Bounded Expectations: Resource Analysis for Probabilistic Programs

Van Chan Ngo
Carnegie Mellon University
Pittsburgh, PA, USA
channgo@cmu.edu

Quentin Carbonneaux
Yale University
New Haven, CT, USA
quentin.carbonneaux@yale.edu

Jan Hoffmann
Carnegie Mellon University
Pittsburgh, PA, USA
jhoffmann@cmu.edu

Abstract

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

Keywords probabilistic programming, resource bound analysis, static analysis

1 Introduction

Probabilistic programming [64, 77] is an increasingly popular technique for implementing and analyzing Bayesian Networks and Markov Chains [41], randomized algorithms [9], cryptographic constructions [11], and machine-learning algorithms [40]. Compared with deterministic programs, reasoning about probabilistic programs adds additional complexity and challenges. As a result, there is a renewed interest in developing automatic and manual analysis and verification techniques that help programmers to reason about their probabilistic code. Examples of such developments are probabilistic program logics [20, 60, 64, 69, 76], automatic probabilistic invariant generation [22, 25], abstract interpretation for probabilistic programs [29, 70, 71], symbolic inference [38], and probabilistic model checking [61].

One important property that is often part of the formal and informal analysis of programs is *resource bound analysis*: What is the amount resources such as time, memory or energy that is required to execute a program? Over the past decade, the programming language community has developed numerous tools that can be used to (semi-)automatically derive non-trivial symbolic resource bounds for imperative [16, 45, 63, 80] and functional programs [6, 32, 52, 86].

Existing techniques for resource bound analysis can be applied to derive bounds on the *worst-case* resource consumption of probabilistic programs. However, if the control-flow is influenced by probabilistic choices then a worst-case analysis is often not applicable because there is no such upper bound. Consider the example *trader*(s_{min},*s*) in Figure 1(a) that implements a 1-dimensional random walk to model the fluctuations of a stock price *s*. With probability $\frac{1}{4}$ the price increases by 1 point and with probability $\frac{3}{4}$ the price decreases by 1 point. After every price change, a trader performs an action *trade(s)*—like buying 10 shares—that can depend on the current price until the stock price falls to s_{min}. In the worst case, *s* will be incremented in every loop iteration and the loop will not terminate. However, the loop terminates with probability 1, called *almost surely* (a.s.) termination [34].

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

In this article, we are introducing a new method for deriving bounds on the expected resource consumption of probabilistic programs. Our technique derives symbolic polynomial bounds, is fully automatic, and generates certificates that are derivations in a quantitative program logic [60, 76]. For example, given the function *trade*(s_{min},*s*), our technique

PL'17, January 01–03, 2017, New York, NY, USA
2017.

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

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

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

automatically derives the bound $2{\cdot}\max(0, s{-}s_{min})$ on the expected number of loop iterations. For the total spending of the trader, our technique automatically derives the bound $5{\cdot}\max(0, s{-}s_{min}) + 10{\cdot}\max(0, s{-}s_{min}){\cdot}\max(0, s_{min}) + 5{\cdot}\max(0, s{-}s_{min})^2$ on the expected value of the variable *cost* in less than 7 seconds. Both bounds are tight in the sense that they precisely describe the expected resource consumption.

To the best of our knowledge, we present the first fully automatic analysis for deriving symbolic bounds on the expected resource consumption of probabilistic programs with probabilistic branchings and sampling from discrete distributions. It is also one of the few techniques that can automatically derive polynomial properties. Different resource metrics can be defined either by using a resource-counter variable or by using *tick* commands. The analysis is compositional, automatically tracks size changes, and derives whole program bounds. Note that derived time bounds also imply *positive termination* [34], that is, termination with bounded expected runtime. Compositionality is particularly tricky for probabilistic programs since the composition of two positively terminating programs is not a positively terminating in general. While we focus on bounds on the expected cost, the analysis can also be used to derive worst-case bounds. Moreover, we can adapt the analysis (following [72]) to also derive lower bounds on the expected resource usage.

Resource bound analysis and static analysis of probabilistic programs have developed largely independently. The key insight of our work is that there are close connections between (manual) quantitative reasoning methods for probabilistic programs and automatic resource analyses for deterministic programs. Our novel analysis combines probabilistic quantitative reasoning using a weakest precondition (WP) calculus [20, 60, 69, 76] with an automatic resource analysis method that is known as automatic amortized resource analysis (AARA) [17, 19, 49, 52], while preserving the best properties of both worlds. On the one hand, we have the strength and conceptual simplicity of the WP calculus. On the other hand, we get template-based bound inference that can be efficiently reduced to off-the-shelf LP solving.

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

In summary, we make the following contributions.

- We describe the first automatic analysis that derives symbolic bounds on the expected resource usage of probabilistic programs.
- We prove the soundness of the method by showing that a successful bound analysis produces derivations in a probabilistic WP calculus [76].
- We show the effectiveness of the technique with a prototype implementation and by successfully analyzing 39 examples from a new benchmark set and from previous work on probabilistic programs and randomized algorithms.

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

2 Probabilistic programs

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

2.1 Essential notions and concepts

The interested readers can find more detailed descriptions in reference textbooks [4].

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

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

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

Figure 3. Abstract syntax of the probabilistic language.

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

2.2 Syntax of probabilistic programs

The probabilistic programming language we use is a simple imperative integer language structured into expressions and commands. The abstract syntax is given by the grammar in Figure 3. The command $\mathsf{id} = e \ \mathsf{bop}\ R$, where R is a (discrete) random variable whose probability distribution is μ_R (written as $R \sim \mu_R$), is a *random sampling* assignment. It first samples according to the distribution μ_R to obtain a sample value then evaluates the expression in which R is replaced by the sample value. Finally, the evaluated value is assigned to id. The command $c_1 \oplus_p c_2$ is a *probabilistic branching*. It executes the command c_1 with probability p, or the command c_2 with probability $(1 - p)$.

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

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

```
while (h<=t) {
  t=t+1;
  h=h+unif(0,10)
  ⊕1/2 skip;
  tick(1);
}
```

Figure 2. The *race* program.

To represent a complete program we use a pair $(c, \mathscr{D})$, where $c \in C$ is the body of the main procedure and $\mathscr{D} : \mathsf{PID} \to C$ is a map from procedure identifiers to their bodies. A command with no procedure calls is called *closed* command. The program *race* in Figure 32 shows a complete example of a probabilistic program that is adapted from [22]. It models a race between a hare (variable h) and a tortoise (variable t). The race ends when the hare is ahead of the tortoise (exit condition $h > t$). After each unit of time (expressed using the tick command, line 6), the tortoise goes one step forward (line 3). With probability $\frac{1}{2}$ (line 5) the hare remains at its position and with the same probability it advances a random number of steps between 0 and 10 (line 4).

3 Expected resource bound analysis

In this section, we show informally how automatic amortized resource analysis (AARA) [18, 49, 52] can be generalized to compute upper bounds on the expected resource consumption of probabilistic programs. We first illustrate with a classic analysis of a one-dimensional random walk how developers currently analyze the expected resource usage. We then recap AARA for imperative programs [18, 19]. Finally, we explain the new concept of the *expected potential method* that we develop in this work and apply it to our random walk and more challenging examples.

3.1 Manual analysis of a simple random walk

Consider the following implementation of a random walk.

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

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

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

This is a recurrence relation of degree 3, but it can be solved more easily by defining $D(n)$ to be $T(n) - T(n-1)$ and rewriting the equation as $3D(n) = 4 + D(n+1)$. One systematic way to find solutions for this equation is to use the generating function $G(z) = \sum_{n \geq 1} D(n) z^n$. Writing D for $D(1)$, we get

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

And thus, using algebra and generating functions, we have

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

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

There are several reasons why this classic method make it hard to automate. First, inferring the recurrence relations

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

3.2 The potential method

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

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

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

$$\frac{\{\Phi\}c_1\{\Phi'\} \qquad \{\Phi'\}c_2\{\Phi''\}}{\{\Phi\}c_1; c_2\{\Phi''\}}\ (\textsc{QSeq})$$

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

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

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

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

3.3 The expected potential method

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

Like in the deterministic case, we would like to work with triples of the form $\{\Phi\}c\{\Phi'\}$ for potential functions $\Phi, \Phi' : \Sigma \to \mathbb{Q}_{\geq 0}$ to ensure compositional reasoning. However, in the case of probabilistic programs we need to take into account the expected value of the potential function Φ' w.r.t the distribution of final states resulting from the program execution. More precisely, the two functions must satisfy the constraint

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

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

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

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

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

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

Consider the simple random walk again whose expected resource usage is $\Phi = 2|[0, x]|$. This is the potential that

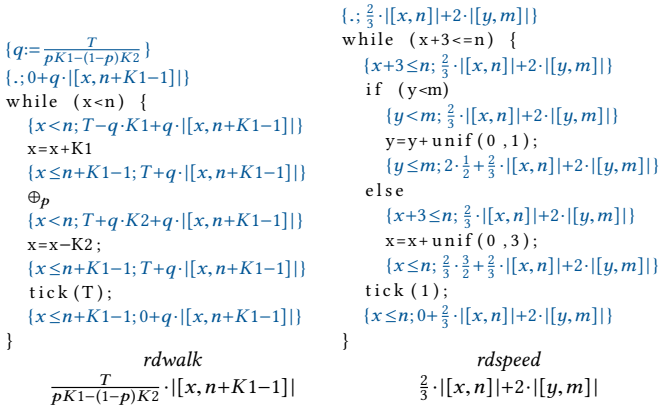


Figure 4. Derivations of bounds for single loop programs.

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

The beauty of the potential method is that this reasoning is in fact sound! But note that this is not a trivial result. For instance, replacing the probabilistic branching by its "average" action $x = x - \frac{1}{2}$ would be unsound in general.

General random walk. Now consider the generalized version *rdwalk* in Figure 4. It simulates a general *one-dimensional random walk* [43] with arbitrary positive constants K_1, K_2, T, and p. The expected number of loop iterations, and thus the expected cost modeled by the command tick (T), is bounded iff $(\star)$ $pK_1-(1-p)K_2>0$. In this case, the expected distance for "forward walking" is bigger than the expected distance for backward walking.

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

3.4 Bound derivations for challenging examples

We show how the expected potential method can derive polynomial bounds for challenging probabilistic programs with single, nested, and sequential loops, as well as procedure calls (more examples are given in Appendix G). All the presented bounds are derived automatically by our tool Absynth whose implementation and inference algorithm are discussed in Sections 5 and 7.

Single loops. Examples *rdwalk* and *rdspeed* in Figure 4 show that our expected potential method can handle *tricky iteration patterns*. Example *rdwalk* has already been discussed. Example *rdspeed* is randomized version of the one from papers on worst-case bound analysis [19, 45]. The iteration first increases y by 0 or 1 until it reaches m randomly according to the uniform distribution. Then x is increased by $k \in [0,3]$ which is sampled uniformly. Absynth derives the tight bound $\frac{2}{3}\cdot|[x,n]|+2\cdot|[y,m]|$. The derivation is similar to the derivation of the bound for *rdwalk*. To reason about the sampling construct, we consider the effect of all possible samples on the potential function. With the same reasoning for probabilistic branching, we then compute the weighted sum on the resulting initial potentials where the weights are assigned following the distribution. In some cases (like in this one), it is sound to just replace the distribution with the expected outcome. However, this does not work in general since the resource consumption could depend on the sample in a non-uniform way.

Nested and sequential loops. The examples in Figure 5 also show how the expected potential method can effectively handle *interacting nested* and *sequential loops*. Example *prnes* contains a nested loop, in which for each iteration of the outer loop, the variable n is increased by 1 or remains unchanged with respective probabilities $\frac{9}{10}$ and $\frac{1}{10}$. In each iteration, the variable y is incremented by 1000. We use the condition y>=100 && $\star$ to express that we can non-deterministically exit the inner loop. Here, the variable y is randomly decremented by 100 or 90 with probability $\frac{1}{2}$. The analysis discovers that only the expected runtime $\frac{5}{95}\cdot|[0,y]| = \frac{1}{19}\cdot|[0,y]|$ of the first execution of the inner loop depends on the initial value of y. For the other executions of the loop, the expected number of ticks is $\frac{1}{19}\cdot 1000$. The derivation infers a tight bound in which $\frac{10}{9}\cdot|[n,0]|$ is the expected number of iterations of the outer loop. The example *prseq* shows the capability of the expected potential method to take into account the interactions between sequential loops by deriving the expected value of the size changes of the variables. In the first loop, the variable y is incremented by sampling from a binomial distribution with parameters $n = 3$

Figure 5. Derivations of bounds on the expected value of ticks for probabilistic programs. Example *prseq* contains a sequential loop so that the iterations of the second loop depend on the the first one and *prnes* contains an interacting nested loop. In the derivation of *trader*, we derive the non-linear bound $\mathrm{inv}(s, s_{\min})$ on the global variable *cost* from the stock price example.

and $p = \frac{2}{3}$. In the second loop, y is decreased by 10 with probability $\frac{2}{3}$ or left unchanged otherwise. It accurately derives the expected value of the size change of y by transferring the potential $|[y,z]|$ to $|[0,y]|$.

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

3.5 Limitations

Deriving symbolic resource bounds is an undecidable problem, thus our technique does not work if loops and recursive functions have complex control flow. Furthermore, our implementation currently only derives polynomial bounds and supports sampling from discrete distributions with a finite domain.

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

4 Derivation system for probabilistic quantitative analysis

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

4.1 Potential functions

The main idea to automate resource analysis using the potential method is to fix the shape of the potential functions. More formally, potential functions are taken to be linear combinations of more elementary *base functions*. Finding the suitable base functions for a given program is discussed in Section 7. For now we assume a given list of N base functions. For convenience, they are represented as a vector $B = (b_1, \ldots, b_N)$, where each $b_i : \Sigma \to \mathbb{Q}$ maps program states to rational numbers. Building on base functions, a potential function is defined by N coefficients $q_1, \ldots, q_N \in \mathbb{Q}$ as $\Phi(\sigma) = \sum_{i=1}^{N} q_i \cdot b_i(\sigma)$. For presentation purposes the coefficients are stored as a vector $Q = (q_1, \ldots, q_N)$, called *potential annotation*. Each potential annotation corresponds to a potential function Φ_Q that we can concisely express as the dot product $\langle Q \cdot B \rangle$.

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

4.2 Judgements

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

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

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

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

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

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

(Q:Skip)
$$\frac{}{\vdash \{\Gamma; Q\}\ \mathsf{skip}\ \{\Gamma; Q\}}$$

(Q:Assert)
$$\frac{}{\vdash \{\Gamma; Q\}\ \mathsf{assert}\ e\ \{\Gamma \wedge e; Q\}}$$

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

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

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

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

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

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

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

(Q:Call)
$$\frac{P : (\Gamma; Q, \Gamma'; Q') \in \Delta \qquad x \in \mathbb{Q}_{\geq 0}}{\vdash \{\Gamma; Q + x\}\ \mathsf{call}\ P\ \{\Gamma'; Q' + x\}}$$

(Q:Tick)
$$\frac{}{\vdash \{\Gamma; Q\}\ \mathsf{tick}\ (q)\ \{\Gamma; Q - q\}}$$

(Relax)
$$\frac{F = (F_1, \ldots, F_N) \qquad \vec{u} = (u_1, \ldots, u_N)^\intercal \quad \forall i.\Gamma \models \Phi_{F_i} \geq 0 \qquad \forall i.u_i \geq 0 \qquad Q' = Q - F\vec{u}}{Q \geq_\Gamma Q'}$$

(Q:Abort)
$$\frac{}{\vdash \{\Gamma; 0\}\ \mathsf{abort}\ \{\Gamma'; Q'\}}$$

(Q:Weaken)
$$\frac{\Gamma \models \Gamma_0 \qquad Q \geq_\Gamma Q_0 \qquad \vdash \{\Gamma_0; Q_0\}c\{\Gamma'_0; Q'_0\} \qquad \Gamma'_0 \models \Gamma' \qquad Q'_0 \geq_{\Gamma'_0} Q'}{\vdash \{\Gamma; Q\}c\{\Gamma'; Q'\}}$$

Figure 6. Inference rules of the derivation system.

4.3 Inference rules

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

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

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

The second probabilistic rule Q:Sample deals with sampling assignments. Recall that R is a random variable following a distribution μ_R. The essence of the rule is that, since we assumed that R is bounded, we can treat a sampling assignment as a (nested) probabilistic branching. Each of the branches contains an assignment $x = x$ bop v with $v \in \mathbb{Z}$ and is executed with probability p, the probability of the event $R = v$. The preconditions of each of the branches are combined like in the Q:PIf rule.

The rules Q:Assign and Q:Weaken are similar to the ones of a previous implementation of AARA for the analysis of non-probabilistic programs [18] but have been adapted to our structured probabilistic language. In the rule Q:Assign for $x = e$, we represent the state transformation as a linear transformation on potential functions. If $\Phi_{Q'}$ is a bound on the expected cost of c', then $\Phi_{Q'}[e/x]$ is the expected resource bound of $x = e; c'$ To encode this constraint as a linear program (this is necessary to enable automation using LP solving), we find all the *stable* base functions, denoted $\mathscr{S}_{x=e}$, that is, all the base functions b_j for which there exists $(a_{i,j})_i \in \mathbb{Q}$ such that $b_j[e/x] = \sum_i a_{i,j} \cdot b_i$. That means a function is stable if its extending with the mapping of x to e can be represented by the set of based functions. Finally, to ensure that the transformation w.r.t the assignment $x = e$ on the potential function $\Phi_{Q'}$ is linear, we require that all the base functions that are not stable have their coefficients set to 0 in Q'. With these constraints, we have that $\Phi_{Q'}[e/x] = \Phi_{AQ'} = \Phi_Q$ where A is the $(N \times N)$ matrix with coefficients $(a_{i,j})$, hence justifying the validity.

