cdot (0.0088 * x^3 - 0.0092 * x^2 * y + 0.002 * x * y^2 - 0.0002 * y^3 + 0.0618 * x^2 - 0.0406 * x * y + 0.0064 * y^2 - 0.0527 * x - 0.0102 * y - 0.0328) + [y > 1 \wedge x > 5], h^*\}$	5	0.57	$-0.3566 * x^5 - 2.2831 * x^4 * y + 3.1151 * x^3 * y^2 - 0.2365 * x^2 * y^3 - 0.5919 * x * y^4 + 0.5847 * y^5 + 4.2396 * x^4 - 7.1539 * x^3 * y + 1.3284 * x^2 * y^2 - 2.0416 * x * y^3 + 1.1293 * y^4 + 0.0868 * x^3 - 0.2857 * x^2 * y + 6.5688 * x * y^2 - 6.7823 * y^3 + 2.7185 * x^2 + 1.5398 * x * y + 3.6667 * y^2 - 6.7017 * x + 1.4053 * y + 0.0001$	32.66%
GrowingWalk Variant2	y	2	4.83	$\max\{[r \leq 0] \cdot (-0.0075 * x^2 - 0.004 * x * y - 0.0027 * y^2 + 0.5230 * x + 1.0174 * y - 0.0362) + [r > 0] \cdot y, h^*\}$	3	1.09	$-0.0013 * x^3 + 0.0006 * x^2 * y - 0.0026 * x^2 * r + 0.0001 * x * y^2 + 0.0082 * x * y * r + 3.1974 * x * r^2 + 0.001 * y^2 * r - 1.1091 * y * r^2 - 19675.0498 * r^3 - 0.0103 * x^2 + 0.0017 * x * y - 4.1484 * x * r - 0.0057 * y^2 + 1.0965 * y * r + 39349.552 * r^2 + 1.048 * x + 1.0165 * y - 19675.6056 * r + 0.8489$	5.03%

Monolithic polynomial upper bound of GeoAr: $-0.0001*x^3+0.0001*x^2*y-0.0011*x^2*z-0.0004*x*y^2-0.0112*x*y*z+0.164*x*z^2+0.0012*y^3+0.0046*y^2*z-1.8186*y*z^2+89866.1344*z^3+0.0027*x*y-0.1236*x*z-0.0137*y^2+2.7194*y*z-179731.0721*z^2+0.9993*x+0.0417*y+89867.2768*z+0.078$.

Solution h^* of Fig-6: $-0.0001*x^4+0.0011*x^3*y-0.001*x^2*y^2+0.0008*x*y^3-0.0001*y^4+0.0016*x^3-0.0195*x^2*y+0.006*x*y^2-0.003*y^3-0.0627*x^2+0.1018*x*y-0.0028*y^2+0.5712*x-0.281*y+0.6009$.

Piecewise polynomial upper bound of Fig-6: $\min\{[x \leq 4] \cdot (-0.0001 * x^4 + 0.0011 * x^3 * y - 0.001 * x^2 * y^2 + 0.0008 * x * y^3 - 0.0001 * y^4 + 0.0023 * x^3 - 0.0182 * x^2 * y + 0.0064 * x * y^2 - 0.0026 * y^3 - 0.0788 * x^2 + 0.0913 * x * y - 0.0094 * y^2 + 0.5530 * x - 0.2782 * y + 0.6027) + [x > 4 \wedge y \leq 5], h^*\}$.

Monolithic polynomial upper bound of Fig-6: $-0.0001*x^5-0.0002*x^4*y-0.0003*x^2*y^3+0.0001*x*y^4-0.0002*y^5+0.0011*x^4+0.0037*x^3*y-0.0008*x^2*y^2+0.0021*x*y^3+0.0005*y^4-0.0012*x^3-0.0361*x^2*y+0.0088*x*y^2-0.0042*y^3-0.084*x^2+0.1432*x*y+0.0064*y^2+0.9708*x-0.6526*y+0.575$.

Monolithic polynomial upper bound of Fig-7: $-0.083*x^2*y+0.0003*x^2*i+48.5638*x*y^2+0.5267*x*y*i-0.018*x*i^2+2600.9691*y^3-36.705*y^2*i-2.646*y*i^2+0.4053*i^3+0.0539*x^2-45.1036*x*y-0.4109*x*i-58912.9534*y^2+37.7582*y*i+0.6223*i^2-3.3923*x+56310.8279*y-0.0114*i+7.2868$.