The essence of the Q:Weaken is that it is always safe to add potential in the precondition and remove potential in the postcondition. This concept of more (or less) potential is made precise by the predicate $Q \geq_\Gamma Q'$. Semantically, $Q \geq_\Gamma Q'$ encodes—using linear constraints—the fact that in all states $\sigma \models \Gamma$, we have $\Phi_{Q'}(\sigma) \geq \Phi_Q(\sigma)$ (see Lemma D.1 in Appendix D). The Relax rule uses *rewrite functions* $(F_i)_i$, as introduced in [18]. Rewrite functions are linear combinations of base functions that can be proved non-negative in the logical context Γ. Using rewrite functions, the idea of the judgement $Q \geq_\Gamma Q'$ is that, to obtain Q', one has to subtract a non-negative quantity from Q.

The rule Q:Call handles procedure calls. The pre- and postcondition for the procedure P are fetched from the specification context Δ. Then, a non-negative *frame* $x \in \mathbb{Q}_{\geq 0}$ is added to the procedure specification. This frame allows to pass some constant potential through the procedure call and is required for the analysis of most non-tail-recursive algorithms. The idea of this framing is that if the triple $\{.; Q\}c\{.; Q'\}$ is valid, we can also take $Q'+x$ as postcondition and the need for the extra cost x required by the continuation command can be threaded up to the precondition that becomes $Q+x$. In the soundness proof, this framing boils down to the "propagation of constants" property on the calculus [76] used in our formal soundness proof.

5 Automatic constraint generation and solving using LP solvers

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

Generating linear constraints. A detailed example of this process is shown in Figure 7. Note that Q:Weaken is applied twice. Since this rule is not syntax-directed, it can be applied at any point during the derivation. In our implementation, we apply it around all assignments. This proved sufficient in practice and limits the number of constraints generated. In the figure, the potential annotations are represented by an upper-case letter P or Q with an optional superscript. For example, Q represents the potential function

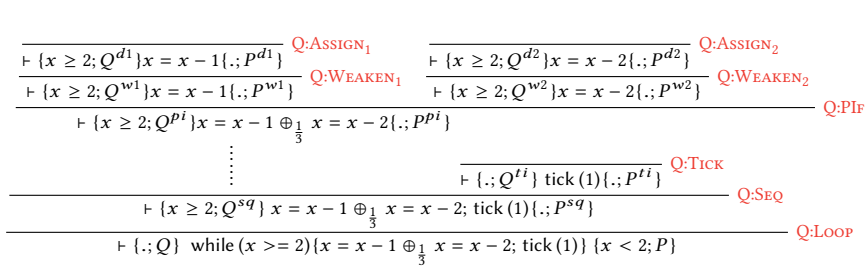
$$q_1 \cdot \mathbf{1} + q_{x0} \cdot |[0,x]| + q_{x1} \cdot |[1,x]| + q_{x2} \cdot |[2,x]|$$

The set of base functions is $\mathbf{1}$ and $|[i,x]|$ for $i \in \{0,1,2\}$. We will see that they are sufficient to infer a bound. Details of how to select base functions are given in Section 7. To apply weakening, we need rewrite functions, we pick

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

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

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



Constraints	Rules
$Q = Q^{sq} = P^{sq} = P$	Q:Loop
$Q^{sq} = Q^{pi} \wedge P^{pi} = Q^{ti} \wedge P^{ti} = P^{sq}$	Q:Seq
$Q^{pi} = \frac{1}{3}\cdot Q^{w1} + \frac{2}{3}\cdot Q^{w2} \wedge P^{pi} = P^{w1} = P^{w2}$	Q:PIf
$Q^{w1} \geq_{(x\geq 2)} Q^{d1} \wedge P^{d1} \geq_{(x\geq 2)} P^{w1}$	Q:Weaken1
$q_1^{d1} = p_1^{d1} \wedge q_{x0}^{d1} = 0 \wedge q_{x1}^{d1} = p_{x0}^{d1} \wedge$ $q_{x2}^{d1} = p_{x1}^{d1} \wedge p_{x2}^{d1} = 0$	Q:Assign1
$Q^{w2} \geq_{(x\geq 2)} Q^{d2}; P^{d2} \geq_{(x\geq 2)} P^{w2}$	Q:Weaken2
$q_1^{d2} = p_1^{d2} \wedge q_{x0}^{d2} = 0 \wedge q_{x1}^{d2} = 0 \wedge$ $q_{x2}^{d2} = p_{x0}^{d2} \wedge p_{x1}^{d2} = 0 \wedge p_{x2}^{d1} = 0$	Q:Assign2
$Q^{ti} = P^{ti} + 1$	Q:Tick

Figure 7. Inference of a derivation using linear constraint solving.

The constraints given in the table in Figure 7 use shorthand notations to constrain all the coefficients of two annotations. For instance $Q = Q^{sq}$ should be expanded into $q_1 = q_1^{sq} \wedge q_{x0} = q_{x0}^{sq} \wedge q_{x1} = q_{x1}^{sq} \wedge q_{x2} = q_{x2}^{sq}$. The most interesting rules are the probabilistic branching, the two weakenings, and the two assignments. For the probabilistic branching, following Q:PIf, the preconditions of the two branches are linearly combined using the weights $\frac{1}{3}$ and $1 - \frac{1}{3} = \frac{2}{3}$.

We now discuss the first weakening. The second one generates an identical set of constraints—but the LP solver will give it a different solution. The most interesting constraints are the ones for $Q^{w1} \geq_{(x\geq 2)} Q^{d1}$. This relation is defined by the rule Relax in Figure 6 and involves finding all the applicable rewrite functions in the logical state $x \geq 2$. As discussed above, F_0, F_1, and F_2 are all applicable, and the following system of constraints, written in matrix notation, is generated.

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

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

For the first assignment Q:Assign$_1$, the stable set discussed in Section 4.3 is $\mathscr{S}_{x=x-1} = \{1, |[0,x]|, |[1,x]|\}$. Indeed, only $|[2,x]|$ is unstable since it becomes $|[3,x]|$ after the assignment $x = x-1$. Since the assignment leaves **1** unchanged and changes $|[0,x]|$ into $|[1,x]|$ and $|[1,x]|$ into $|[2,x]|$, the system of constraints generated is

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

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

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

6 Soundness of the analysis

The soundness of the analysis is proved with respect to an operational semantics based on Markov decision processes (see Appendix A). It leverages previous work on probabilistic programs by relying on the soundness of a weakest pre-expectation (WP) calculus [60, 76]. The weakest pre-expectation $\mathrm{ert}\,[c, \mathscr{D}](f)(\sigma)$ is the (exact) expected amount of resources consumed by a program $(c, \mathscr{D})$ started in state σ if it is followed by a computation that has an expected resource consumption given by a function $f : \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. See Appendix B for more details and a formal definition.

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

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

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

Proof. The proof is done by induction on the program structure and the derivation using the inference rules. See Appendix D for details. □

7 Implementation and experiments

In this section, we first describe the implementation of the automatic analysis in the tool Absynth. Then, we evaluate the performance of our tool on a set of challenging examples.[2]

7.1 Implementation

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

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

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

Generating base and rewrite functions. Our analysis can work with every set of base functions. While it would be possible to to fix a set of functions once and for all as in previous work on resource analysis [19, 47], we found that it is more effective to select the base functions for each program using a heuristic [18]. The abstract interpretation (AI) used in Absynth to infer logical contexts derives linear inequalities between program variables and uses a Presburger decision procedure. Our implementation uses these inequalities to heuristically generate a set of base and rewrite functions. For example, if $n > x$ at one program point, the heuristic will add the monomial $\max(0, n - x)$ as a base function. Higher-degree base functions can be constructed by considering successive powers and products of simpler base functions. One can use a more complex and powerful AI such as the Apron library [57]. In practice, we found that our simple AI is sufficient to infer many bounds and provides good performance.

When the heuristic adds a new base function, a set of rewrite functions is enriched to allow transfers of potential to and from the new base function. For instance, for the base function $\max(0, n - x)$ we add the rewrite function $F = \max(0, n - x) - \max(0, n - x - 1) - 1$. F can be used for an assignment $x = x + 1$ when $n - x > 0$. So if $\max(0, n - x)$ is the potential before the assignment, F can be used to turn it into $\max(0, n - (x + 1)) + 1$ which, after the assignment, becomes $\max(0, n - x) + 1$, effectively extracting one unit of constant potential.

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

7.2 Experimental evaluation

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

The programs *bayesian* [41], *filling* [78], *race*, *2drwalk*, *robot*, *roulette* [22], and *sampling* [60] have been described in the literature on probabilistic programs, in which their expected resource consumption has been analyzed manually. The programs *C4B_**, *prseq*, *prseq*, *preseq_bin*, *prspeed*, *rdseql*, *rdspeed*, and *recursive* are probabilistic versions of deterministic examples from previous work [18, 19, 45]. The other examples are either adaptations of classic randomized algorithms or handcrafted new programs that demonstrate particular capabilities of our analysis. Section 3 contains some representative listings.

To measure the expected resource usage of all examples by simulation, we uniformly chose the range of inputs to be 1000 to 5000 and allowed only 1 input variable to vary while choosing fixed random values for other inputs.[3] We sampled the resource usage 10000 times for each input using the GSL-GNU scientific library [1]. We then compared the results to our statically computed bounds. The simulation is implemented in C^{++} and consists of more than 5000 LOC.

The experiments were run on a machine with an Intel Core i5 2.4 GHz processor and 8GB of RAM under macOS 10.13.1. The LP solver we use is CoinOr CLP [81].

Results. The results of the evaluation are compiled in Table 1. The table is split into linear and non-linear bounds. It contains the inferred bounds, the total time taken by Absynth, and the means in percentage of the absolute errors between the measured expected values and the inferred bounds. In general, the analysis finds bounds quickly: All the examples are processed in less than 10 seconds. The analysis time mainly depends on three factors: the number of variables in the program, the number of base functions, and the size of the distribution's domain in the sampling commands. The user can specify a maximal degree of the bounds to control the number of base functions under consideration. Our inference rule for the sampling commands is very precise but the

[2] The source code of the examples, Absynth, the experiments, and the simulation-based comparison have been submitted as auxiliary material.

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

Program	Expected bound	Error(%)	Time(s)
Linear programs			
2drwalk	$2\cdot\lvert[d, n+1]\rvert$	0.170	2.278
bayesian	$5\cdot\lvert[0, n]\rvert$	0	0.272
ber	$2\cdot\lvert[x, n]\rvert$	0.026	0.008
bin	$0.2\cdot\lvert[0, n+9]\rvert$	0.290	0.281
C4B_t09	$8.27273\cdot\lvert[0, x]\rvert$	5.362	0.061
C4B_t13	$1.25\cdot\lvert[0, x]\rvert + \lvert[0, y]\rvert$	0.009	0.045
C4B_t15	$2\cdot\lvert[0, x]\rvert$	A.S	0.044
C4B_t19	$\lvert[0, k+i+51]\rvert + 2\cdot\lvert[100, i]\rvert$	2.711	0.058
C4B_t30	$0.5\cdot\lvert[0, x+2]\rvert + 0.5\cdot\lvert[0, y+2]\rvert$	W.C	0.032
C4B_t61	$0.060606\cdot\lvert[0, l-1]\rvert + \lvert[0, l]\rvert$	0.754	0.028
condand	$\lvert[0, m]\rvert + \lvert[0, n]\rvert$	A.S	0.010
cooling	$0.42\cdot\lvert[0, t+5]\rvert + \lvert[\text{st}, \text{mt}]\rvert$	0.192	0.079
fcall	$2\cdot\lvert[x, n]\rvert$	0.025	0.008
filling	$0.037037\cdot\lvert[0, \text{vol}+2]\rvert+$ $0.333333\cdot\lvert[0, \text{vol}+10]\rvert+$ $0.296296\cdot\lvert[0, \text{vol}+11]\rvert$	0.713	0.615
hyper	$5\cdot\lvert[x, n]\rvert$	0.061	0.013
linear01	$0.6\cdot\lvert[0, x]\rvert$	0.036	0.016
miner	$7.5\cdot\lvert[0, n]\rvert$	0.071	0.077
prdwalk	$1.14286\cdot\lvert[x, n+4]\rvert$	0.128	0.052
prnes	$68.4795\cdot\lvert[0, -n]\rvert + 0.052631\cdot\lvert[0, y]\rvert$	0.122	0.057
prseq	$1.65\cdot\lvert[y, x]\rvert + 0.15\cdot\lvert[0, y]\rvert$	0.144	0.057
prseq_bin	$1.65\cdot\lvert[y, x]\rvert + 0.15\cdot\lvert[0, y]\rvert$	0.150	0.082
prspeed	$2\cdot\lvert[y, m]\rvert + 0.666667\cdot\lvert[x, n]\rvert$	0.039	0.057
race	$0.666667\cdot\lvert[h, t+9]\rvert$	0.294	0.245
rdseql	$2.25\cdot\lvert[0, x]\rvert + \lvert[0, y]\rvert$	0.007	0.025
rdspeed	$2\cdot\lvert[y, m]\rvert + 0.666667\cdot\lvert[x, n]\rvert$	0.039	0.040
rdwalk	$2\cdot\lvert[x, n+1]\rvert$	0.075	0.012
robot	$0.384615\cdot\lvert[0, n+6]\rvert$	R.D	2.658
roulette	$4.93333\cdot\lvert[n, 10010]\rvert$	0.282	1.216
sampling	$2\cdot\lvert[0, n]\rvert$	0.026	3.347
sprdwalk	$2\cdot\lvert[x, n]\rvert$	0.032	0.017
Polynomial programs			
complex	$6\cdot\lvert[0, m]\rvert\cdot\lvert[0, n]\rvert + 3\cdot\lvert[0, n]\rvert+\lvert[0, y]\rvert$	0.118	3.415
multirace	$2\cdot\lvert[0, m]\rvert\cdot\lvert[0, n]\rvert + 4\cdot\lvert[0, n]\rvert$	0.703	9.034
pol04	$4.5\cdot\lvert[0, x]\rvert^2 + 7.5\cdot\lvert[0, x]\rvert$	0.779	0.585
pol05	$\lvert[0, x]\rvert^2 + \lvert[0, x]\rvert$	0.431	0.353
pol06	$0.625\cdot\lvert[\min, s]\rvert+$ $2\cdot\lvert[\min, s]\rvert\cdot\lvert[0, \min]\rvert + 0.625\cdot\lvert[\min, s]\rvert^2$	A.S	7.066
pol07	$1.5\cdot\lvert[0, n-2]\rvert\cdot\lvert[0, n-1]\rvert$	0.008	4.534
rdbub	$3\cdot\lvert[0, n]\rvert^2$	0.106	0.190
recursive	$0.25\cdot\lvert[l, h]\rvert^2+1.75\cdot\lvert[l, h]\rvert$	0.281	3.791
trader	$5\cdot\lvert[s_{\min}, s]\rvert^2 + 5\cdot\lvert[s_{\min}, s]\rvert+$ $10\cdot\lvert[s_{\min}, s]\rvert\cdot\lvert[0, s_{\min}]\rvert$	0.251	7.262

Table 1. Automatically-derived bounds on the expected number of ticks with Absynth.

price we pay for the precision is a linear constraint set whose size is proportional to the range of the sampling distribution.

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

However, there is no guarantee that Absynth infers asymptotically tight bounds and there are many classes of bounds that Absynth cannot derive. For example, for the programs whose errors are denoted by *A.S* in Table 1 we did not compute asymptotically tight bounds. *C4B_t15* has logarithmic expected cost, thus the best bound that Absynth can derive is a linear bound. Similarly, $|[0,n]|+|[0,m]|$ is the best bound that can be inferred for *condand* whose expected cost is $2\cdot\min\{|[0,n]|, |[0,m]|\}$. Another source of imprecise constant factors in the bounds is rounding. The program *robot* has an imprecise constant factor, denoted *R.D* in the table, because it contains a deep nesting of probabilistic choices.

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

8 Related work

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

Resource bound analysis. Most closely related to our work is prior work on AARA for deterministic programs. AARA has been introduced in [52] for automatically deriving linear worst-case bounds for first-order functional programs. The technique has been generalized to derive polynomial bounds [48, 50, 51, 54, 55], lower bounds [72], and to handle (strictly evaluated) programs with arrays and references [67], higher-order functions [49, 58], lazy functional programs [79, 83], object-oriented programs [53, 56], and user defined data types [49, 59]. It also has been integrated into separation logic [5] and proof assistants [24, 74]. A distinctive common theme of sharing is compositionality and automatic bound inference via LP solving.

In contrast to our work, all prior research on AARA targets deterministic programs and derives worst-case bounds rather than bounds on the expected resource usage. In our formulation of AARA for probabilistic programs, we build on prior work that integrated AARA into Hoare logic to derive bounds for imperative code [17–19, 73], a new technique for deriving polynomial bounds on the expected resource usage of programs with probabilistic sampling and branching.

Beyond AARA there exists many other approaches to automatic worst-case resource bound analysis for deterministic programs. They are based on sized types [82], linear

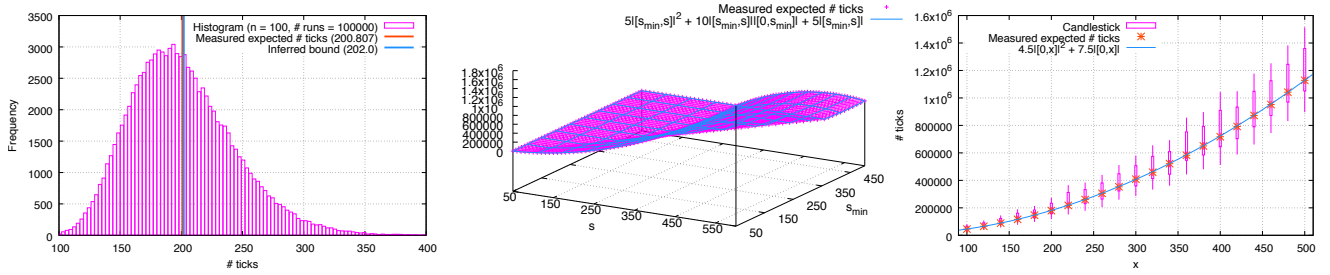


Figure 8. Comparison of automatically derived bounds with measured cost samples. On the left: histogram of the distribution of #ticks for *rdwalk* with $n = 100$. On the right: The inferred bound on the expected #ticks (blue lines) compared to the measured expected values for various input sizes (red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

dependent types [65, 66], refinement types [28, 86], annotated type systems [30, 31], defunctionalization [6], recurrence relations [2, 3, 12, 32, 36, 44, 63], abstract interpretation [13, 21, 45, 80, 85], template based assume-guarantee reasoning [68], measure functions [26], and techniques from term rewriting [7, 16, 37, 75]. These techniques do not apply to probabilistic programs and do not derive bounds on expected resource usage.

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

We are only aware of few works that study the analysis of expected resource usage of probabilistic programs. Chatterjee et al. [27] propose a technique for solving recurrence relations that arise in the analysis of expected runtime cost. Their technique can derive bounds of the form $O(\log n)$, $O(n)$, and $O(n \log n)$. Similarly, Flajolet et al. [35] describe an automatic for average-case analysis that is based on generating functions and that can be seen as a method for solving recurrences. While these techniques apply to recurrences that describe the resource usage of randomized algorithms, the works do not propose a technique for deriving recurrences from a program. It is therefore not a push-button analysis for probabilistic programs but complementary to our work since they can derive logarithmic bounds.

Analysis of probabilistic programs. Considering work on analyzing probabilistic programs, most closely related is a recent line of work by Kaminski et al. [60, 76]. The goal of this work is to characterize the expected runtime of probabilistic programs. However, they use a WP calculus to derive pre-expectations and do not consider any automation. The technique can be seen as a generalization of quantitative Hoare logic [17, 19] for AARA to the probabilistic setting but does not provide support for automatic reasoning. In fact, when attempting to generalize AARA to probabilistic programs we were first unaware of the existing work and rediscovered some of the proof rules. Our contributions are new specialized proof rules that allow for automation using LP solving and a prototype implementation of the new technique. While our soundness proof is original, it leverages the proof by Kaminski et al. by relying on the soundness of the rules for weakest preconditions.

The use of pre-expectations for reasoning about probabilistic programs dates back to the pioneering work of Kozen and others [20, 64, 69]. It has been automated using constraint generation [62] and abstract interpretation [23] to derive quantitative invariants. However, it is unclear how to use them to automatically derive symbolic (polynomial) bounds like in our work.

Another body of research relies probabilistic pushdown automata and martingale theory to analyze the termination time [15] and the expected number of steps [33]. The use of martingale theory to automatically analyze probabilistic programs has been pioneered in [22]. While their technique also relies on linear constraints, it is proving almost-sure termination instead of resource bounds and relies on Farka's lemma. More general methods [25] are able to synthesize polynomial ranking-supermartingales for proving termination.

Abstract interpretation has also been applied to probabilistic programs [29, 70, 71] but we are not aware of its application to derive bounds on the expected resource usage. Another approach to automatically analyze probabilistic programs is based on symbolic inference [38] and analyzing execution paths with statistical techniques [14, 39, 78]. In the context of analyzing differential privacy, there are works with limited automation that focus on deriving bounds on the privacy budget for probabilistic programs [10, 46].

9 Conclusion

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

In the future, we plan to study how to build on the introduced technique to automatically derive tail bounds, that is, worst-case bounds that hold with high probability. We are also working on a more direct soundness argument that also works for non-monotone resources. Finally, we plan to build on Resource Aware ML [49] to apply the expected potential method to (higher-order) functional programs.

References

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

[2] E. Albert, J. C. Fernández, and G. Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, 2015.

[3] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symp. (SAS'12)*, 2012.

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

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

[6] M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[7] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[9] G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. Formal Certification of Randomized Algorithms. Technical report, 2016. http://justinh.su/files/papers/ellora.pdf.

[10] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P.-Y. Strub. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 411–424. IEEE Computer Society, 2014.

[11] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal Certification of Code-based Cryptographic Proofs. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages (POPL'09)*, pages 90–101, New York, NY, USA, 2009. ACM.

[12] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[13] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, 2010.

[14] M. Borges, A. Filieri, M. d'Amorim, C. S. Pasareanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *Conference on Programming Language Design and Implementation (PLDI'14)*, pages 123–132, 2014.

[15] T. Brázdil, S. Kiefer, A. Kucera, and I. H. Vareková. Runtime analysis of probabilistic programs with unbounded recursion. *J. Comput. Syst. Sci.*, 81(1):288–310, 2015.

[16] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

[17] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *35th Conference on Programming Language Design and Implementation (PLDI'14)*, 2014. Artifact submitted and approved.

[18] Q. Carbonneaux, J. Hoffmann, T. Reps, and Z. Shao. Automated Resource Analysis with Coq Proof Objects. In *29th International Conference on Computer-Aided Verification (CAV'17)*, 2017.

[19] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*, 2015. Artifact submitted and approved.

[20] O. Celiku and A. McIver. Compositional Specification and Analysis of Cost-Based Properties in Probabilistic Programs. In *Formal Methods, International Symposium of Formal Methods Europe (FM'05)*, pages 107–122, 2005.

[21] P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[22] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis using martingales. In *Computer-Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer-Verlag, 2013.

[23] A. Chakarov and S. Sankaranarayanan. Expectation invariants as fixed points of probabilistic programs. In *Static Analysis Symposium (SAS'14)*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer-Verlag, 2014.

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

[25] K. Chatterjee, H. Fu, and A. K. Goharshady. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference (CAV'16)*, pages 3–22, 2016.

[26] K. Chatterjee, H. Fu, and A. K. Goharshady. Non-polynomial Worst-Case Analysis of Recursive Programs. In *Computer Aided Verification - 29th Int. Conf. (CAV'17)*, 2017.

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

[28] E. Çiçek, D. Garg, and U. A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[29] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *Programming Languages and Systems - 21st European Symposium on Programming (ESOP'12)*, pages 169–193, 2012.

[30] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

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

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

[33] J. Esparza, A. Kucera, and R. Mayr. Quantitative analysis of probabilistic pushdown automata: Expectations and variances. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 117–126, 2005.

[34] L. M. Ferrer Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 489–501, New York, NY, USA, 2015. ACM.

[35] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-case Analysis of Algorithms. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.

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

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

[38] T. Gehr, S. Misailovic, and M. T. Vechev. 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*, pages 62–83, 2016.

[39] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 166–176, 2012.

[40] Z. Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521:452–459, 2015.

[41] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering (FOSE'14)*, pages 167–181, 2014.

[42] F. Gretz, J. Katoen, and A. McIver. Operational versus Weakest Pre–Expectation Semantics for the Probabilistic Guarded Command Language. *Performance Evaluation*, 73:110–132, 2014.

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

[44] B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.

[45] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, 2009.

[46] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential Privacy Under Fire. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 33–33. USENIX Association, 2011.

[47] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[48] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[49] J. Hoffmann, A. Das, and S.-C. Weng. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[50] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*, 2010.

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

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

[53] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, 2006.

[54] M. Hofmann and G. Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, 2014.

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

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

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

[58] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[59] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, 2009.

[60] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP'16)*. Springer, 2016.

[61] J. Katoen. The probabilistic model checking landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 31–45, 2016.

[62] J. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *Static Analysis - 17th International Symposium (SAS'10)*, pages 390–406, 2010.

[63] Z. Kincaid, J. Breck, A. F. Boroujeni, and T. Reps. Compositional recurrence analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI'17)*, 2017.

[64] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[65] U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[66] U. D. Lago and B. Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, 2013.

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

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

[69] A. McIver and C. Morgan. *Abstraction, Refinement and Proof For Probabilistic Systems (Monographs in Computer Science)*. Springer Verlag, 2004.

[70] D. Monniaux. Backwards abstract interpretation of probabilistic programs. In *Programming Languages and Systems, 10th European Symposium on Programming (ESOP'01)*, pages 367–382, 2001.

[71] D. Monniaux. Abstract interpretation of programs as markov decision processes. *Sci. Comput. Program.*, 58(1-2):179–205, 2005.

[72] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[73] H. R. Nielson. A hoare-like proof system for analysing the computation time of programs. *Sci. Comput. Program.*, 9(2):107–136, 1987.

[74] T. Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[75] L. Noschinski, F. Emmes, and J. Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning*, 51(1):27–56, 2013.

[76] F. Olmedo, B. L. Kaminski, J. Katoen, and C. Matheja. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 672–681, 2016.

[77] A. Pfeffer. *Practical Probabilistic Programming*. Manning, 2016.

[78] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *ACM conference on Programming Languages Design and Implementation (PLDI'13)*, pages 447–458. ACM Press, 2013.

[79] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, 2012.

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

[81] The CLP Team. CLP. https://projects.coin-or.org/Clp, 2018.

[82] P. Vasconcelos. *Space Cost Analysis Using Sized Types.* PhD thesis, School of Computer Science, University of St Andrews, 2008.

[83] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[84] W. Wechler. Universal Algebra for Computer Scientists. *EATCS Monographs on Theoretical Computer Science*, 25, 1992.

[85] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, 2011.

[86] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

A Operational cost semantics

Following existing work [76], we provide an operational semantics for probabilistic programs using pushdown MDPs extended with a reward function. Interested readers can find more details about MDPs in the literature [8, 33].

A program state $\sigma : \mathsf{VID} \to \mathbb{Z}$ is a map from variable identifiers to integer values. We write $[\![e]\!]_\sigma$ to denote the value of the expression e in the program state σ. We write $\sigma[v/x]$ for the program state σ extended with the mapping of x to v. For a probability distribution μ, we use $[\![\mu : v]\!]$ to indicate the probability that μ assigns to value v. We use Σ to denote the set of program states.

Given a program $(c, \mathscr{D})$, let L be the finite set of all program locations. Let $\ell_0 \in L$ be the initial location of c, $\downarrow$ be the special symbol indicating a termination of a procedure, and let Term be a special symbol for the termination of the whole program. For simplicity, we assume the existence of auxiliary functions that can be defined inductively on commands. The function $\mathsf{init} : C \to L$ maps procedure bodies to the initial program locations, the functions $\mathsf{cmd} : L \to C$ maps locations to their corresponding commands, and $\mathsf{fcmd} : L \to L \cup \{\downarrow\}$ and $\mathsf{scmd} : L \to L \cup \{\downarrow\}$ map locations to their first and second successors, respectively. If a location ℓ has no such successor, then $\mathsf{fcmd}(\ell) = \downarrow$ and $\mathsf{scmd}(\ell) = \downarrow$.

Probabilistic transitions. A program configuration is of the form (ℓ, σ), where $\sigma \in \Sigma$ is the current program state and $\ell \in L \cup \{\downarrow, \mathsf{Term}\}$ is a program location indicating the current command or a special symbol. We view the configurations as states of a pushdown MDP (the general definition follows) with transitions of the form

$$(\ell, \sigma) \xrightarrow{\alpha, \gamma, \overline{\gamma}, p} (\ell', \sigma')$$

Pushdown MDPs operate on stack of locations that act as return addresses. In a such a transition, (ℓ, σ) is the current configuration, $\alpha \in \mathsf{Act}$ is an action, γ is the program location on top of the return stack (or ϵ to indicate an empty stack), $\overline{\gamma}$ is a finite sequence of program locations to be pushed on the return stack, p is the probability of the transition, and (ℓ', σ') is the configuration after the transition. Like in a standard MPD, at each program configuration, the sum of probabilities of the outgoing edges labeled with a fixed action is either 0 or 1. The transition rules of our pushdown MDP is given in Figure 9. In the rules, we identify a singleton symbol γ with a one element sequence.

The set of actions is given by $\mathsf{Act} := \{\mathsf{Th}, \tau, \mathsf{El}\}$ where Th is the action for the *then* branch, El is the action for the *else* branch of a non-deterministic choice command, and τ is the standard action for other transitions. In the rule S:Call for procedure calls, the location of the command after the call command is pushed on the location stack and the control is moved to the first location of the body of the callee procedure. When a procedure terminates it reaches a state of the form $(\downarrow, \sigma)$. If the location stack is non-empty then the rule S:Return is applicable and a return location ℓ' is popped from the stack. If the stack is empty then the rule S:Term is applicable and the program terminates.

Resource consumption. To complete our pushdown MDP semantics, we have to define the resource consumption of a computation. We do so by introducing a *reward function* that assigns a reward to each configuration. The resource consumption of an execution is then the sum of the rewards of the visited configurations.

For simplicity, we assume that the cost of the program is defined exclusively by the $\mathsf{tick}(q)$ command, which consumes $q \geq 0$ resource units. Thus we assign the reward q to configurations with locations ℓ that contain the command $\mathsf{tick}(q)$ and a reward of 0 to other configurations.

To facilitate composition, it is handy to define the reward function with respect to a function $f : \Sigma \to \mathbb{R} \cup \{\infty\}$ that defines the reward of a continuation after the program terminates. So we define the reward of the configuration (Term, σ) to be $f(\sigma)$.

Pushdown MDPs. In summary, the semantics of a probabilistic program $(c, \mathscr{D})$ with the initial state σ_0 is defined by the pushdown MDP $\mathfrak{M}^f_{\sigma_0}[\![c, \mathscr{D}]\!] = (S, s_0, \mathsf{Act}, P, \mathscr{E}, \epsilon, \mathfrak{Re})$, where

- $S := \{(\ell, \sigma) \mid \ell \in L \cup \{\downarrow, \mathsf{Term}\}, \sigma \in \Sigma\}$,
- $s_0 := (\ell_0, \sigma_0)$,
- $\mathsf{Act} := \{\mathsf{Th}, \tau, \mathsf{El}\}$,
- the transition probability relation P is defined by the rules in Figure 9,
- $\mathscr{E} := L \cup \{\epsilon\}$
- ϵ is the bottom–of–stack symbol, and
- $\mathfrak{Re} : S \to \mathbb{R}^+_0$ is the reward function defined as follows.

$$\mathfrak{Re}(s) := \begin{cases} f(\sigma) & \text{if } s = (\mathsf{Term}, \sigma) \\ q & \text{if } s = (\ell, \sigma) \wedge \mathsf{cmd}(\ell) = \mathsf{tick}(q) \\ 0 & \text{otherwise} \end{cases}$$

Expected resource usage. The expected resource usage of the program $(c, \mathscr{D})$ is the expected reward collected when the associated pushdown MDP $\mathfrak{M}^f_\sigma[\![(c, \mathscr{D})]\!]$ eventually reaches the set of terminated states $(\mathsf{Term}, \_)$ from the starting state $s_0 = (\ell_0, \sigma)$, denoted $\mathsf{ExpRew}^{\mathfrak{M}^f_\sigma[\![c, \mathscr{D}]\!]}(\mathsf{Term})$. Formally, it is defined as

$$\begin{cases} \infty & \text{if } \inf_{\mathfrak{S}} \sum_{\widehat{\pi} \in \Pi(s_0, \mathsf{Term})} \mathbb{P}^{\mathfrak{M}^f_{\mathfrak{S},\sigma}[\![c, \mathscr{D}]\!]}(\widehat{\pi}) < 1 \\ \sup_{\mathfrak{S}} \sum_{\widehat{\pi} \in \Pi(s_0, \mathsf{Term})} \mathbb{P}^{\mathfrak{M}^f_{\mathfrak{S},\sigma}[\![c, \mathscr{D}]\!]}(\widehat{\pi}) \cdot \mathfrak{Re}(\widehat{\pi}) & \text{otherwise} \end{cases}$$

where $\mathfrak{S}$ is a *scheduler* for the pushdown MDP mapping a finite sequence of states to an action. Intuitively, it resolves the non-determinism by giving an action given a sequence of states that has been visited. Thus, $\mathfrak{S}$ induces a Markov chain, denoted $\mathfrak{M}^f_{\mathfrak{S},\sigma}[\![c, \mathscr{D}]\!]$, from $\mathfrak{M}^f_\sigma[\![c, \mathscr{D}]\!]$. $\Pi(s_0, \mathsf{Term})$ denotes the set of all finite paths from s_0 to some state (Term, σ') in

$$\frac{}{(\downarrow,\sigma) \xrightarrow{\tau,\epsilon,\epsilon,1} (\,\mathsf{Term}\,,\sigma)}\ (\textsc{S:Term})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{skip} \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:Skip})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{abort}}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell,\sigma)}\ (\textsc{S:Abort})$$

$$\frac{}{(\downarrow,\sigma) \xrightarrow{\tau,\ell',\epsilon,1} (\ell',\sigma)}\ (\textsc{S:Return})$$

$$\frac{[\![e]\!]_\sigma = \mathsf{true} \qquad \mathsf{cmd}\,(\ell) = \mathsf{assert}\ e \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:Assert})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{tick} \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:Tick})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{while}\ e\ c \qquad [\![e]\!]_\sigma = \mathsf{true} \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:LoopB})$$

$$\frac{\mathsf{cmd}\,(\ell) = c_1 \oplus_p c_2 \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,p} (\ell',\sigma)}\ (\textsc{S:PIfL})$$

$$\frac{\mathsf{cmd}\,(\ell) = c_1 \oplus_p c_2 \qquad \mathsf{scmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1-p} (\ell',\sigma)}\ (\textsc{S:PIfR})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{while}\ e\ c \qquad [\![e]\!]_\sigma = \mathsf{false} \qquad \mathsf{scmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:LoopE})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{id} = e \qquad \mathsf{fcmd}\,(\ell) = \ell' \qquad \sigma' = \sigma[[\![e]\!]_\sigma/\ \mathsf{id}\,]}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma')}\ (\textsc{S:Assign})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{id} = e\ \mathsf{bop}\ R \qquad \mathsf{fcmd}\,(\ell) = \ell' \qquad [\![\mu_R : v]\!] = p > 0 \qquad \sigma' = \sigma[[\![e]\!]_\sigma\ \mathsf{bop}\ v/\ \mathsf{id}\,]}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,p} (\ell',\sigma')}\ (\textsc{S:Sample})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{if} \star c_1\ \mathsf{else}\ c_2 \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\mathsf{Th},\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:NonDetL})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{if} \star c_1\ \mathsf{else}\ c_2 \qquad \mathsf{scmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\mathsf{El},\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:NonDetR})$$

$$\frac{\mathsf{cmd}\,(\ell) = \mathsf{call}\ P \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma.\ell',1} (\,\mathsf{init}\,(\mathscr{D}),\sigma)}\ (\textsc{S:Call})$$

$$\frac{[\![e]\!]_\sigma = \mathsf{true} \qquad \mathsf{cmd}\,(\ell) = \mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2 \qquad \mathsf{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:IfT})$$

$$\frac{[\![e]\!]_\sigma = \mathsf{false} \qquad \mathsf{cmd}\,(\ell) = \mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2 \qquad \mathsf{scmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)}\ (\textsc{S:IfF})$$

Figure 9. Rules of the probabilistic pushdown transition relation.