Solution (upper) h^* of inv-Pend variant: $0.0058*pAD^2*pA+0.0023*pAD^2*cV-0.1313*pAD^2*cP-0.6278*pAD*pA^2-0.2352*pAD*pA*cV-4.2984*pAD*pA*cP+0.0034*pAD*cV^2-0.0776*pAD*cV*cP+0.2901*pAD*cP^2-3.3499*pA^3+1.2174*pA^2*cV-18.4697*pA^2*cP+0.8063*pA*cV^2+7.4278*pA*cV*cP+2.1607*pA*cP^2+0.1664*cV^3+0.0048*cV^2*cP-0.5863*cV*cP^2-101.7368*cP^3+0.7678*pAD^2+4.7849*pAD*pA-0.1664*pAD*cV-3.5565*pAD*cP+28.2784*pA^2-2.7311*pA*cV-20.9853*pA*cP-1.1597*cV^2+5.9637*cV*cP+60.4194*cP^2-0.0002*pA+7.1495*cV+0.001*cP+1.0$.

Piecewise polynomial upper bound of inv-Pend variant: $\min\{[cp > 0.5 \vee cp < -0.5 \vee pA > 0.1 \vee pA < -0.1] \cdot (0.0058 * pAD^2 * pA - 0.0011 * pAD^2 * cV - 0.1313 * pAD^2 * cP - 0.6279 * pAD * pA^2 - 0.2408 * pAD * pA * cV - 4.2984 * pAD * pA * cP - 0.0124 * pAD * cV^2 - 0.0021 * pAD * cV * cP + 0.2901 * pAD * cP^2 - 3.3498 * pA^3 + 0.4776 * pA^2 * cV - 18.4697 * pA^2 * cP + 0.4734 pA * cV^2 + 5.5455 * pA * cV * cP + 2.1607 * pA * cP^2 + 0.1014 * cV^3 - 0.0334 * cV^2 * cP - 3.4879 * cV * cP^2 - 101.7368 * cP^3 + 0.5916 * pAD^2 + 4.0443 * pAD * pA + 0.0057 * pAD * cV - 3.5023 * pAD * cP + 26.6426 * pA^2 - 1.1436 * pA * cV - 20.7584 * pA * cP - 0.5132 * cV^2 + 5.5468 * cV * cP + 60.3921 * cP^2 - 0.4489 * pAD - 1.5038 * pA + 5.2348 * cV + 0.0688 * cP + 0.3238) + [-0.5 \leq cp \leq 0.5 \wedge -0.1 \leq pA \leq 0.1], h^*\}$.

Monolithic polynomial upper bound of inv-Pend variant: $0.2264 * pAD^4 + 1.1448 * pAD^3 * pA - 0.1026 * pAD^3 * cV - 0.1107 * pAD^3 * cP + 5.2869 * pAD^2 * pA^2 - 0.4937 * pAD^2 * pA * cV - 0.8938 * pAD^2 * pA * cP + 0.3036 * pAD^2 * cV^2 + 0.0478 * pAD^2 * cV * cP + 0.4208 * pAD^2 * cP^2 + 6.8201 * pAD *$

$pA^3 - 3.2518 * pAD * pA^2 * cV - 2.3942 * pAD * pA^2 * cP + 1.3927 * pAD * pA * cV^2 + 0.7868 * pAD * pA * cV * cP + 4.5143 * pAD * pA * cP^2 - 0.1912 * pAD * cV^3 - 0.1023 * pAD * cV^2 * cP - 0.1906 * pAD * cV * cP^2 - 2.8734 * pAD * cP^3 + 53.6801 * pA^4 + 1.323 * pA^3 * cV - 6.8123 * pA^3 * cP + 5.2663 * pA^2 * cV^2 + 2.473 * pA^2 * cV * cP + 47.9517 * pA^2 * cP^2 - 0.5451 * pA * cV^3 - 0.7983 * pA * cV^2 * cP - 0.9821 * pA * cV * cP^2 - 20.6044 * pA * cP^3 + 0.0986 * cV^4 + 0.0333 * cV^3 * cP + 0.3483 * cV^2 * cP^2 + 4.5559 * cV * cP^3 + 30.7504 * cP^4 - 0.3716 * pAD^3 + 3.8 * pAD^2 * pA + 0.4985 * pAD^2 * cV - 0.0606 * pAD^2 * cP - 9.7537 * pAD * pA^2 - 6.8904 * pAD * pA * cV + 0.4876 * pAD * pA * cP - 0.748 * pAD * cV^2 - 0.1874 * pAD * cV * cP - 3.858 * pAD * cP^2 - 11.7619 * pA^3 + 18.5549 * pA^2 * cV - 0.4732 * pA^2 * cP + 4.6011 * pA * cV^2 - 0.8554 * pA * cV * cP - 9.3429 * pA * cP^2 + 1.9127 * cV^3 + 0.0857 * cV^2 * cP + 5.2916 * cV * cP^2 - 3.7534 * cP^3 + 4.1573 * pAD^2 + 17.8582 * pAD * pA + 0.2247 * pAD * cV - 2.5151 * pAD * cP + 34.1498 * pA^2 + 1.7805 * pA * cV - 5.4381 * pA * cP - 10.9045 * cV^2 + 1.196 * cV * cP + 10.6625 * cP^2 - 0.0001 * pA + 53.8573 * cV + 1.0.$