$\mathfrak{M}^f_{\mathfrak{S},\sigma}[\![c,\mathscr{D}]\!]$, $\mathbb{P}^{\mathfrak{M}^f_\sigma[\![c,\mathscr{D}]\!]}(\widehat{\pi})$ is the probability of the finite path $\widehat{\pi}$. And $\mathfrak{Re}(\widehat{\pi})$ is the *cumulative reward* collected along $\widehat{\pi}$.

B Weakest pre-expectation transformer

A weakest pre-expectation (WP) calculus [42, 69] expresses the resource usage of program $(c,\mathscr{D})$ using an *expected runtime transformer* given in continuation-passing style. The transformer used to analyze our language is defined in Table 2; it operates on the set of functions $\mathbb{T} := \{f \mid f : \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}\}$, usually called *expectations*. In our case, it is a good intuition to think of expectations as mere potential functions. More precisely, the transformer $\mathsf{ert}\,[c,\mathscr{D}](f)(\sigma)$ computes the expected number of ticks consumed by the program $(c,\mathscr{D})$ from the input state σ and followed by a computation that has an expected tick consumption given by f. Because f is evaluated in the final state and $\mathsf{ert}\,[c,\mathscr{D}](f)$ is evaluated in the initial state, they are called the *pre-* and *post-expectation*, respectively. If one chooses the post-expectation f to be the constantly zero function then $\mathsf{ert}\,[c,\mathscr{D}](\mathbf{0})$ is the expected number of ticks for the program.

Definition of the expected cost transformer. The rules defining the expected cost transformer follow the structure of the command c. We describe the intuition behind a few rules of the transformer. If c is $\mathsf{tick}\,(q)$, the expected cost for c followed by a computation of expected cost f is $\mathbf{q} + f$, because the (deterministic) cost of the $\mathsf{tick}\,(q)$ command is precisely q. For a sequence statement, $\mathsf{ert}\,[c_1; c_2,\mathscr{D}]$ is defined as the application of $\mathsf{ert}\,[c_1,\mathscr{D}]$ to the expected value obtained from $\mathsf{ert}\,[c_2,\mathscr{D}]$; this is the usual continuation-passing style definition. For a conditional statement, $\mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2,\mathscr{D}]$ is defined as the expected cost of the branch that will be executed. For a non-deterministic choice, $\mathsf{ert}\,[\,\mathsf{if} \star c_1\ \mathsf{else}\ c_2,\mathscr{D}]$ is the maximum between the expected costs of two branches. For a probabilistic branching, $\mathsf{ert}\,[c_1 \oplus_p c_2,\mathscr{D}]$ is the weighted sum of the expected costs of two branches. Similarly, the expected cost of a sampling assignment is defined by considering all the outcomes weighted according the random variable's distribution. Lastly, the expected cost of loops and procedure calls are expressed using least fixed points. For procedure calls, an auxiliary cost transformer $\mathsf{ert}\,[\cdot]^\sharp_X(f)$ is needed (See Section C.2). It is parameterized over another expected cost transformer $X : \mathbb{T} \to \mathbb{T}$. Its definition is almost identical to the one of the regular expected cost transformer except for procedure calls where $\mathsf{ert}\,[\,\mathsf{call}\ P]^\sharp_X(f) = X(f)$. The justification that the fixed points used in the definition exist can be found in the previous work [42, 69].

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

Table 2. Definition of the expected cost transformer $\mathsf{ert}\,.\,\mathbb{E}_{\mu_R}[h]:=\sum_v\mathbb{P}(R=v)\cdot h(v)$ represents the expected value of the random variable h w.r.t the distribution μ_R. $\max\{f_1,f_2\}:=\lambda\sigma.\ \max\{f_1(\sigma),f_2(\sigma)\}$. $\mathsf{lfp}\,X.F(X)$ is the least fixed point of the function F.

Example use of the transformer. For example, let c be the body of the example loop *rdwalk1* from Section 3.1, and let f be the expectation function $2x$. Then the expected cost transformer $\mathsf{ert}\,[c,\mathscr{D}](f)$ is computed as follows.

$$\begin{aligned}
&\mathsf{ert}\,[x=x-1\oplus_{3/4}x=x+1;\mathsf{tick}\,(1),\mathscr{D}]\,(2x)\\
&=\mathsf{ert}\,[x=x-1\oplus_{3/4}x=x+1,\mathscr{D}]\,(\mathsf{ert}\,[\mathsf{tick}\,(1),\mathscr{D}]\,(2x))\\
&=\mathsf{ert}\,[x=x-1\oplus_{3/4}x=x+1,\mathscr{D}]\,(1+2x)\\
&=\tfrac{3}{4}\,\mathsf{ert}\,[x=x-1,\mathscr{D}]\,(1+2x)+\\
&\quad\tfrac{1}{4}\,\mathsf{ert}\,[x=x+1,\mathscr{D}]\,(1+2x)\\
&=\tfrac{3}{4}(1+2x-2)+\tfrac{1}{4}(1+2x+2)\\
&=2x=f
\end{aligned}$$

In fact, this computation has established that f is an invariant for the body of the loop. Because the expected cost of loops is defined as a fixed point, finding invariants is critical for the analysis of programs with loops. In general, however, finding exact invariants like the one we just found is a hard problem. Instead, one can find a so-called upper invariant, and those provide upper bounds on the loop's cost. Inferring such upper invariants is precisely the role of the rule Q:Loop in our system.

Soundness of the transformer. The following theorem states the soundness of the expected cost transformer with respect to the MDP-based semantics.

Theorem B.1 (Soundness of the transformer). *Let $(c,\mathscr{D})$ be a probabilistic program and $f\in\mathbb{T}$ be an expectation. Then, for every program state $\sigma\in\Sigma$, the following holds*

$$\mathsf{ExpRew}^{\mathfrak{M}^{f}_{\sigma}[\![c,\mathscr{D}]\!]}(\mathsf{Term})=\mathsf{ert}\,[c,\mathscr{D}](f)(\sigma)$$

Proof. By induction on the command c. The details can be found in [60, 76]. □

C Bounded loops and recursive procedure calls

C.1 Bounded loops

The expected cost transformer for while loops is defined using the fixed point techniques. Alternatively, we can express the expected cost transformer for loops using bounded loops. A bounded loop is obtained by successively unrolling the loop up to a finite number of executions of the loop body.

Lemma C.1. *Let c be a command w.r.t a declaration $\mathscr{D}$. Then*

$$\mathsf{ert}\,[\,\mathsf{while}\ e\ c,\mathscr{D}]=\mathsf{ert}\,[\,\mathsf{if}\ e\ \{c;\mathsf{while}\ e\ c\}\ \mathsf{else}\ \ \mathsf{skip}\,,\mathscr{D}]$$

Proof. For all $f\in\mathbb{T}$, consider the following characteristic function

$$F_f(X)=[\![e:\mathsf{true}]\!]\cdot\mathsf{ert}\,[c,\mathscr{D}](X)+[\![e:\mathsf{false}]\!]\cdot f$$

We reason as follows

$$\begin{aligned}
&\mathsf{ert}\,[\,\mathsf{while}\ e\ c,\mathscr{D}](f)\\
&\quad\dagger\ \text{Table 2}\ \dagger\\
=\ &\mathsf{lfp}\ F_f\\
&\quad\dagger\ \text{Definition of fixed point}\ \dagger\\
=\ &F_f(\mathsf{lfp}\ F_f)\\
=\ &[\![e:\mathsf{true}]\!]\cdot\mathsf{ert}\,[c,\mathscr{D}](\mathsf{lfp}\ F_f)+[\![e:\mathsf{false}]\!]\cdot f\\
&\quad\dagger\ \text{Table 2}\ \dagger\\
=\ &[\![e:\mathsf{true}]\!]\cdot\mathsf{ert}\,[c,\mathscr{D}](\mathsf{ert}\,[\,\mathsf{while}\ e\ c,\mathscr{D}](f))+\\
&[\![e:\mathsf{false}]\!]\cdot f\\
&\quad\dagger\ \text{Table 2}\ \dagger\\
=\ &[\![e:\mathsf{true}]\!]\cdot\mathsf{ert}\,[c;\mathsf{while}\ e\ c,\mathscr{D}](f)+[\![e:\mathsf{false}]\!]\cdot f\\
&\dagger\ \ \mathsf{ert}\,[\,\mathsf{skip}\,,\mathscr{D}](f)=f\ \dagger\\
=\ &[\![e:\mathsf{true}]\!]\cdot\mathsf{ert}\,[c;\mathsf{while}\ e\ c,\mathscr{D}](f)+\\
&[\![e:\mathsf{false}]\!]\cdot\mathsf{ert}\,[\,\mathsf{skip}\,,\mathscr{D}](f)\\
&\quad\dagger\ \text{Table 2}\ \dagger\\
=\ &\mathsf{ert}\,[\,\mathsf{if}\ e\ \{c;\mathsf{while}\ e\ c\}\ \mathsf{else}\ \ \mathsf{skip}\,,\mathscr{D}](f)
\end{aligned}$$

□

The n^{th} *bounded* execution of a while loop command, denoted $\text{while}^k\ e\ c$, is defined as follows.

$$\text{while}^0 e\ c := \text{abort}$$
$$\text{while}^{n+1} e\ c := \text{if } e\ \{c; \text{while}^n e\ c\} \text{ else skip}$$

The following theorem states that the expected cost transformer can be expressed via the supremum of a sequence of bounded executions.

Theorem C.2. *Let $\mathscr{D}$ be a declaration, the following holds for all $n \in \mathbb{N}$ and $f \in \mathbb{T}$.*

$$\sup_n \text{ert}[\text{while}^n\ e\ c, \mathscr{D}](f) = \text{lfp}\, X.(\llbracket e : \text{true} \rrbracket \cdot \text{ert}[c, \mathscr{D}](X) + \llbracket e : \text{false} \rrbracket \cdot f)$$

Proof. For all $f \in \mathbb{T}$, consider the following characteristic function

$$F_f(X) = \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c, \mathscr{D}](X) + \llbracket e : \text{false} \rrbracket \cdot f$$

Let $C_n := \text{while}^n e\ c$, we first prove that for every $n \in \mathbb{N}$, $\text{ert}[C_n, \mathscr{D}](f) = F_f^n(\mathbf{0})$, where $F_f^0 := \text{id}$ and $F_f^{k+1} := F_f \circ F_f^k$. The proof is done by induction on n.

- *Base case.* It is intermediately satisfied because
$$\text{ert}[\text{abort}, \mathscr{D}](f) = \mathbf{0} = F_f^0(\mathbf{0})$$
- *Induction case.* Assume that $\text{ert}[C_n, \mathscr{D}](f) = F_f^n(\mathbf{0})$, we reason as follows

$$\begin{aligned}
& \text{ert}[C_{n+1}, \mathscr{D}](f) \\
& \quad \dagger \text{ Definition of bounded loops } \dagger \\
= \ & \text{ert}[\text{if } e\ \{c; \text{while}^n e\ c\} \text{ else skip}, \mathscr{D}](f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
= \ & \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c; \text{while}^n e\ c, \mathscr{D}](f) + \llbracket e : \text{false} \rrbracket \cdot f \\
& \quad \dagger \text{ Table 2 } \dagger \\
= \ & \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c, \mathscr{D}](\text{ert}[\text{while}^n e\ c, \mathscr{D}](f)) + \\
& \llbracket e : \text{false} \rrbracket \cdot f \\
& \quad \dagger \text{ By I.H } \dagger \\
= \ & \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c, \mathscr{D}](F_f^n(\mathbf{0})) + \llbracket e : \text{false} \rrbracket \cdot f \\
& \quad \dagger \text{ Definition of } F_f^{k+1} \ \dagger \\
= \ & F_f^{n+1}(\mathbf{0})
\end{aligned}$$

F_f is monotone because of the monotonicity of ert. Therefore, using Kleene's Fixed Point Theorem, if holds that

$$\begin{aligned}
\sup_n \text{ert}[C_n, \mathscr{D}](f) &= \sup_n F_f^n(\mathbf{0}) = \text{lfp}\ X.F_f(X) \\
&= \text{ert}[\text{while } e\ c, \mathscr{D}](f)
\end{aligned}$$

□

C.2 Characterization of procedure call

The detailed definition of the characteristic function for (recursive) procedure call is given in Table 3. The following lemma says that if a procedure P has a closed body (e.g., there is no procedure calls) then the characteristic function w.r.t the expected cost transformer of the body of P gives exactly the expected cost transformer of the command call P.

Lemma C.3. *For every command c and closed command c', the following holds where the declaration $\mathscr{D}(P) = c'$.*

$$\text{ert}[c]^\sharp_{\text{ert}[c', \mathscr{D}]} = \text{ert}[c, \mathscr{D}]$$

Proof. The proof is done by induction on the structure of c. If c has the form that is different from call P, then by I.H, the definition of the expected cost transformer, and the characteristic function, it follows directly. For instance, we illustrate the proof with the conditional and loop commands. If c is of the form if $e\ c_1$ else c_2. For all $f \in \mathbb{T}$, we reason as follows.

$$\begin{aligned}
& \text{ert}[c]^\sharp_{\text{ert}[c', \mathscr{D}]}(f) \\
& \quad \dagger \text{ Table 3 } \dagger \\
= \ & \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c_1]^\sharp_{\text{ert}[c', \mathscr{D}]}(f) + \\
& \llbracket e : \text{false} \rrbracket \cdot \text{ert}[c_2]^\sharp_{\text{ert}[c', \mathscr{D}]}(f) \\
& \quad \dagger \text{ By I.H for } \text{ert}[c_1]^\sharp_{\text{ert}[c', \mathscr{D}]} \text{ and } \text{ert}[c_2]^\sharp_{\text{ert}[c', \mathscr{D}]} \ \dagger \\
= \ & \llbracket e : \text{true} \rrbracket \cdot \text{ert}[c_1, \mathscr{D}](f) + \llbracket e : \text{false} \rrbracket \cdot \text{ert}[c_2, \mathscr{D}](f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
= \ & \text{ert}[c, \mathscr{D}](f)
\end{aligned}$$

If c is of the form while $e\ c_1$, then we have the following for all $f \in \mathbb{T}$.

$$\begin{aligned}
& \text{ert}[c]^\sharp_{\text{ert}[c', \mathscr{D}]}(f) \\
& \quad \dagger \text{ Table 3 } \dagger \\
= \ & \text{lfp}\, Y.(\llbracket e : \text{true} \rrbracket \cdot \text{ert}[c_1]^\sharp_{\text{ert}[c', \mathscr{D}]}(Y) + \llbracket e : \text{false} \rrbracket \cdot f) \\
& \quad \dagger \text{ By I.H for } \text{ert}[c_1]^\sharp_{\text{ert}[c', \mathscr{D}]}(Y) \ \dagger \\
= \ & \text{lfp}\, Y.(\llbracket e : \text{true} \rrbracket \cdot \text{ert}[c_1, \mathscr{D}](Y) + \llbracket e : \text{false} \rrbracket \cdot f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
= \ & \text{ert}[c, \mathscr{D}](f)
\end{aligned}$$

We consider the case that c is of the form call P. For all $n \geq 1$, $\text{call}^{\mathscr{D}}_n P = c'[\text{call}^{\mathscr{D}}_{n-1} P / \text{call}\, P] = c'$ since c' is closed. Hence, for all $f \in \mathbb{T}$, we have the following.

$$\begin{aligned}
& \text{ert}[\text{call } P, \mathscr{D}](f) \\
& \quad \dagger \text{ Theorem } C.5 \ \dagger \\
= \ & \sup_n \text{ert}[\text{call}^{\mathscr{D}}_n P](f) \\
& \quad \dagger \ \text{ert}[\text{call}^{\mathscr{D}}_0 P](f) = \mathbf{0};\ \text{call}^{\mathscr{D}}_{n+1} P = c' \ \dagger \\
= \ & \sup_n \text{ert}[\text{call}^{\mathscr{D}}_{n+1} P](f) = \sup_n \text{ert}[c', \mathscr{D}](f) \\
& \quad \dagger \text{ The supremum of constant sequence } \dagger \\
= \ & \text{ert}[c', \mathscr{D}](f) \\
& \quad \dagger \text{ Table 3 } \dagger \\
= \ & \text{ert}[\text{call}\, P]^\sharp_{\text{ert}[c', \mathscr{D}]}(f)
\end{aligned}$$

□

C.3 Syntactic replacement of procedure call

Table 4 gives the formal inductive definition of the syntactic replacement of procedure calls $c[c' / \text{call}\, P]$ on the structure of the command c. The following lemma says the property of the replacement by a closed command w.r.t the expected cost transformer ert, where the declaration $\mathscr{D}(P) = c'$.

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

Table 3. Characterization of procedure call.

c	$c[c'/\,\mathrm{call}\,P]$
skip , abort , assert e, weaken , tick (q), id $= e$, id $= e$ bop R	c
call P	c'
if e c_1 else c_2	if e $c_1[c'/\,\mathrm{call}\,P]$ else $c_2[c'/\,\mathrm{call}\,P]$
if $\star$ c_1 else c_2	if $\star$ $c_1[c'/\,\mathrm{call}\,P]$ else $c_2[c'/\,\mathrm{call}\,P]$
$c_1 \oplus_p c_2$	$c_1[c'/\,\mathrm{call}\,P] \oplus_p c_2[c'/\,\mathrm{call}\,P]$
$c_1; c_2$	$c_1[c'/\,\mathrm{call}\,P]; c_2[c'/\,\mathrm{call}\,P]$
while e c	while e $c[c'/\,\mathrm{call}\,P]$

Table 4. Syntactic replacement of procedure call.