Solution (lower) h^* of INV-PEND VARIANT: $0.0008 * pAD^2 * pA - 0.0023 * pAD^2 * cV + 0.0991 * pAD^2 * cP + 0.4931 * pAD * pA^2 + 0.1464 * pAD * pA * cV + 3.1026 * pAD * pA * cP - 0.0138 * pAD * cV^2 + 0.0529 * pAD * cV * cP - 0.2253 * pAD * cP^2 + 2.4945 * pA^3 - 1.1719 * pA^2 * cV + 13.2225 * pA^2 * cP - 0.66 * pA * cV^2 - 5.8813 * pA * cV * cP - 1.6276 * pA * cP^2 - 0.1308 * cV^3 - 0.0137 * cV^2 * cP - 0.2948 * cV * cP^2 + 70.0898 * cP^3 - 0.6053 * pAD^2 - 3.6586 * pAD * pA + 0.2164 * pAD * cV + 2.8831 * pAD * cP - 20.411 * pA^2 + 2.3725 * pA * cV + 16.7142 * pA * cP + 0.9814 * cV^2 - 5.002 * cV * cP - 44.9405 * cP^2 - 5.7109 * cV + 1.0.$

Piecewise polynomial lower bound of INV-PEND VARIANT: $\max\{[cp > 0.5 \vee cp < -0.5 \vee pA > 0.1 \vee pA < -0.1] \cdot (0.0011 * pAD^2 * pA + 0.0012 * pAD^2 * cV + 0.0983 * pAD^2 * cP + 0.5052 * pAD * pA + 0.1661 * pAD * pA * cV + 3.1298 * pAD * pA * cP + 0.0035 * pAD * cV^2 - 0.0028 * pAD * cV * cP - 0.1801 * pAD * cP^2 + 2.5528 * pA^3 - 0.5263 * pA^2 * cV + 13.3751 * pA^2 * cP - 0.3872 * pA * cV^2 - 4.3942 * pA * cV * cP - 1.4142 * pA * cP^2 - 0.0803 * cV^3 + 0.0045 * cV^2 * cP + 1.8662 * cV * cP^2 + 70.0747 * cP^3 - 0.4694 * pAD^2 - 3.0860 * pAD * pA + 0.0414 * pAD * cV + 2.8487 * pAD * cP - 19.2280 * pA^2 + 0.9998 * pA * cV + 16.5897 * pA * cP + 0.4500 * cV^2 - 4.5457 * cV * cP - 44.9216 * cP^2 + 0.3480 * pAD + 1.1902 * cV^2 - 4.5456 * cV * cP - 44.9216 * cP^2 + 0.3480 * pAD + 1.1903 * pA + 0.999 * cV - 0.0688 * cP + 1.606) + [-0.5 \le cp \le 0.5 \wedge -0.1 \le pA \le 0.1], h^*\}.$

Monolithic polynomial lower bound of INV-PEND VARIANT: $-0.2235 * pAD^4 - 1.1293 * pAD^3 * pA + 0.1015 * pAD^3 * cV + 0.1091 * pAD^3 * cP - 5.2183 * pAD^2 * pA^2 + 0.4869 * pAD^2 * pA * cV + 0.8825 * pAD^2 * pA * cP - 0.3 * pAD^2 * cV^2 - 0.0472 * pAD^2 * cV * cP - 0.4159 * pAD^2 * cP^2 - 6.7225 * pAD * pA^3 + 3.227 * pAD * pA^2 * cV + 2.3658 * pAD * pA^2 * cP - 1.3787 * pAD * pA * cV^2 - 0.7789 * pAD * pA * cV * cP - 4.4659 * pAD * pA * cP^2 + 0.189 * pAD * cV^3 + 0.1012 * pAD * cV^2 * cP + 0.189 * pAD * cV * cP^2 + 2.8456 * pAD * cP^3 - 53.0957 * pA^4 - 1.3037 * pA^3 * cV + 6.7519 * pA^3 * cP - 5.2076 * pA^2 * cV^2 - 2.4518 * pA^2 * cV * cP - 47.4808 * pA^2 * cP^2 + 0.54 * pA * cV^3 + 0.7896 * pA * cV^2 * cP + 0.9738 * pA * cV * cP^2 + 20.4078 * pA * cP^3 - 0.0975 * cV^4 - 0.033 * cV^3 * cP - 0.3448 * cV^2 * cP^2 - 4.5124 * cV * cP^3 - 30.4568 * cP^4 + 0.3659 * pAD^3 - 3.7578 * pAD^2 * pA - 0.4937 * pAD^2 * cV + 0.06 * pAD^2 * cP + 9.654 * pAD * pA^2 + 6.8253 * pAD * pA * cV - 0.4846 * pAD * pA * cP + 0.7351 * pAD * cV^2 + 0.1847 * pAD * cV * cP + 3.8138 * pAD * cP^2 + 11.671 * pA^3 - 18.344 * pA^2 * cV + 0.4762 * pA^2 * cP - 4.5653 * pA * cV^2 + 0.8466 * pA * cV * cP + 9.2637 * pA * cP^2 - 1.8886 * cV^3 - 0.0833 * cV^2 * cP - 5.2318 * cV * cP^2 + 3.7165 * cP^3 - 4.1093 * pAD^2 - 17.6591 * pAD * pA - 0.204 * pAD * cV + 2.4836 * pAD * cP - 33.745 * pA^2 - 1.6881 * pA * cV + 5.3915 * pA * cP + 10.7733 * cV^2 - 1.1846 * cV * cP - 10.4965 * cP^2 + 0.0001 * pA - 53.2106 * cV + 1.0.$

E BENCHMARK PROGRAMS

This section presents the benchmark programs used in our experiments, along with the invariants employed in our algorithms. In addition, we show the results of checking the prerequisites of Theorems 4.10 and 4.11(P2), as discussed in Section 6.

E.1 Programs in Linear Experiments

This section contains the benchmark programs in our linear experiments, i.e., in Tables 1 and 5.

Example E.1 (Geo).

$$C_{\text{Geo}}: \quad \texttt{while}\,(\,0 \le c\,)\,\{ \\ \quad \{c := 1\}\ [0.5]\ \{x := x+1\} \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \le x$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.2 (k-geo).

$$C_{\text{k-geo}}: \quad \texttt{while}\,(\,k \le N\,)\,\{ \\ \quad \{k := k+1; y := y+x; x := 0\}\ [0.5]\ \{x := x+1\} \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \wedge 0 \le y \wedge k \le N+1$, and synthesize dbRSM $k - N$.

Example E.3 (Binomial-random).

$$C_{\text{Bin-ran}}: \quad \texttt{while}\,(\,i \le 10\,)\,\{ \\ \quad \{x := x+1\}\ [0.5]\ \{x := 0\} \\ \quad \{y := y+x; i := i+1\}\ [0.9]\ \{y := y+1; i := 0\} \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \le i \le 11 \wedge 0 \le x \wedge 0 \le y$, and there is a probability $p \ge 0.9^{10}$ that the program will terminate immediately for every ten loop iterations.

Example E.4 (Coin).

$$C_{\text{Coin}}: \quad \texttt{while}\,(\,x = y\,)\,\{ \\ \quad \{x := 0\}\ [3/4]\ \{x := 1\} \\ \quad \{y := 0\}\ [3/4]\ \{y := 1\} \\ \quad i := i+1; \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \le i \wedge 0 \le x \le 1 \wedge 0 \le y \le 1$, and every loop iteration terminates directly with probability $p = \frac{5}{8}$.