Lemma C.4. *For every command c and closed command c', the following holds*

$$\mathrm{ert}\,[c[c'/\,\mathrm{call}\,P], \mathscr{D}] = \mathrm{ert}\,[c, \mathscr{D}]$$

Proof. The proof is done by induction on the structure of c. If c has the form that is different from call P, then by I.H, the definition of the expected cost transformer, and the syntactic replacement of procedure calls, it follows directly. For instance, we illustrate the proof with the conditional and loop commands. If c is of the form if e c_1 else c_2. We reason as follows.

$$\begin{aligned}
&\mathrm{ert}\,[c[c'/\,\mathrm{call}\,P], \mathscr{D}]\\
&\quad\dagger \text{ Table 4 } \dagger\\
=\ &\mathrm{ert}\,[\,\mathrm{if}\ e\ c_1[c'/\,\mathrm{call}\,P]\ \mathrm{else}\ c_2[c'/\,\mathrm{call}\,P], \mathscr{D}]\\
&\quad\dagger \text{ Table 2 } \dagger\\
=\ &[\![e : \mathrm{true}]\!]\cdot \mathrm{ert}\,[c_1[c'/\,\mathrm{call}\,P], \mathscr{D}] +\\
&[\![e : \mathrm{false}]\!]\cdot \mathrm{ert}\,[c_2[c'/\,\mathrm{call}\,P], \mathscr{D}]\\
&\quad\dagger \text{ By I.H for } \mathrm{ert}\,[c_1[c'/\,\mathrm{call}\,P], \mathscr{D}],\ \mathrm{ert}\,[c_2[c'/\,\mathrm{call}\,P], \mathscr{D}] \ \dagger\\
=\ &[\![e : \mathrm{true}]\!]\cdot \mathrm{ert}\,[c_1, \mathscr{D}] + [\![e : \mathrm{false}]\!]\cdot \mathrm{ert}\,[c_2, \mathscr{D}]\\
&\quad\dagger \text{ Table 2 } \dagger\\
=\ &\mathrm{ert}\,[c, \mathscr{D}]
\end{aligned}$$

If c is of the form while e c_1, then for all $f \in \mathbb{T}$, we have the following.

$$\begin{aligned}
&\mathrm{ert}\,[c[c'/\,\mathrm{call}\,P], \mathscr{D}](f)\\
&\quad\dagger \text{ Table 4 } \dagger\\
=\ &\mathrm{ert}\,[\,\mathrm{while}\ e\ c_1[c'/\,\mathrm{call}\,P]](f)\\
&\quad\dagger \text{ Table 2 } \dagger\\
=\ &\mathrm{lfp}\,X.([\![e : \mathrm{true}]\!]\cdot \mathrm{ert}\,[c_1[c'/\,\mathrm{call}\,P], \mathscr{D}](X) +\\
&[\![e : \mathrm{false}]\!]\cdot f)\\
&\quad\dagger \text{ By I.H for } \mathrm{ert}\,[c_1[c'/\,\mathrm{call}\,P], \mathscr{D}] \ \dagger\\
=\ &\mathrm{lfp}\,X.([\![e : \mathrm{true}]\!]\cdot \mathrm{ert}\,[c_1, \mathscr{D}](X) + [\![e : \mathrm{false}]\!]\cdot f)\\
&\quad\dagger \text{ Table 2 } \dagger\\
=\ &\mathrm{ert}\,[c, \mathscr{D}](f)
\end{aligned}$$

We consider the case that c is of the form call P. For all $n \geq 1$, $\mathrm{call}^{\mathscr{D}}_n P = c'[\,\mathrm{call}^{\mathscr{D}}_{n-1} P/\,\mathrm{call}\,P] = c'$ since c' is closed.

Hence, for all $f \in \mathbb{T}$, we have the following.

$$\begin{array}{ll} & \mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}](f) \\ & \dagger \text{ Theorem } C.5 \dagger \\ = & \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) \\ & \dagger\ \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_0 P](f) = \mathbf{0};\ \mathsf{call}^{\mathscr{D}}_{n+1} P = c' \dagger \\ = & \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_{n+1} P](f) = \sup_n \mathsf{ert}\,[c', \mathscr{D}](f) \\ & \dagger \text{ The supremum of constant sequence } \dagger \\ = & \mathsf{ert}\,[c', \mathscr{D}](f) \\ & \dagger\ \mathsf{call}\ P[c'/\mathsf{call}\ P] = c' \dagger \\ = & \mathsf{ert}\,[\,\mathsf{call}\ P[c'/\mathsf{call}\ P], \mathscr{D}](f) \end{array}$$

□

C.4 Finite approximation for procedure call

The expected cost transformer for (recursive) procedure calls is defined using the fixed point techniques. Alternatively, we borrow the approach in [76] to express the expected cost transformer for procedure calls as the limit of its finite approximations, or truncations, and show that they are equivalent. Let P be a procedure, the n^{th} *inlining* call of P w.r.t a declaration $\mathscr{D}$, denoted $\mathsf{call}^{\mathscr{D}}_n P$, is defined as follows.

$$\begin{array}{l} \mathsf{call}^{\mathscr{D}}_0 P := \mathsf{abort} \\ \mathsf{call}^{\mathscr{D}}_{n+1} P := \mathscr{D}(P)[\,\mathsf{call}^{\mathscr{D}}_n P/\,\mathsf{call}\ P] \end{array}$$

where $c[c'/\,\mathsf{call}\ P]$ is defined in Table 4. Intuitively, $\mathsf{call}^{\mathscr{D}}_n P$ is a sequence of approximations of $\mathsf{call}\ P$ where the "worst" approximation $\mathsf{call}^{\mathscr{D}}_0 P$, while the approximation gets more precise when n increases.

The expected cost transformer for procedure calls using the limit of its finite approximation is equivalent to the transformer defined using the fixed point techniques as stated by the following theorem.

Theorem C.5 (Limit of finite approximations). *Let P be a procedure w.r.t a declaration $\mathscr{D}$, the following holds for all $n \in \mathbb{N}$.*

$$\sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P] = \mathsf{lfp}\ X.(\,\mathsf{ert}\,[\mathscr{D}(P)]^\sharp_X)$$

Proof. For all $f \in \mathbb{T}$, consider the following characteristic function

$$F_f(X) = \mathsf{ert}\,[\mathscr{D}(P)]^\sharp_X(f)$$

where $\mathsf{ert}\,[c]^\sharp_X(f)$ is defined in Table 3. We first prove that for every $n \in \mathbb{N}$, $\mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) = F^n_f(\mathbf{0})$, where $F^0_f := \mathsf{id}$ and $F^{k+1}_f := F_f \circ F^k_f$. The proof is done by induction on n.

- *Base case.* It is intermediately satisfied because

$$\mathsf{ert}\,[\,\mathsf{abort}, \mathscr{D}](f) = \mathbf{0} = F^0_f(\mathbf{0})$$

- *Induction case.* Assume that $\mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) = F^n_f(\mathbf{0})$, we reason as follows

$$\begin{array}{ll} & \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_{n+1} P](f) \\ & \dagger \text{ Definition of inlining } \dagger \\ = & \mathsf{ert}\,[\mathscr{D}(P)[\,\mathsf{call}^{\mathscr{D}}_n P/\ \mathsf{call}\ P]](f) \\ & \dagger \text{ Lemma C.4 for closed command } \mathsf{call}^{\mathscr{D}}_n P \dagger \\ = & \mathsf{ert}\,[\mathscr{D}(P), \mathscr{D}](f) \\ & \dagger \text{ Lemma C.3 for closed command } \mathsf{call}^{\mathscr{D}}_n P \dagger \\ = & \mathsf{ert}\,[\mathscr{D}(P)]^\sharp_{\mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P]}(f) \\ & \dagger \text{ By I.H } \dagger \\ = & \mathsf{ert}\,[\mathscr{D}(P)]^\sharp_{F^n_f(\mathbf{0})}(f) \\ & \dagger \text{ Definition of } F_f \dagger \\ = & F_f(F^n_f(\mathbf{0})) \\ & \dagger \text{ Definition of } F^{n+1}_f \dagger \\ = & F^{n+1}_f(\mathbf{0}) \end{array}$$

F_f is monotone because of the monotonicity of $\mathsf{ert}^\sharp_X[\cdot]$. Therefore, using Kleene's Fixed Point Theorem, if holds that

$$\sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P] = \mathsf{lfp}\ X.(\,\mathsf{ert}\,[\mathscr{D}(P)]^\sharp_X) = \mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}]$$

□

D Proof of the soundness

D.1 Potential relax

Lemma D.1 (Potential relax). *For each program state σ such that $\sigma \models \Gamma$, if $Q \succeq_\Gamma Q'$, then $\Phi_Q(\sigma) \geq \Phi_{Q'}(\sigma)$.*

Proof. Lemma D.1 is proved using the rule Q:Relax as follows. Let $B = (b_1, \cdots, b_n)^\intercal$ be the set of all based functions, then we have

$$\begin{array}{lllll} \Phi_Q(\sigma) & = & \langle Q \cdot B\rangle(\sigma) & = & \sum_{i=1}^n q_i \cdot b_i(\sigma) \\ \Phi_{Q'}(\sigma) & = & \langle Q' \cdot B\rangle(\sigma) & = & \sum_{i=1}^n q'_i \cdot b_i(\sigma) \\ \Phi_{F_k}(\sigma) & = & \langle F_k \cdot B\rangle(\sigma) & = & \sum_{i=1}^n f^k_i \cdot b_i(\sigma) \end{array}$$

Consider any program state σ such that $\sigma \models \Gamma$, we reason as follows

$$\begin{array}{ll} & (q'_1, \cdots, q'_n) \\ & \dagger\ Q' = Q - F\vec{u} \text{ and matrix multiplication } \dagger \\ = & (q_1, \cdots, q_n) - (\sum_{k=1}^N f^k_1 \cdot u_k, \cdots, \sum_{k=1}^N f^k_n \cdot u_k) \\ & \dagger \text{ Vector subtraction } \dagger \\ = & ((q_1 - \sum_{k=1}^N f^k_1 \cdot u_k), \cdots, (q_n - \sum_{k=1}^N f^k_n \cdot u_k)) \end{array}$$

$$\begin{array}{ll} & \Phi_{Q'}(\sigma) \\ & \dagger \text{ Observation above } \dagger \\ = & \sum_{i=1}^n (q_i - \sum_{k=1}^N f^k_i \cdot u_k) \cdot b_i(\sigma) \\ & \dagger \text{ Algebra } \dagger \\ = & \sum_{i=1}^n (q_i \cdot b_i(\sigma) - \sum_{k=1}^N f^k_i \cdot u_k \cdot b_i(\sigma)) \\ = & \sum_{i=1}^n q_i \cdot b_i(\sigma) - \sum_{i=1}^n \sum_{k=1}^N f^k_i \cdot u_k \cdot b_i(\sigma) \\ = & \sum_{i=1}^n q_i \cdot b_i(\sigma) - \sum_{k=1}^N (\sum_{i=1}^n f^k_i \cdot u_k \cdot b_i(\sigma)) \\ & \dagger\ \forall i.\sigma.\Phi_{F_i}(\sigma) \geq 0; \forall i.u_i \geq 0 \dagger \\ \leq & \sum_{i=1}^n q_i \cdot b_i(\sigma) = \Phi_Q(\sigma) \end{array}$$

□

D.2 Soundness of the automatic analysis

Let $(c, \mathscr{D})$ be a program, the proof is done by induction on the program structure and the derivation using the inference rules.

Skip For all program states $\sigma \in \Sigma$, we have

$$\begin{aligned} & \mathrm{ert}\,[\,\mathrm{skip}\,,\mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) \\ & \dagger\ \text{Table 2}\ \dagger \\ =\ & \mathscr{T}(\Gamma;Q)(\sigma) \le \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

Abort For all program states $\sigma \in \Sigma$, we have

$$\begin{aligned} & \mathrm{ert}\,[\,\mathrm{abort}\,,\mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) \\ & \dagger\ \text{Table 2}\ \dagger \\ =\ & \mathbf{0}(\sigma) \le \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

Assert Consider any program state $\sigma \in \Sigma$, if $\sigma \not\models \Gamma$, then it follows

$$\begin{aligned} & \mathrm{ert}\,[\,\mathrm{assert}\ e,\mathscr{D}](\mathscr{T}(\Gamma \wedge e;Q))(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ \le\ & \infty = \mathscr{T}(\Gamma;Q) \end{aligned}$$

If $\sigma \models \Gamma$, we have

$$\begin{aligned} & \mathrm{ert}\,[\,\mathrm{assert}\ e,\mathscr{D}](\mathscr{T}(\Gamma \wedge e;Q))(\sigma) \\ & \dagger\ \text{Table 2}\ \dagger \\ =\ & [\![e : \mathrm{true}]\!]\cdot\mathscr{T}(\Gamma;Q)(\sigma) \\ \le\ & \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

Weaken Consider any state $\sigma \in \Sigma$, if $\sigma \models \Gamma_2'$ then $\sigma \models \Gamma_1'$ because $\Gamma_2' \models \Gamma_1'$. We have $Q_2' \succeq_{\Gamma_2'} Q_1'$, it holds that

$$\begin{aligned} & \mathscr{T}(\Gamma_2';Q_2')(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \Phi_{Q_2'}(\sigma) \\ & \dagger\ \text{Lemma D.1}\ \dagger \\ \ge\ & \Phi_{Q_1'}(\sigma) = \mathscr{T}(\Gamma_1';Q_1')(\sigma) \end{aligned}$$

If $\sigma \not\models \Gamma_2'$, then we get

$$\begin{aligned} & \mathscr{T}(\Gamma_2';Q_2')(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \infty \ge \mathscr{T}(\Gamma_1';Q_1')(\sigma) \end{aligned}$$

Therefore, we have $\mathscr{T}(\Gamma_2';Q_2')(\sigma) \ge \mathscr{T}(\Gamma_1';Q_1')(\sigma)$ for all $\sigma \in \Sigma$. Similarly, it follows $\mathscr{T}(\Gamma_1;Q_1)(\sigma) \ge \mathscr{T}(\Gamma_2;Q_2)(\sigma)$ for all $\sigma \in \Sigma$. For all states $\sigma \in \Sigma$, we get

$$\begin{aligned} & \mathscr{T}(\Gamma_1;Q_1)(\sigma) \\ & \dagger\ \text{Observation above}\ \dagger \\ \ge\ & \mathscr{T}(\Gamma_2;Q_2)(\sigma) \\ & \dagger\ \text{By I.H for } C\ \dagger \\ \ge\ & \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma_2';Q_2'))(\sigma) \\ & \dagger\ \text{Observation above and monotonicity of ert}\ \dagger \\ \ge\ & \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma_1';Q_1'))(\sigma) \end{aligned}$$

Tick Consider any state $\sigma \in \Sigma$ such that $\sigma \not\models \Gamma$, then

$$\begin{aligned} & \mathscr{T}(\Gamma;Q)(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \infty \ge \mathrm{ert}\,[\,\mathrm{tick}\,(q),\mathscr{D}](\mathscr{T}(\Gamma;Q-q))(\sigma) \end{aligned}$$

For all states $\sigma \in \Sigma$ such that $\sigma \models \Gamma$, we have

$$\begin{aligned} & \mathrm{ert}\,[\,\mathrm{tick}\,(q),\mathscr{D}](\mathscr{T}(\Gamma;Q-q))(\sigma) \\ & \dagger\ \text{Table 2}\ \dagger \\ =\ & q + \mathscr{T}(\Gamma;Q-q)(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & q + \Phi_Q(\sigma) - q \le \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

Assignment Suppose c is of the form $x = e$. Hence, the automatic analysis derivation ends with an application of the rule Q:Assign. Consider any program state σ such that $\sigma \not\models \Gamma[e/x]$,

$$\begin{aligned} & \mathscr{T}(\Gamma[e/x];Q)(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \infty \ge \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma;Q'))(\sigma) \end{aligned}$$

If $\sigma \models \Gamma[e/x]$, then we reason as follows

$$\begin{aligned} & \mathrm{ert}\,[x = e,\mathscr{D}](\mathscr{T}(\Gamma;Q'))(\sigma) \\ & \dagger\ \text{Table 2}\ \dagger \\ =\ & \mathscr{T}(\Gamma;Q')(\sigma[e/x]) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \max(\Gamma[e/x](\sigma),\Phi_{Q'}(\sigma[e/x])) = \Phi_{Q'}(\sigma[e/x]) \\ & \dagger\ \text{Definition of potential function and rule Q:Assign}\ \dagger \\ =\ & \textstyle\sum_{j\in\mathscr{S}_{x=e}} q_j'\cdot b_j[e/x](\sigma) + \sum_{j\notin\mathscr{S}_{x=e}} 0\cdot b_j[e/x](\sigma) \\ & \dagger\ \text{Rule Q:Assign}\ \dagger \\ =\ & \textstyle\sum_{j\in\mathscr{S}_{x=e}} q_j'\cdot \sum_i a_{i,j}\cdot b_i(\sigma) = \Phi_Q(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \mathscr{T}(\Gamma[e/x];Q) \end{aligned}$$

Sampling Suppose c is of the form $x = e$ bop R, where R is a random variable distributed by the probability distribution μ_R and $R \in [a,b]$. Hence, the automatic analysis derivation ends with an application of the rule Q:Sample. Let $\Gamma^i = \Gamma'[e \text{ bop } v_i/x]$, then for all i, we have $\Gamma \models \Gamma_i$. Consider any program state σ such that $\sigma \not\models \Gamma$,

$$\begin{aligned} & \mathscr{T}(\Gamma;Q)(\sigma) \\ & \dagger\ \text{Definition of translation function}\ \dagger \\ =\ & \infty \ge \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma) \end{aligned}$$