Example E.5 (Martingale).

$$C_{\text{Mart}}: \quad \texttt{while}\,(\,0 < x\,)\,\{ \\ \quad \{y := y+x; x := 0\}\ [0.5]\ \{y := y-x; x := 2*x\} \\ \quad i := i+1; \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \le x$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.6 (Growing Walk).

$$C_{\text{Growing Walk}}: \quad \texttt{while}\,(\,0 \le x\,)\,\{ \\ \quad \{x := x+1; y := y+x\}\ [0.5]\ \{x := -1\} \\ \}$$

In this probabilistic program, we take the **invariant** $I = -1 \leq x$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.7 (Growing Walk variant1).

$$
\begin{aligned}
C_{\text{Growing Walk1}}: \quad & \texttt{while}\,(\,0 \leq x\,)\,\{ \\
& \quad \{x := x - 1; y := y + x\}\ [0.5]\ \{x := 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = -1 \leq x$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.8 (Expected Time).

$$
\begin{aligned}
C_{\text{Expected Time}}: \quad & \texttt{while}\,(\,0 \leq x\,)\,\{ \\
& \quad \{x := x - 1; t := t + 1\}\ [0.9]\ \{x := 10; t := t + 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = -1 \leq x \leq 10$, and there is a probability $p \geq 0.9^{10}$ that the program will terminate immediately for every ten loop iterations.

Example E.9 (Zero Conference variant).

$$
\begin{aligned}
C_{\text{Zero-Conf-Var}}: \quad & \texttt{while}\,(\,\mathit{established} \leq 0 \land \mathit{start} \leq 1\,)\,\{ \\
& \quad \texttt{if}\,(\,\mathit{start} \geq 1\,)\,\{ \\
& \quad\quad \{\mathit{start} := 0\}\ [0.3]\ \{\mathit{start} := 0; \mathit{established} := 1\,\}\,\} \\
& \quad \texttt{else}\,\{\,\{\mathit{curprobe} := \mathit{curprobe} + 1\}\ [0.99]\ \{\mathit{start} := 1; \mathit{curprobe} := \mathit{curprobe} - 1\}\,\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq \mathit{start} \leq 1 \land 0 \leq \mathit{est} \leq 1$, and for the prerequisite (P2) checking, when $\mathit{start} = 1$, the loop iteration terminates directly with probability $p = 0.7$. When $\mathit{start} = 0$, the value of start has the probability of 0.01 to become 1 and turn to the branch of $\mathit{start} = 1$.

Example E.10 (Equal Probability Grid Family).

$$
\begin{aligned}
C_{\text{Equal-Prob-Grid-Family}}: \quad & \texttt{while}\,(\,a \leq 10 \land b \leq 10 \land \mathit{goal} = 0\,)\,\{ \\
& \quad \texttt{if}\,(\,b \geq 10\,)\,\{ \\
& \quad\quad \{\mathit{goal} := 1\}\ [0.5]\ \{\mathit{goal} := 2\}\,\} \\
& \quad \texttt{else}\,\{ \\
& \quad\quad \texttt{if}\,(\,a \geq 10\,)\,\{ \\
& \quad\quad\quad a := a - 1\,\} \\
& \quad\quad \texttt{else}\,\{ \\
& \quad\quad\quad \{a := a + 1\}\ [0.5]\ \{b := b + 1\}\,\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq a \leq 10 \land 0 \leq b \leq 10 \land \mathit{goal} \geq 0$, and we synthesize dbRSM $10 - b$.

Example E.11 (RevBin).

$$
\begin{aligned}
C_{\text{RevBin}}:\quad & \texttt{while}\,(\,1 \le x\,)\,\{\\
& \quad \{x := x - 1; z := z + 1\}\ [0.5]\ \{z := z + 1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x$, and we synthesize dbRSM x.

Example E.12 (Fair Coin).

$$
\begin{aligned}
C_{\text{Fair Coin}}:\quad & \texttt{while}\,(\,x \le 0 \land y \le 0\,)\,\{\\
& \quad \{x := 0\}\ [0.5]\ \{x := 1; i := i + 1\}\\
& \quad \{y := 0\}\ [0.5]\ \{y := 1; i := i + 1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \le 1 \land 0 \le y \le 1$, and every loop iteration terminates directly with probability $p = 0.25$.

Example E.13 (Bernoulli's St. Petersburg Paradox variant).

$$
\begin{aligned}
C_{\text{St. Petersburg1}}:\quad & \texttt{while}\,(\,x \le 0\,)\,\{\\
& \quad \{x := 1\}\ [0.75]\ \{y := 2 * y\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \le 1 \land y \le 0$, and every loop iteration terminates directly with probability $p = 0.75$.

E.2 Programs in Polynomial Experiments

This section contains the benchmarks in our polynomial experiments, i.e., in Tables 3 and 7.

Example E.14 (GeoAr).

$$
\begin{aligned}
C_{\text{GeoAr}}:\quad & \texttt{while}\,(\,0 < z\,)\,\{\\
& \quad y := y + 1;\\
& \quad \{x := x + y\}\ [0.9]\ \{z := 0\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \land 0 \le y \land 0 \le z$, and every loop iteration terminates directly with probability $p = 0.1$.

Example E.15 (Bin0).

$$
\begin{aligned}
C_{\text{Bin0}}:\quad & \texttt{while}\,(\,n > 0\,)\,\{\\
& \quad \{x := x + y; n := n - 1\}\ [0.5]\ \{n := n - 1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \land 0 \le y \land 0 \le n$, and synthesize dbRSM n.

Example E.16 (Bin2).

$$
\begin{aligned}
C_{\text{Bin2}}:\quad & \texttt{while}\,(\,n > 0\,)\,\{\\
& \quad \{x := x + 1; n := n - 1\}\ [0.5]\ \{x := x + y; n := n - 1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq x \wedge 0 \leq y \wedge 0 \leq n$, and we synthesize dbRSM n.

*Example E.17 (*DepRV*).*

$$
\begin{aligned}
C_{\text{DepRV}}: \quad & \texttt{while}\,(\,n > 0\,)\,\{\\
& \quad \{x := x+1; n := n-1\}\ [0.5]\ \{y := y+1; n := n-1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq x \wedge 0 \leq y \wedge 0 \leq n$, and synthesize dbRSM n.

*Example E.18 (*Prinsys*).*

$$
\begin{aligned}
C_{\text{Prinsys}}: \quad & \texttt{while}\,(\,x = 0\,)\,\{\\
& \quad \{x := 0\}\ [0.5]\ \{\{x := -1\}\ [0.5]\ \{x := 1\}\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = -1 \leq x \leq 1$, and every loop iteration terminates directly with probability $p = 0.5$.

*Example E.19 (*Sum0*).*

$$
\begin{aligned}
C_{\text{Sum0}}: \quad & \texttt{while}\,(\,n > 0\,)\,\{\\
& \quad \{x := x+n; n := n-1\}\ [0.5]\ \{n := n-1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = i \geq 0$, and synthesize dbRSM n.

*Example E.20 (*Duel Boy*).*

$$
\begin{aligned}
C_{\text{Duel}}: \quad & \texttt{while}\,(\,x \geq 1)\,\{\\
& \quad \texttt{if}\,(\,t > 0\,)\,\{\\
& \quad\quad \{x := 0\}\ [0.5]\ \{t := 1-t\}\\
& \quad \}\texttt{else}\{\{x := 0\}\ [0.75]\ \{t := 1-t\}\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq x \leq 1 \wedge 0 \leq t \leq 1$, and every loop iteration terminates directly with probability $p \geq 0.5$.

*Example E.21 (*brp*).*

$$
\begin{aligned}
C_{\text{brp}}: \quad & \texttt{while}\,(\,sent < 800 \wedge failed < 10\,)\,\{\\
& \quad \{sent := sent+1; failed = 0\}\ [0.99]\ \{failed := failed+1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq failed \leq 10 \wedge 0 \leq sent \leq 800$, and there is a probability $p = 0.01^{10}$ that the program will terminate immediately for every ten loop iterations.

*Example E.22 (*chain*).*

$$
\begin{aligned}
C_{\text{chain}}: \quad & \texttt{while}\,(\,y \leq 0 \wedge x < 100\,)\,\{\\
& \quad \{y := 1\}\ [0.01]\ \{x := x+1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq x \leq 100 \wedge 0 \leq y \leq 1$, and every loop iteration terminates directly with probability $p = 0.01$.

Example E.23 (grid-small).

$$
\begin{aligned}
C_{\text{grid-small}}: \quad & \texttt{while}\,(\,a < 10 \wedge b < 10\,)\,\{ \\
& \quad \{a := a + 1\}\ [0.5]\ \{b := b + 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq a \leq 11 \wedge 0 \leq b \leq 11$, and synthesize dbRSM $19 - (a + b)$.