If $\sigma \models \Gamma$, then for all i, we have $\sigma \models \Gamma^i$ and the following hold

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q_i)(\sigma) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \Phi_{Q_i}(\sigma) \\ & \dagger \text{ By I.H for assignment } \dagger \\ \geq & \mathsf{ert}\,[x = e \text{ bop } v_i, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathscr{T}(\Gamma'; Q')(\sigma[e \text{ bop } v_i/x]) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \max(\Gamma^i(\sigma), \Phi_{Q'}(\sigma[e \text{ bop } v_i/x])) \\ = & \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]) \end{array}$$

$$\begin{array}{ll} & \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.\mathscr{T}(\Gamma'; Q')(\sigma[e \text{ bop } v/x])) \\ & \dagger \text{ Definition of expectation } \dagger \\ = & \sum_i [\![\mu_R : v_i]\!] \cdot (\max(\Gamma^i(\sigma), \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]))) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \sum_i p_i \cdot \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]) \\ & \dagger \text{ Observation above } \dagger \\ \leq & \sum_i p_i \cdot \Phi_{Q_i}(\sigma) \\ & \dagger \text{ By rule Q:Sample } \dagger \\ = & \Phi_Q(\sigma) = \mathscr{T}(\Gamma; Q)(\sigma) \end{array}$$

If Suppose c is of the form if e c_1 else c_2, thus the automatic analysis derivation ends with an application of the rule Q:If. Consider any program state σ such that $\sigma \not\models \Gamma$,

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \infty \geq \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

If $\sigma \models \Gamma \wedge e$, then we have

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \mathscr{T}(\Gamma \wedge e; Q)(\sigma) \\ & \dagger \text{ By I.H for } c_1 \dagger \\ \geq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger\ [\![e : \text{false}]\!](\sigma) = 0; [\![e : \text{true}]\!](\sigma) = 1 \dagger \\ = & [\![e : \text{true}]\!](\sigma) \cdot \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) + \\ & [\![e : \text{false}]\!](\sigma) \cdot \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

If $\sigma \models \Gamma \wedge \neg e$, then we get

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \mathscr{T}(\Gamma \wedge \neg e; Q)(\sigma) \\ & \dagger \text{ By I.H for } c_2 \dagger \\ \geq & \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger\ [\![e : \text{false}]\!](\sigma) = 1; [\![e : \text{true}]\!](\sigma) = 0 \dagger \\ = & [\![e : \text{true}]\!](\sigma) \cdot \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) + \\ & [\![e : \text{false}]\!](\sigma) \cdot \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

Nondeterministic branching Suppose c is of the form if $\star$ c_1 else c_2, thus the automatic analysis derivation ends with an application of the rule Q:NonDet. Consider any program state $\sigma \in \Sigma$, we get

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ By I.H for } c_1 \dagger \\ \geq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ By I.H for } c_2 \dagger \\ \geq & \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Algebra } \dagger \\ \geq & \max\{\mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma), \\ & \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma)\} \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

Probabilistic branching Suppose c is of the form $c_1 \oplus_p c_2$, thus the automatic analysis ends with an application of the rule Q:PIf. Consider any program state σ such that $\sigma \not\models \Gamma$, we have

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ Definition of translation function } \dagger \\ = & \infty \geq \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

If $\sigma \models \Gamma$, then it follows

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ Definition of translation function and rule Q:PIf } \dagger \\ = & \Phi_Q(\sigma) = p \cdot \Phi_{Q_1}(\sigma) + (1-p) \cdot \Phi_{Q_2}(\sigma) \\ = & p \cdot \mathscr{T}(\Gamma; Q_1)(\sigma) + (1-p) \cdot \mathscr{T}(\Gamma; Q_2)(\sigma) \\ & \dagger \text{ By I.H for } c_1 \text{ and } c_2 \dagger \\ \geq & p \cdot \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) + \\ & (1-p) \cdot \mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \end{array}$$

Sequence Suppose c is of the form $c_1; c_2$, thus the automatic analysis derivation ends with an application of the rule Q:Seq. Consider any program state $\sigma \in \Sigma$, we get

$$\begin{array}{ll} & \mathscr{T}(\Gamma; Q)(\sigma) \\ & \dagger \text{ By I.H for } c_1 \dagger \\ \geq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & \dagger \text{ By I.H for } c_2 \text{ and monotonicity of ert } \dagger \\ \geq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](\mathscr{T}(\Gamma''; Q'')))(\sigma) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma''; Q''))(\sigma) \end{array}$$

Loop

Suppose c is of the form while e c_1, thus the automatic analysis ends with an application of the rule Q:Loop. For any program state σ, we consider the characteristic function $F_{\mathscr{T}(\Gamma \wedge \neg e; Q)}(\mathscr{T}(\Gamma; Q))$ defined as

$$[\![e : \text{true}]\!] \cdot \mathsf{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma; Q)) + [\![e : \text{false}]\!] \cdot \mathscr{T}(\Gamma \wedge \neg e; Q)$$

If $\sigma \not\models \Gamma$, then

$$\begin{aligned} & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q))(\sigma) \\ & \quad \dagger \text{ Definition of translation function } \dagger \\ \leq\ & \infty = \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

If $\sigma \models \Gamma \wedge \neg e$, then we have

$$\begin{aligned} & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q))(\sigma) \\ =\ & [\![e : \text{true}]\!](\sigma)\cdot \text{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) + \\ & [\![e : \text{false}]\!](\sigma)\cdot\mathscr{T}(\Gamma\wedge\neg e;Q)(\sigma) \\ & \dagger\ [\![e : \text{false}]\!](\sigma) = 1; [\![e : \text{true}]\!](\sigma) = 0\ \dagger \\ =\ & \mathscr{T}(\Gamma\wedge\neg e;Q)(\sigma) \\ & \quad \dagger \text{ Definition of translation function } \dagger \\ =\ & \Phi_Q(\sigma) = \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

If $\sigma \models \Gamma \wedge e$, we get

$$\begin{aligned} & \mathscr{T}(\Gamma\wedge e;Q))(\sigma) \\ & \quad \dagger \text{ Definition of translation function } \dagger \\ =\ & \mathscr{T}(\Gamma;Q))(\sigma) = \Phi_Q(\sigma) \\ & \quad \dagger \text{ By I.H for } c_1 \dagger \\ \geq\ & \text{ert}\,[c_1, \mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) \\ & \dagger\ [\![e : \text{false}]\!](\sigma) = 0; [\![e : \text{true}]\!](\sigma) = 1\ \dagger \\ =\ & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q))(\sigma) \end{aligned}$$

Thus, for all program states $\sigma \in \Sigma$, we have

$$\begin{aligned} & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q))(\sigma) && \leq \mathscr{T}(\Gamma;Q)(\sigma) \\ & \quad \dagger \text{ Definition of } \sqsubseteq \dagger \\ \Leftrightarrow\ & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q)) && \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \quad \dagger \text{ Park's Theorem}^4\ [84] \dagger \\ \Rightarrow\ & \text{lfp}\ F_{\mathscr{T}(\Gamma\wedge\neg e;Q)} && \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \quad \dagger \text{ Table 2 } \dagger \\ \Leftrightarrow\ & \text{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma\wedge\neg e;Q)) && \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \quad \dagger \text{ Definition of } \sqsubseteq \dagger \\ \Leftrightarrow\ & \text{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma\wedge\neg e;Q'))(\sigma) && \leq \mathscr{T}(\Gamma;Q)(\sigma) \end{aligned}$$

Following Theorem C.2, the expected cost transformer for loops can be expressed as

$$\text{ert}\,[\,\text{while}\ e\ c_1, \mathscr{D}] = \sup_n \text{ert}\,[\,\text{while}^n\ e\ c_1, \mathscr{D}]$$

Alternatively, we can prove the soundness of the rule Q:Loop by showing that for all program states σ

$$\forall n.\, \text{ert}\,[\,\text{while}^n\ e\ c_1, \mathscr{D}](\mathscr{T}(\Gamma\wedge\neg e;Q))(\sigma) \leq \mathscr{T}(\Gamma;Q)(\sigma)$$

Procedure call

Suppose c is of the form call P, thus the automatic analysis ends with applications of the rules Q:Call and ValidCtx. Since by Theorem C.5, $\text{ert}\,[\,\text{call}\ P, \mathscr{D}] = \sup_n \text{ert}\,[\,\text{call}_n^{\mathscr{D}} P]$, where the n^{th}-inlining of procedure call $\text{call}_n^{\mathscr{D}} P$ is defined in Section C.4. To prove that

$$\text{ert}\,[\,\text{call}\ P, \mathscr{D}](\mathscr{T}(\Gamma';Q'+c))(\sigma) \leq \mathscr{T}(\Gamma;Q+c)(\sigma)$$

for all program states σ, it is sufficient to show that

$$\forall n.\, \text{ert}\,[\,\text{call}_n^{\mathscr{D}} P](\mathscr{T}(\Gamma';Q'+c))(\sigma) \leq \mathscr{T}(\Gamma;Q+c)(\sigma)$$

[4] If $H : \mathscr{D} \to \mathscr{D}$ is a continuous function over an ω-cpo $(\mathscr{D}, \sqsubseteq)$ with bottom element, then $H(d) \sqsubseteq d$ implies $\text{lfp}\ H \sqsubseteq d$ for all $d \in \mathscr{D}$.

The proof is done by induction on the natural value n.

- *Base case.* It is intermediately satisfied because

$$\begin{aligned} & \text{ert}\,[\,\text{call}_0^{\mathscr{D}} P](\mathscr{T}(\Gamma';Q'+c))(\sigma) \\ & \quad \dagger \text{ Definition of } \text{call}_n^{\mathscr{D}} P \dagger \\ =\ & \text{ert}\,[\,\text{abort}\,](\mathscr{T}(\Gamma';Q'+c))(\sigma) \\ & \quad \dagger \text{ Table 2 } \dagger \\ =\ & \mathbf{0}(\sigma) \leq \mathscr{T}(\Gamma;Q+c)(\sigma) \end{aligned}$$

- *Induction case.* We reason as follows

$$\begin{aligned} & \dagger \text{ Rules Q:Call, ValidCtx, and by I.H for } \mathscr{D}(P) \dagger \\ & \dagger \text{ where the current body of } P \text{ is } \text{call}_n^{\mathscr{D}} P \dagger \\ & \text{ert}\,[\mathscr{D}(P), \mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma) \leq \mathscr{T}(\Gamma;Q)(\sigma) \\ & \dagger \text{ Algebra } \dagger \\ \Leftrightarrow\ & \text{ert}\,[\mathscr{D}(P), \mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma) + x \leq \mathscr{T}(\Gamma;Q)(\sigma) + x \\ & \quad \dagger \text{ Definition of translation function } \dagger \\ \Leftrightarrow\ & \text{ert}\,[\mathscr{D}(P), \mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma) + x \leq \mathscr{T}(\Gamma;Q+x)(\sigma) \\ & \quad \dagger \text{ Property of ert } \dagger \\ \Rightarrow\ & \text{ert}\,[\mathscr{D}(P), \mathscr{D}](\mathscr{T}(\Gamma';Q') + x)(\sigma) \leq \mathscr{T}(\Gamma;Q+x)(\sigma) \\ & \quad \dagger \text{ Definition of translation function } \dagger \\ \Leftrightarrow\ & \text{ert}\,[\mathscr{D}(P), \mathscr{D}](\mathscr{T}(\Gamma';Q'+x))(\sigma) \leq \mathscr{T}(\Gamma;Q+x)(\sigma) \\ & \quad \dagger \text{ Lemma C.4 where } \text{call}_n^{\mathscr{D}} P \text{ is closed } \dagger \\ \Leftrightarrow\ & \text{ert}\,[\mathscr{D}(P)[\,\text{call}_n^{\mathscr{D}} P/\,\text{call}\ P], \mathscr{D}](\mathscr{T}(\Gamma';Q'+c))(\sigma) \\ & \leq \mathscr{T}(\Gamma;Q+c)(\sigma) \\ & \quad \dagger \text{ Definition of replacement in Table 4 } \dagger \\ \Leftrightarrow\ & \text{ert}\,[\,\text{call}_{n+1}^{\mathscr{D}} P](\mathscr{T}(\Gamma';Q'+c))(\sigma) \leq \mathscr{T}(\Gamma;Q+c)(\sigma) \end{aligned}$$

E Basic properties of the expected cost transformer

We write $\sqsubseteq$ to denote the point-wise order relation between expectations. Formally, let $f, g \in \mathbb{T}$, $f \sqsubseteq g$ if and only if $f(\sigma) \leq g(\sigma)$ for all program states $\sigma \in \Sigma$.

E.1 The ω-complete partial order

Lemma E.1 (ω-cpo). *$(\mathbb{T}, \sqsubseteq)$ is an ω-complete partial order (cpo) with bottom element $\lambda\sigma.\mathbf{0}$ and top element $\lambda\sigma.\infty$.*

Proof. To prove that $(\mathbb{T}, \sqsubseteq)$ is an ω-cpo, we need to show that $(\mathbb{T}, \sqsubseteq)$ is a partial order set and then to show that every ω-chain $S \subseteq \mathbb{T}$ has a supremum. Such a supremum can be constructed by taking the point-wise supremum $\sup S = \lambda\sigma.\ \sup\{f(\sigma) \mid f \in S\}$ which always exists because every subset of $\mathbb{R}_0^+$ has a supremum. Hence, it remains to prove $(\mathbb{T}, \sqsubseteq)$ is a partial order set.

Antisymmetry $f \sqsubseteq g \wedge g \sqsubseteq f$

$$\begin{aligned} & \quad \dagger \text{ Definition of } \sqsubseteq \dagger \\ \Leftrightarrow\ & \forall\sigma.f(\sigma) \leq g(\sigma) \wedge g(\sigma) \leq f(\sigma) \\ \Leftrightarrow\ & \forall\sigma.f(\sigma) \leq g(\sigma) \leq f(\sigma) \\ \Rightarrow\ & \forall\sigma.f(\sigma) = g(\sigma) \\ \Leftrightarrow\ & f = g \end{aligned}$$

Reflexivity $\forall\sigma.f(\sigma) = f(\sigma)$

$$\begin{aligned} \Rightarrow\ & \forall\sigma.f(\sigma) \leq f(\sigma) \\ & \quad \dagger \text{ Definition of } \sqsubseteq \dagger \\ \Leftrightarrow\ & f \sqsubseteq f \end{aligned}$$

Transitivity $f \sqsubseteq g \wedge g \sqsubseteq h$

$$\begin{array}{ll} & \dagger \text{ Definition of } \sqsubseteq \ \dagger \\ \Leftrightarrow & \forall \sigma. f(\sigma) \leq g(\sigma) \wedge g(\sigma) \leq h(\sigma) \\ \Leftrightarrow & \forall \sigma. f(\sigma) \leq g(\sigma) \leq h(\sigma) \\ \Rightarrow & \forall \sigma. f(\sigma) \leq h(\sigma) \\ & \dagger \text{ Definition of } \sqsubseteq \ \dagger \\ \Leftrightarrow & f \sqsubseteq h \end{array}$$

□

E.2 Basic algebraic properties

The transformer ert enjoys many algebraic properties such as *monotonicity*, *propagation of constants*, *scaling*, and *continuity* properties which we summarize in the following lemma.

Lemma E.2 (Algebraic properties of ert). *For every program $(c, \mathscr{D})$, every $f, g \in \mathbb{T}$, every increasing ω-chain $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$, every constant expectation $\mathbf{k} = \lambda\sigma.k, k \in \mathbb{R}_0^+$, and every constant $r \in \mathbb{R}_0^+$, the following properties hold.*

- *Continuity:*

$$\sup_n \text{ert}[c, \mathscr{D}](f_n) = \text{ert}[c, \mathscr{D}](\sup_n f_n)$$

- *Monotonicity:*

$$f \sqsubseteq g \Rightarrow \text{ert}[c, \mathscr{D}](f) \sqsubseteq \text{ert}[c, \mathscr{D}](g)$$

- *Propagation of constants:*

$$\text{ert}[c, \mathscr{D}](\mathbf{k} + f) \sqsubseteq \mathbf{k} + \text{ert}[c, \mathscr{D}](f)$$

- *Sub-additivity:*

$$\text{ert}[c, \mathscr{D}](f + g) \sqsubseteq \text{ert}[c, \mathscr{D}](f) + \text{ert}[c, \mathscr{D}](g)$$

if c is non-determinism-free

- *Scaling:*

$$\min(1, r) \cdot \text{ert}[c, \mathscr{D}](f) \sqsubseteq \text{ert}[c, \mathscr{D}](r \cdot f)$$
$$\text{ert}[c, \mathscr{D}](r \cdot f) \sqsubseteq \max(1, r) \cdot \text{ert}[c, \mathscr{D}](f)$$

- *Preservation of ∞:*

$$\text{ert}[c, \mathscr{D}](\infty) = \infty \quad \text{if } c \text{ is abort-free}$$

E.2.1 Continuity

Let $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ be an increasing ω-chain. The proof is done by induction on the structure of the command c.