Example E.24 (grid-big).

$$
\begin{aligned}
C_{\text{grid-big}}: \quad & \texttt{while}\,(\,a < 1000 \wedge b < 1000\,)\,\{ \\
& \quad \{a := a + 1\}\ [0.5]\ \{b := b + 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq a \leq 1001 \wedge 0 \leq b \leq 1001$, and synthesize dbRSM $1999 - (a + b)$.

Example E.25 (cav-2).

$$
\begin{aligned}
C_{\text{cav-2}}: \quad & \texttt{while}\,(\,h \leq t\,)\,\{ \\
& \quad \{h := h + 10\}\ [0.25]\ \{\texttt{skip}\}; \\
& \quad \{t := t + 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq t \wedge 0 \leq h \wedge h \geq t + 1$, and synthesize dbRSM $t - h$.

Example E.26 (cav-4).

$$
\begin{aligned}
C_{\text{cav-4}}: \quad & \texttt{while}\,(\,y \geq 1\,)\,\{ \\
& \quad \{y := 1\}\ [0.5]\ \{y := 0\}; \\
& \quad \{x := x + 1\} \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq y \leq 1 \wedge x \geq 0$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.27 (fig-6).

$$
\begin{aligned}
C_{\text{fig-6}}: \quad & \texttt{while}\,(\,x \leq 4\,)\,\{ \\
& \quad \{x := x - 1\}\ [0.5]\ \{x := x + 3\}; \\
& \quad \{\texttt{skip}\}\ [0.3333]\ \{\{y := y + 1\}\ [0.5]\ \{y := y + 2\}\}; \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $y \geq 0 \wedge x \leq 7$, and synthesize dbRSM $4 - x$.

Example E.28 (fig-7)*.*

$$C_{\text{fig-7}}: \quad \texttt{while}\,(\,y \leq 0\,)\,\{ \\ \quad \{y := 0\}\ [0.5]\ \{y := 1\}; \\ \quad x := 2 * x; \\ \quad i := i + 1; \\ \}$$

In this probabilistic program, we take the **invariant** $I = i \geq 0 \land x > 0 \land 0 \leq y \leq 1$, and every loop iteration terminates directly with probability $p = 0.5$.

Example E.29 (inv-Pend variant)*.*

$$C_{\text{inv-Pend variant}}: \quad \texttt{while}\,(\,exitcond \leq 0\,)\,\{ \\ \quad \texttt{if}\,(-0.5 \leq cP)\,\{ \\ \quad\quad \texttt{if}\,(cP \leq 0.5)\,\{ \\ \quad\quad\quad \texttt{if}\,(-0.1 \leq pA)\,\{ \\ \quad\quad\quad\quad \texttt{if}\,(pA \leq 0.1)\,\{ \\ \quad\quad\quad\quad\quad exitcond := 1; \\ \quad\quad\quad\quad \}\texttt{else}\{\texttt{skip}\} \\ \quad\quad\quad \}\texttt{else}\{\texttt{skip}\} \\ \quad\quad \}\texttt{else}\{\texttt{skip}\} \\ \quad \}\texttt{else}\{\texttt{skip}\} \\ \\ \quad cP := cP + 0.01 * cV; \\ \quad \{cV := 0.02 * cP + 0.5 * cV - 0.3 * pA - 0.06 * pAD - 1\}\ [0.5] \\ \quad \{cV := 0.02 * cP + 0.5 * cV - 0.3 * pA - 0.06 * pAD + 1\}; \\ \quad pA := pA + 0.01 * pAD; \\ \quad \{pAD := 0.04 * cP + 0.07 * cV - 0.51 * pA + 0.85 * pAD - 0.8\}\ [0.5] \\ \quad \{pAD := 0.04 * cP + 0.07 * cV - 0.51 * pA + 0.85 * pAD + 0.8\}; \\ \}$$

In this probabilistic program, we take the **invariant** $I = cV \geq 0$, and synthesize a dbRSM $0.7747 * cP^2 + 0.0004 * cV^2 + 0.0222 * pA^2 + 0.0005 * pAD^2 + 0.0298 * cP * cV - 0.0919 * cP * pA - 0.0168 * cP * pAD - 0.0019 * cV * pA - 0.0003 * cV * pAD + 0.0014 * pA * pAD$ (cut to 10^{-4} precision).