Skip

$$\begin{array}{ll} & \sup_n \text{ert}[\text{skip}, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 } \dagger \\ = & \sup_n f_n \\ & \dagger \text{ Table 2 } \dagger \\ = & \text{ert}[\text{skip}, \mathscr{D}](\sup_n f_n) \end{array}$$

Abort

$$\begin{array}{ll} & \sup_n \text{ert}[\text{abort}, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 and algebra } \dagger \\ = & \mathbf{0} \\ = & \text{ert}[\text{abort}, \mathscr{D}](\sup_n f_n) \end{array}$$

Assert

$$\begin{array}{ll} & \sup_n \text{ert}[\text{assert } e, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 and algebra } \dagger \\ = & [\![e : \text{true}]\!] \cdot (\sup_n f_n) \\ = & \text{ert}[\text{assert } e, \mathscr{D}](\sup_n f_n) \end{array}$$

Weaken

$$\begin{array}{ll} & \sup_n \text{ert}[\text{weaken}, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 } \dagger \\ = & \sup_n f_n \\ & \dagger \text{ Table 2 } \dagger \\ = & \text{ert}[\text{weaken}, \mathscr{D}](\sup_n f_n) \end{array}$$

Tick

$$\begin{array}{ll} & \sup_n \text{ert}[\text{tick}(q), \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 } \dagger \\ = & \sup_n (\mathbf{q} + f_n) \\ & \dagger \text{ Algebra } \dagger \\ = & \mathbf{q} + \sup_n f_n \\ & \dagger \text{ Table 2 } \dagger \\ = & \text{ert}[\text{tick}(q), \mathscr{D}](\sup_n f_n) \end{array}$$

Assignment

$$\begin{array}{ll} & \sup_n \text{ert}[x = e, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 } \dagger \\ = & \sup_n (f_n[e/x]) = (\sup_n f_n)[e/x] \\ & \dagger \text{ Table 2 } \dagger \\ = & \text{ert}[x = e, \mathscr{D}](\sup_n f_n) \end{array}$$

Sampling The proof relies on the Lebesgue's Monotone Convergence Theorem (LMCT).

$$\begin{array}{ll} & \sup_n \text{ert}[x = e \text{ bop } R, \mathscr{D}](f_n) \\ & \dagger \text{ Table 2 } \dagger \\ = & \sup_n (\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v. f_n(\sigma[e \text{ bop } v/x]))) \\ & \dagger \text{ LMCT } \dagger \\ = & \lambda\sigma.\mathbb{E}_{\mu_R}(\sup_n \lambda v. f_n(\sigma[e \text{ bop } v/x])) \\ = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v. \sup_n f_n(\sigma[e \text{ bop } v/x])) \\ & \dagger \text{ Table 2 } \dagger \\ = & \text{ert}[x = e \text{ bop } R, \mathscr{D}](\sup_n f_n) \end{array}$$

If The proof relies on the Monotone Sequence Theorem (MST), that is, if $\langle a_n \rangle$ is a monotonic sequence in $\mathbb{R}_0^+$ then

$\sup_n a_n = \lim_{n\to\infty} a_n$.

$$\begin{aligned}
& \sup_n \operatorname{ert}[\text{if } e\ c_1 \text{ else } c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \sup_n ([\![e : \text{true}]\!] \cdot \operatorname{ert}[c_1, \mathscr{D}](f_n) + \\
& \quad [\![e : \text{false}]\!] \cdot \operatorname{ert}[c_2, \mathscr{D}](f_n)) \\
& \quad \dagger \text{ MST } \dagger \\
=\ & \lim_{n\to\infty} ([\![e : \text{true}]\!] \cdot \operatorname{ert}[c_1, \mathscr{D}](f_n) + \\
& \quad [\![e : \text{false}]\!] \cdot \operatorname{ert}[c_2, \mathscr{D}](f_n)) \\
=\ & [\![e : \text{true}]\!] \cdot \lim_{n\to\infty} \operatorname{ert}[c_1, \mathscr{D}](f_n) + \\
& \quad [\![e : \text{false}]\!] \cdot \lim_{n\to\infty} \operatorname{ert}[c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ MST } \dagger \\
=\ & [\![e : \text{true}]\!] \cdot \sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n) + \\
& \quad [\![e : \text{false}]\!] \cdot \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
=\ & [\![e : \text{true}]\!] \cdot \operatorname{ert}[c_1, \mathscr{D}](\sup_n f_n) + \\
& \quad [\![e : \text{false}]\!] \cdot \operatorname{ert}[c_2, \mathscr{D}](\sup_n f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\text{if } e\ c_1 \text{ else } c_2, \mathscr{D}](\sup)_n f_n
\end{aligned}$$

Nondeterministic branching

$$\begin{aligned}
& \sup_n \operatorname{ert}[\text{if } \star\ c_1 \text{ else } c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \sup_n(\max\{\operatorname{ert}[c_1, \mathscr{D}](f_n), \operatorname{ert}[c_2, \mathscr{D}](f_n)\}) \\
\sqsupseteq\ & \max\{\sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n), \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n)\} \\
& \quad \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
=\ & \max\{\operatorname{ert}[c_1, \mathscr{D}](\sup_n f_n), \operatorname{ert}[c_2, \mathscr{D}](\sup)_n f_n\} \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\text{if } \star\ c_1 \text{ else } c_2, \mathscr{D}](\sup_n f_n)
\end{aligned}$$

Let $A = \max\{\sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n), \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n)\}$. Assume that

$$A \sqsubset \sup_n(\max\{\operatorname{ert}[c_1, \mathscr{D}](f_n), \operatorname{ert}[c_2, \mathscr{D}](f_n)\})$$

Then there exists $m \in \mathbb{N}$ such that

$$\begin{aligned}
A \ \sqsubset\ & \max\{\operatorname{ert}[c_1, \mathscr{D}](f_m), \operatorname{ert}[c_2, \mathscr{D}](f_m)\} \\
& \quad \dagger \text{ Definition of supremum } \dagger \\
\sqsubseteq\ & \max\{\sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n), \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n)\} \\
=\ & A
\end{aligned}$$

Therefore, we get

$$\begin{aligned}
A \ =\ & \max\{\sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n), \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n)\} \\
=\ & \operatorname{ert}[\text{if } \star\ c_1 \text{ else } c_2, \mathscr{D}](\sup_n f_n) \\
\sqsupseteq\ & \sup_n(\max\{\operatorname{ert}[c_1, \mathscr{D}](f_n), \operatorname{ert}[c_2, \mathscr{D}](f_n)\}) \\
=\ & \sup_n \operatorname{ert}[\text{if } \star\ c_1 \text{ else } c_2, \mathscr{D}](f_n)
\end{aligned}$$

By observation above, it follows.

Probabilistic branching The proof relies on the Monotone Sequence Theorem (MST), that is, if $\langle a_n \rangle$ is a monotonic sequence in $\mathbb{R}_0^+$ then $\sup_n a_n = \lim_{n\to\infty} a_n$.

$$\begin{aligned}
& \sup_n \operatorname{ert}[c_1 \oplus_p c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \sup_n(p \cdot \operatorname{ert}[c_1, \mathscr{D}](f_n) + (1-p) \cdot \operatorname{ert}[c_2, \mathscr{D}](f_n)) \\
& \quad \dagger \text{ MST } \dagger \\
=\ & \lim_{n\to\infty}(p \cdot \operatorname{ert}[c_1, \mathscr{D}](f_n) + (1-p) \cdot \operatorname{ert}[c_2, \mathscr{D}](f_n)) \\
=\ & p \cdot \lim_{n\to\infty} \operatorname{ert}[c_1, \mathscr{D}](f_n) + (1-p) \cdot \lim_{n\to\infty} \operatorname{ert}[c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ MST } \dagger \\
=\ & p \cdot \sup_n \operatorname{ert}[c_1, \mathscr{D}](f_n) + (1-p) \cdot \sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
=\ & p \cdot \operatorname{ert}[c_1, \mathscr{D}](\sup_n f_n) + (1-p) \cdot \operatorname{ert}[c_2, \mathscr{D}](\sup_n f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[c_1 \oplus_p c_2, \mathscr{D}](\sup)_n f_n
\end{aligned}$$

Sequence

$$\begin{aligned}
& \sup_n \operatorname{ert}[c_1; c_2, \mathscr{D}](f_n) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \sup_n(\operatorname{ert}[c_1, \mathscr{D}](\operatorname{ert}[c_2, \mathscr{D}](f_n))) \\
& \quad \dagger \text{ By I.H on } c_1 \dagger \\
=\ & \operatorname{ert}[c_1, \mathscr{D}](\sup_n \operatorname{ert}[c_2, \mathscr{D}](f_n)) \\
& \quad \dagger \text{ By I.H on } c_2 \dagger \\
=\ & \operatorname{ert}[c_1, \mathscr{D}](\operatorname{ert}[c_2, \mathscr{D}](\sup_n f_n)) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[c_1; c_2, \mathscr{D}](\sup_n f_n)
\end{aligned}$$

Loop By I.H for $\text{while}^k e\ c$ (defined in Section C.1), we get

$$\sup_n \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}](f_n) = \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}](\sup_n f_n)$$

On the other hand, by Theorem C.2, $\operatorname{ert}[\text{while } e\ c, \mathscr{D}] = \sup_k \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}]$, where $\text{while}^k e\ c$ is the k^{th} *bounded* execution of a while loop command, we have

$$\begin{aligned}
& \operatorname{ert}[\text{while } e\ c, \mathscr{D}](\sup_n f_n) \\
=\ & \sup_k \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}](\sup_n f_n) \\
& \quad \dagger \text{ Observation above } \dagger \\
=\ & \sup_k(\sup_n \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}](f_n)) \\
=\ & \sup_n(\sup_k \operatorname{ert}[\text{while}^k e\ c, \mathscr{D}](f_n)) \\
& \quad \dagger \text{ By Theorem } C.2 \dagger \\
=\ & \sup_n(\operatorname{ert}[\text{while } e\ c, \mathscr{D}](f_n))
\end{aligned}$$

Procedure call Because $\text{call}_k^{\mathscr{D}} P$ (defined in Section C.4) is closed command for all $k \in \mathbb{N}$, by I.H we get

$$\sup_n \operatorname{ert}[\text{call}_k^{\mathscr{D}} P](f_n) = \operatorname{ert}[\text{call}_k^{\mathscr{D}} P](\sup_n f_n)$$

On the other hand, by Theorem C.5, we have

$$\operatorname{ert}[\text{call } P, \mathscr{D}] = \sup_k \operatorname{ert}[\text{call}_k^{\mathscr{D}} P]$$

where $\text{call}^{\mathscr{D}}_k P$ is the k^{th}-inlining of procedure call, we have

$$\begin{aligned}
& \text{ert}\,[\,\text{call}\ P, \mathscr{D}](\sup_n f_n) \\
= \;& \sup_k \text{ert}\,[\,\text{call}^{\mathscr{D}}_k P](\sup_n f_n) \\
& \dagger \text{ Observation above } \dagger \\
= \;& \sup_k(\sup_n \text{ert}\,[\,\text{call}^{\mathscr{D}}_k P](f_n)) \\
= \;& \sup_n(\sup_k \text{ert}\,[\,\text{call}^{\mathscr{D}}_k P](f_n)) \\
& \dagger \text{ By Theorem } C.5 \dagger \\
= \;& \sup_n(\text{ert}\,[\,\text{call}\ P, \mathscr{D}](f_n))
\end{aligned}$$

E.2.2 Monotonicity

The monotonicity follows from the continuity of ert as follows.

$$\begin{aligned}
\text{ert}\,[c, \mathscr{D}](g) & \quad \dagger\ f \sqsubseteq g\ \dagger \\
= \;& \text{ert}\,[c, \mathscr{D}](\sup\{f, g\}) \\
& \dagger \text{ Continuity } \dagger \\
= \;& \sup\{\text{ert}\,[c, \mathscr{D}](f), \text{ert}\,[c, \mathscr{D}](g)\} \\
& \dagger \text{ Definition of supremum } \dagger \\
\sqsupseteq \;& \text{ert}\,[c, \mathscr{D}](f)
\end{aligned}$$

E.2.3 Propagation of constants

The proof is done by induction on the structure of the command c.

Skip

$$\begin{aligned}
& \text{ert}\,[\,\text{skip}\,, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + f \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[\,\text{skip}\,, \mathscr{D}](f)
\end{aligned}$$

Abort

$$\begin{aligned}
& \text{ert}\,[\,\text{abort}\,, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 and algebra } \dagger \\
= \;& \mathbf{0} \\
\sqsubseteq \;& \mathbf{k} + \text{ert}\,[\,\text{abort}\,, \mathscr{D}](f)
\end{aligned}$$

Assert

$$\begin{aligned}
& \text{ert}\,[\,\text{assert}\ e, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 and algebra } \dagger \\
= \;& [\![e : \text{true}]\!]\cdot(\mathbf{k} + f) = [\![e : \text{true}]\!]\cdot\mathbf{k} + [\![e : \text{true}]\!]\cdot f \\
& \dagger\ [\![e : \text{true}]\!] \le 1\ \dagger \\
\sqsubseteq \;& \mathbf{k} + \text{ert}\,[\,\text{assert}\ e, \mathscr{D}](f)
\end{aligned}$$

Weaken

$$\begin{aligned}
& \text{ert}\,[\,\text{weaken}\,, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + f \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[\,\text{weaken}\,, \mathscr{D}](f)
\end{aligned}$$

Tick

$$\begin{aligned}
& \text{ert}\,[\,\text{tick}\,(q), \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{q} + \mathbf{k} + f \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[\,\text{tick}\,(q), \mathscr{D}](f)
\end{aligned}$$

Assignment

$$\begin{aligned}
& \text{ert}\,[x = e, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + f[e/x] \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[x = e, \mathscr{D}](f)
\end{aligned}$$

Sampling The proof relies on the linearity property of expectations (LPE).

$$\begin{aligned}
& \text{ert}\,[x = e\ \text{bop}\ R, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(\mathbf{k} + f)(\sigma[e\ \text{bop}\ v/x])) \\
& \dagger \text{ LPE and } \mathbf{k}(\sigma[e\ \text{bop}\ v/x]) = \mathbf{k}(\sigma)\ \dagger \\
= \;& \mathbf{k} + \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \text{bop}\ v/x])) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[x = e\ \text{bop}\ R, \mathscr{D}](f)
\end{aligned}$$

If

$$\begin{aligned}
& \text{ert}\,[\,\text{if}\ e\ c_1\ \text{else}\ c_2, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& [\![e : \text{true}]\!]\cdot \text{ert}\,[c_1, \mathscr{D}](\mathbf{k} + f) + \\
& [\![e : \text{false}]\!]\cdot \text{ert}\,[c_2, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
\sqsubseteq \;& [\![e : \text{true}]\!]\cdot(\mathbf{k} + \text{ert}\,[c_1, \mathscr{D}](f)) + \\
& [\![e : \text{false}]\!]\cdot(\mathbf{k} + \text{ert}\,[c_2, \mathscr{D}](f)) \\
& \dagger \text{ Algebra } \dagger \\
= \;& \mathbf{k} + [\![e : \text{true}]\!]\cdot \text{ert}\,[c_1, \mathscr{D}](f) + \\
& [\![e : \text{false}]\!]\cdot \text{ert}\,[c_2, \mathscr{D}](f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[\,\text{if}\ e\ c_1\ \text{else}\ c_2, \mathscr{D}](f)
\end{aligned}$$

Nondeterministic branching

$$\begin{aligned}
& \text{ert}\,[\,\text{if}\ \star\ c_1\ \text{else}\ c_2, \mathscr{D}](\mathbf{k} + f) \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \max\{\text{ert}\,[c_1, \mathscr{D}](\mathbf{k} + f), \text{ert}\,[c_2, \mathscr{D}](\mathbf{k} + f)\} \\
& \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
\sqsubseteq \;& \max\{\mathbf{k} + \text{ert}\,[c_1, \mathscr{D}](f), \mathbf{k} + \text{ert}\,[c_2, \mathscr{D}](f)\} \\
& \dagger \text{ Algebra } \dagger \\
= \;& \mathbf{k} + \max\{\text{ert}\,[c_1, \mathscr{D}](f), \text{ert}\,[c_2, \mathscr{D}](f)\} \\
& \dagger \text{ Table 2 } \dagger \\
= \;& \mathbf{k} + \text{ert}\,[\,\text{if}\ \star\ c_1\ \text{else}\ c_2, \mathscr{D}](f)
\end{aligned}$$

Probabilistic branching

$$\begin{aligned}
& \operatorname{ert}[c_1 \oplus_p c_2, \mathscr{D}](\mathbf{k}+f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & p\cdot \operatorname{ert}[c_1, \mathscr{D}](\mathbf{k}+f) + (1-p)\cdot \operatorname{ert}[c_2, \mathscr{D}](\mathbf{k}+f) \\
& \quad \dagger \text{ By I.H on } c_1 \text{ and } c_2 \ \dagger \\
\sqsubseteq\ & p\cdot(\mathbf{k}+\operatorname{ert}[c_1, \mathscr{D}](f)) + (1-p)\cdot(\mathbf{k}+\operatorname{ert}[c_2, \mathscr{D}](f)) \\
& \quad \dagger \text{ Algebra } \dagger \\
=\ & \mathbf{k} + p\cdot \operatorname{ert}[c_1, \mathscr{D}](f) + (1-p)\cdot \operatorname{ert}[c_2, \mathscr{D}](f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \mathbf{k} + \operatorname{ert}[c_1 \oplus_p c_2, \mathscr{D}](f)
\end{aligned}$$

Sequence

$$\begin{aligned}
& \operatorname{ert}[c_1; c_2, \mathscr{D}](\mathbf{k}+f) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[c_1, \mathscr{D}](\operatorname{ert}[c_2, \mathscr{D}](\mathbf{k}+f)) \\
& \quad \dagger \text{ By I.H on } c_2 \ \dagger \\
\sqsubseteq\ & \operatorname{ert}[c_1, \mathscr{D}](\mathbf{k}+\operatorname{ert}[c_2, \mathscr{D}](f)) \\
& \quad \dagger \text{ By I.H on } c_1 \ \dagger \\
\sqsubseteq\ & \mathbf{k}+\operatorname{ert}[c_1, \mathscr{D}](\operatorname{ert}[c_2, \mathscr{D}](f)) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \mathbf{k}+\operatorname{ert}[c_1; c_2, \mathscr{D}](f)
\end{aligned}$$

Loop Consider the characteristic function w.r.t the expectation f

$$F_f := [\![e : \text{true}]\!]\cdot \operatorname{ert}[c, \mathscr{D}](X) + [\![e : \text{false}]\!]\cdot f$$