Example E.30 (CAV-7)*.*

$$C_{\text{CAV-7}}: \quad \texttt{while}\,(\,i \leq 4\,)\,\{ \\ \quad \{x := x + 1; i := i + 1\}\ [1 - 0.2 * i]\ \{x := x + 1\} \\ \}$$

In this probabilistic program, we take the **invariant** $I = 0 \leq i \leq 5 \land 0 \leq x$, and synthesize dbRSM $-i$.

Example E.31 (cav-5).

$$
\begin{aligned}
C_{\text{cav-5}}:\quad & \texttt{while}\,(\,10 \le \mathit{money}\,)\,\{\\
& \quad \{\mathit{bet} := 5\}\ [0.5]\ \{\mathit{bet} := 10\};\\
& \quad \mathit{money} := \mathit{money} - \mathit{bet};\\
& \quad \mathit{bank\_guard}\ :\approx \text{Uniform}(0.0, 1.0)\\
& \quad \texttt{if}\,(\mathit{bank\_guard} \le 0.94737)\,\{\\
& \quad\quad \mathit{col1\_guard}\ :\approx \text{Uniform}(0.0, 1.0);\\
& \quad\quad \texttt{if}\,(\mathit{col1\_guard} \le 0.33333)\,\{\\
& \quad\quad\quad \mathit{flip\_guard1}\ :\approx \text{Uniform}(0.0, 1.0);\\
& \quad\quad\quad \texttt{if}\,(\mathit{flip\_guard1} \le 0.5)\,\{\\
& \quad\quad\quad\quad \mathit{money} := \mathit{money} + 1.5 * \mathit{bet};\\
& \quad\quad\quad \}\texttt{else}\{\mathit{money} := \mathit{money} + 1.1 * \mathit{bet};\}\\
& \quad\quad \}\texttt{else}\{\\
& \quad\quad\quad \mathit{col2\_guard}\ :\approx \text{Uniform}(0.0, 1.0);\\
& \quad\quad\quad \texttt{if}\,(\mathit{col2\_guard1} \le 0.5)\,\{\\
& \quad\quad\quad\quad \mathit{flip\_guard2}\ :\approx \text{Uniform}(0.0, 1.0);\\
& \quad\quad\quad\quad \texttt{if}\,(\mathit{flip\_guard2} \le 0.33333)\,\{\\
& \quad\quad\quad\quad\quad \mathit{money} := \mathit{money} + 1.5 * \mathit{bet};\\
& \quad\quad\quad\quad \}\texttt{else}\{\mathit{money} := \mathit{money} + 1.1 * \mathit{bet};\}\\
& \quad\quad\quad \}\texttt{else}\{\\
& \quad\quad\quad\quad \mathit{flip\_guard3}\ :\approx \text{Uniform}(0.0, 1.0);\\
& \quad\quad\quad\quad \texttt{if}\,(\mathit{flip\_guard3} \le 0.66667)\,\{\\
& \quad\quad\quad\quad\quad \mathit{money} := \mathit{money} + 0.3 * \mathit{bet};\\
& \quad\quad\quad\quad \}\texttt{else}\{\texttt{skip}\}\\
& \quad\quad\quad \}\\
& \quad\quad \}\\
& \quad \}\\
& \quad i := i + 1\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le i \wedge -1 \le \mathit{money}$, and synthesize dbRSM $\mathit{money} - 10$.

Example E.32 (add).

$$
\begin{aligned}
C_{\text{add}}:\quad & \texttt{while}\,(\,y \le 1\,)\,\{\\
& \quad \{y := y + 1\}\ [0.2]\ \{x := x + 1\}\\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \le x \wedge 0 \le y \le 2$, and synthesize dbRSM $1 - y$.

Example E.33 (Growing Walk Variant2).

$$
\begin{aligned}
C_{\text{Growing Walk Variant2}}: \quad & \texttt{while}\,(\,r \leq 0\,)\,\{ \\
& \quad \{r := 0\}\ [0.5]\ \{r := 1\}; \\
& \quad \{y := y + x * r; \\
& \quad \{x := x + 1; \\
& \}
\end{aligned}
$$

In this probabilistic program, we take the **invariant** $I = 0 \leq x \wedge 0 \leq y \wedge 0 \leq r \leq 1$, and every loop iteration terminates directly with probability $p = 0.5$.