We first need to show that $F_{\mathbf{k}+f}(\mathbf{k}+\operatorname{lfp} F_f) \sqsubseteq \mathbf{k}+\operatorname{lfp} F_f$. Then following Park's Theorem[5] [84], we get $\operatorname{lfp} F_{\mathbf{k}+f} \sqsubseteq \mathbf{k}+\operatorname{lfp} F_f$.

$$\begin{aligned}
& F_{\mathbf{k}+f}(\mathbf{k}+\operatorname{lfp} F_f) \\
& \quad \dagger \text{ Definition of } F_{\mathbf{k}+f} \ \dagger \\
=\ & [\![e : \text{true}]\!]\cdot \operatorname{ert}[c, \mathscr{D}](\mathbf{k}+\operatorname{lfp} F_f) + [\![e : \text{false}]\!]\cdot(\mathbf{k}+f) \\
& \quad \dagger \text{ By I.H on } c \ \dagger \\
\sqsubseteq\ & [\![e : \text{true}]\!]\cdot(\mathbf{k}+\operatorname{ert}[c, \mathscr{D}](\operatorname{lfp} F_f)) + [\![e : \text{false}]\!]\cdot(\mathbf{k}+f) \\
& \quad \dagger \text{ Algebra } \dagger \\
=\ & \mathbf{k} + [\![e : \text{true}]\!]\cdot \operatorname{ert}[c, \mathscr{D}](\operatorname{lfp} F_f) + [\![e : \text{false}]\!]\cdot f \\
& \quad \dagger \text{ Definition of } F_f \ \dagger \\
=\ & \mathbf{k} + F_f(\operatorname{lfp} F_f) \\
& \quad \dagger \text{ Definition of lfp } \dagger \\
=\ & \mathbf{k} + \operatorname{lfp} F_f
\end{aligned}$$

Procedure call Because $\operatorname{call}^{\mathscr{D}}_n P$ (defined in Section C.4) is closed command for all $n \in \mathbb{N}$, by I.H we get

$$\operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P](\mathbf{k}+f) \sqsubseteq \mathbf{k}+\operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P](f)$$

On the other hand, by Theorem C.5, we have

$$\operatorname{ert}[\operatorname{call} P, \mathscr{D}] = \sup_n \operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P]$$

where $\operatorname{call}^{\mathscr{D}}_n P$ is the n^{th}-inlining of procedure call, we have

$$\begin{aligned}
& \operatorname{ert}[\operatorname{call} P, \mathscr{D}](\mathbf{k}+f) \\
=\ & \sup_n \operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P](\mathbf{k}+f) \\
& \quad \dagger \text{ Observation above } \dagger \\
\sqsubseteq\ & \sup_n(\mathbf{k}+\operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P](f)) \\
=\ & \mathbf{k}+\sup_n \operatorname{ert}[\operatorname{call}^{\mathscr{D}}_n P](f) \\
& \quad \dagger \text{ By Theorem } C.5 \ \dagger \\
=\ & \mathbf{k}+\operatorname{ert}[\operatorname{call} P, \mathscr{D}](f)
\end{aligned}$$

E.2.4 Sub-additivity

The proof is done by induction on the structure of the command c. Note that c is *non-determinism* free, that is, c contains no non − deterministic commands.

Skip

$$\begin{aligned}
& \operatorname{ert}[\operatorname{skip}, \mathscr{D}](f+g) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & f+g \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\operatorname{skip}, \mathscr{D}](f) + \operatorname{ert}[\operatorname{skip}, \mathscr{D}](g)
\end{aligned}$$

Abort

$$\begin{aligned}
& \operatorname{ert}[\operatorname{abort}, \mathscr{D}](f+g) \\
& \quad \dagger \text{ Table 2 and algebra } \dagger \\
=\ & \mathbf{0} \\
=\ & \operatorname{ert}[\operatorname{abort}, \mathscr{D}](f) + \operatorname{ert}[\operatorname{abort}, \mathscr{D}](g)
\end{aligned}$$

Assert

$$\begin{aligned}
& \operatorname{ert}[\operatorname{assert} e, \mathscr{D}](f+g) \\
& \quad \dagger \text{ Table 2 and algebra } \dagger \\
=\ & [\![e : \text{true}]\!]\cdot(f+g) = [\![e : \text{true}]\!]\cdot f + [\![e : \text{true}]\!]\cdot g \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\operatorname{assert} e, \mathscr{D}](f) + \operatorname{ert}[\operatorname{assert} e, \mathscr{D}](g)
\end{aligned}$$

Weaken

$$\begin{aligned}
& \operatorname{ert}[\operatorname{weaken}, \mathscr{D}](f+g) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & f+g \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\operatorname{weaken}, \mathscr{D}](f) + \operatorname{ert}[\operatorname{weaken}, \mathscr{D}](g)
\end{aligned}$$

Tick

$$\begin{aligned}
& \operatorname{ert}[\operatorname{tick}(q), \mathscr{D}](f+g) \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \mathbf{q}+f+g \\
& \quad \dagger \text{ Algebra } \dagger \\
\sqsubseteq\ & \mathbf{q}+f+\mathbf{q}+g \\
& \quad \dagger \text{ Table 2 } \dagger \\
=\ & \operatorname{ert}[\operatorname{tick}(q), \mathscr{D}](f) + \operatorname{ert}[\operatorname{tick}(q), \mathscr{D}](g)
\end{aligned}$$

[5] If $H : \mathscr{D} \to \mathscr{D}$ is a continuous function over an ω-cpo $(\mathscr{D}, \sqsubseteq)$ with bottom element, then $H(d) \sqsubseteq d$ implies $\operatorname{lfp} H \sqsubseteq d$ for all $d \in \mathscr{D}$.

Assignment

$$\begin{array}{ll} & \mathsf{ert}\,[x = e, \mathscr{D}](f+g) \\ & \dagger \text{ Table 2 } \dagger \\ = & f[e/x] + g[e/x] \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[x = e, \mathscr{D}](f) + \mathsf{ert}\,[x = e, \mathscr{D}](g) \end{array}$$

Sampling The proof relies on the linearity property of expectations (LPE).

$$\begin{array}{ll} & \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](f+g) \\ & \dagger \text{ Table 2 } \dagger \\ = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(f+g)(\sigma[e \text{ bop } v/x])) \\ & \dagger \text{ Linearity of expectations } \dagger \\ = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e \text{ bop } v/x]))+ \\ & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.g(\sigma[e \text{ bop } v/x])) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](f) + \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](g) \end{array}$$

If

$$\begin{array}{ll} & \mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f+g) \\ & \dagger \text{ Table 2 } \dagger \\ = & [\![e : \text{true}]\!]\cdot \mathsf{ert}\,[c_1, \mathscr{D}](f+g)+ \\ & [\![e : \text{false}]\!]\cdot \mathsf{ert}\,[c_2, \mathscr{D}](f+g) \\ & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ \sqsubseteq & [\![e : \text{true}]\!]\cdot(\mathsf{ert}\,[c_1, \mathscr{D}](f) + \mathsf{ert}\,[c_1, \mathscr{D}](g))+ \\ & [\![e : \text{false}]\!]\cdot(\mathsf{ert}\,[c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_2, \mathscr{D}](g)) \\ & \dagger \text{ Table 2 and algebra } \dagger \\ = & \mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f) + \mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](g) \end{array}$$

Probabilistic branching

$$\begin{array}{ll} & \mathsf{ert}\,[c_1 \oplus_p c_2, \mathscr{D}](f+g) \\ & \dagger \text{ Table 2 } \dagger \\ = & p\cdot \mathsf{ert}\,[c_1, \mathscr{D}](f+g) + (1-p)\cdot \mathsf{ert}\,[c_2, \mathscr{D}](f+g) \\ & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ \sqsubseteq & p\cdot(\mathsf{ert}\,[c_1, \mathscr{D}](f) + \mathsf{ert}\,[c_1, \mathscr{D}](g))+ \\ & (1-p)\cdot(\mathsf{ert}\,[c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_2, \mathscr{D}](g)) \\ & \dagger \text{ Table 2 and algebra } \dagger \\ = & \mathsf{ert}\,[c_1 \oplus_p c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_1 \oplus_p c_2, \mathscr{D}](g) \end{array}$$

Sequence

$$\begin{array}{ll} & \mathsf{ert}\,[c_1; c_2, \mathscr{D}](f+g) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f+g)) \\ & \dagger \text{ By I.H on } c_2 \dagger \\ \sqsubseteq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_2, \mathscr{D}](g)) \\ & \dagger \text{ By I.H on } c_1 \dagger \\ \sqsubseteq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f)) + \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](g)) \\ & \dagger \text{ Table 2 } \dagger \\ = & \mathsf{ert}\,[c_1; c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_1; c_2, \mathscr{D}](g) \end{array}$$

Loop Consider the characteristic function w.r.t the expectation f

$$F_f := [\![e : \text{true}]\!]\cdot \mathsf{ert}\,[c, \mathscr{D}](X) + [\![e : \text{false}]\!]\cdot f$$

We first need to show that $F_{\mathbf{k}+f}(\mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g) \sqsubseteq \mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g$. Then following Park's Theorem [84], we get $\mathsf{lfp}\ F_{\mathbf{k}+f} \sqsubseteq \mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g$.

$$\begin{array}{ll} & F_{\mathbf{k}+f}(\mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g) \\ & \dagger \text{ Definition of } F_{\mathbf{k}+f} \dagger \\ = & [\![e : \text{true}]\!]\cdot \mathsf{ert}\,[c, \mathscr{D}](\mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g)+ \\ & [\![e : \text{false}]\!]\cdot(\mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g) \\ & \dagger \text{ By I.H on } c \dagger \\ \sqsubseteq & [\![e : \text{true}]\!]\cdot(\mathsf{ert}\,[c, \mathscr{D}](\mathsf{lfp}\ F_f) + \mathsf{ert}\,[c, \mathscr{D}](\mathsf{lfp}\ F_g))+ \\ & [\![e : \text{false}]\!]\cdot(\mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g) \\ & \dagger \text{ Algebra } \dagger \\ = & [\![e : \text{true}]\!]\cdot \mathsf{ert}\,[c, \mathscr{D}](\mathsf{lfp}\ F_f) + [\![e : \text{false}]\!]\cdot \mathsf{lfp}\ F_f+ \\ & [\![e : \text{true}]\!]\cdot \mathsf{ert}\,[c, \mathscr{D}](\mathsf{lfp}\ F_g) + [\![e : \text{false}]\!]\cdot \mathsf{lfp}\ F_g \\ & \dagger \text{ Definition of } F_f \text{ and } F_g \dagger \\ = & F_f(\mathsf{lfp}\ F_f) + F_g(\mathsf{lfp}\ F_g) \\ & \dagger \text{ Definition of lfp } \dagger \\ = & \mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g \end{array}$$

Procedure call Because $\mathsf{call}^{\mathscr{D}}_n P$ (defined in Section C.4) is closed command for all $n \in \mathbb{N}$, by I.H we get

$$\mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f+g) \sqsubseteq \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) + \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](g)$$

On the other hand, by Theorem C.5, we get

$$\mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}] = \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P]$$

where $\mathsf{call}^{\mathscr{D}}_n P$ is the n^{th}-inlining of procedure call, we have

$$\begin{array}{ll} & \mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}](f+g) \\ = & \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f+g) \\ & \dagger \text{ Observation above } \dagger \\ \sqsubseteq & \sup_n(\mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) + \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](g)) \\ = & \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](f) + \sup_n \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_n P](g) \\ & \dagger \text{ By Theorem C.5 } \dagger \\ = & \mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}](f) + \mathsf{ert}\,[\,\mathsf{call}\ P, \mathscr{D}](g) \end{array}$$

E.2.5 Scaling

The proof is done by induction on the structure of the command c.

Skip

$$\begin{array}{ll} & \min(1,r)\cdot f \sqsubseteq r\cdot f \sqsubseteq \max(1,r)\cdot f \\ & \dagger \text{ Table 2 } \dagger \\ \Leftrightarrow & \min(1,r)\cdot f \sqsubseteq \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot f \\ & \dagger \text{ Table 2 } \dagger \\ \Leftrightarrow & \min(1,r)\cdot \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](f) \sqsubseteq \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](r\cdot f) \\ & \sqsubseteq \max(1,r)\cdot \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](f) \end{array}$$

Abort

$$\begin{aligned}
&\min(1,r)\cdot\mathbf{0} \sqsubseteq r\cdot\mathbf{0} \sqsubseteq \max(1,r)\cdot\mathbf{0}\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{abort}\,,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[\,\mathsf{abort}\,,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[\,\mathsf{abort}\,,\mathscr{D}](f)
\end{aligned}$$

Assert

$$\begin{aligned}
&\min(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot f \sqsubseteq r\cdot[\![e:\ \mathsf{true}\,]\!]\cdot f\\
&\sqsubseteq \max(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot f\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{assert}\ e,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[\,\mathsf{assert}\ e,\mathscr{D}](r\cdot f)\\
&\sqsubseteq \max(1,r)\cdot \operatorname{ert}[\,\mathsf{assert}\ e,\mathscr{D}](f)
\end{aligned}$$

Weaken

$$\begin{aligned}
&\min(1,r)\cdot f \sqsubseteq r\cdot f \sqsubseteq \max(1,r)\cdot f\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot f \sqsubseteq \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot f\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](r\cdot f)\\
&\sqsubseteq \max(1,r)\cdot \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](f)
\end{aligned}$$

Tick

$$\begin{aligned}
&\mathbf{q} + \min(1,r)\cdot f \sqsubseteq \mathbf{q} + r\cdot f \sqsubseteq \mathbf{q} + \max(1,r)\cdot f\\
\Rightarrow\ &\min(1,r)(\mathbf{q}+f) \sqsubseteq \mathbf{q} + r\cdot f \sqsubseteq \max(1,r)(\mathbf{q}+f)\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{tick}\,(q),\mathscr{D}](f) \sqsubseteq \operatorname{ert}[\,\mathsf{tick}\,(q),\mathscr{D}](r\cdot f)\\
&\sqsubseteq \max(1,r)\cdot \operatorname{ert}[\,\mathsf{tick}\,(q),\mathscr{D}](f)
\end{aligned}$$

Assignment

$$\begin{aligned}
&\min(1,r)\cdot f[e/x] \sqsubseteq r\cdot f[e/x] \sqsubseteq \max(1,r)\cdot f[e/x]\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[x=e,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[x=e,\mathscr{D}](r\cdot f)\\
&\sqsubseteq \max(1,r)\cdot \operatorname{ert}[x=e,\mathscr{D}](f)
\end{aligned}$$

Sampling The proof relies on the linearity property of expectations (LPE).

$$\begin{aligned}
&\min(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&r\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&\max(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x]))\\
\Leftrightarrow\ &\min(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&\lambda\sigma.r\cdot\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&\max(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x]))\\
&\dagger \text{ LPE } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(r\cdot f)(\sigma[e\ \mathsf{bop}\ v/x])) \sqsubseteq\\
&\max(1,r)\cdot\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathsf{bop}\ v/x]))\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[x=e\ \mathsf{bop}\ R,\mathscr{D}](f) \sqsubseteq\\
&\operatorname{ert}[x=e\ \mathsf{bop}\ R,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[x=e\ \mathsf{bop}\ R,\mathscr{D}](f)
\end{aligned}$$

If

$$\begin{aligned}
&\dagger \text{ By I.H on } c_1 \dagger\\
\Rightarrow\ &\min(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[c_1,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f)\\
&\dagger \text{ Algebra } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f) \sqsubseteq\\
&[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f)\\
&\dagger \text{ By I.H on } c_2 \dagger\\
\Rightarrow\ &\min(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f) +\\
&\min(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f) \sqsubseteq\\
&[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](r\cdot f) + \operatorname{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f) +\\
&\max(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f)\\
&\dagger \text{ Algebra } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f) +\\
&\min(1,r)\cdot[\![e:\ \mathsf{false}\,]\!]\cdot \operatorname{ert}[c_2,\mathscr{D}](f) \sqsubseteq\\
&[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](r\cdot f) + [\![e:\ \mathsf{false}\,]\!]\cdot \operatorname{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot[\![e:\ \mathsf{true}\,]\!]\cdot \operatorname{ert}[c_1,\mathscr{D}](f) +\\
&\max(1,r)\cdot[\![e:\ \mathsf{false}\,]\!]\cdot \operatorname{ert}[c_2,\mathscr{D}](f)\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](f) \sqsubseteq\\
&\operatorname{ert}[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](f)
\end{aligned}$$

Nondeterministic branching

$$\begin{aligned}
&\dagger \text{ By I.H on } c_1 \dagger\\
\Rightarrow\ &\min(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[c_1,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f)\\
&\dagger \text{ By I.H on } c_2 \dagger\\
\Rightarrow\ &\min(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f)\\
&\dagger \text{ Algebra } \dagger\\
\Leftrightarrow\ &\max\{\min(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f),\\
&\min(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f)\}\\
&\sqsubseteq \max\{\operatorname{ert}[c_1,\mathscr{D}](r\cdot f), \operatorname{ert}[c_2,\mathscr{D}](r\cdot f)\} \sqsubseteq\\
&\max\{\max(1,r)\cdot \operatorname{ert}[c_1,\mathscr{D}](f),\\
&\max(1,r)\cdot \operatorname{ert}[c_2,\mathscr{D}](f)\}\\
\Leftrightarrow\ &\min(1,r)\cdot \max\{\operatorname{ert}[c_1,\mathscr{D}](f), \operatorname{ert}[c_2,\mathscr{D}](f)\} \sqsubseteq\\
&\max\{\operatorname{ert}[c_1,\mathscr{D}](r\cdot f), \operatorname{ert}[c_2,\mathscr{D}](r\cdot f)\} \sqsubseteq\\
&\max(1,r)\cdot \max\{\operatorname{ert}[c_1,\mathscr{D}](f), \operatorname{ert}[c_2,\mathscr{D}](f)\}\\
&\dagger \text{ Table 2 } \dagger\\
\Leftrightarrow\ &\min(1,r)\cdot \operatorname{ert}[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](f) \sqsubseteq\\
&\operatorname{ert}[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](r\cdot f) \sqsubseteq\\
&\max(1,r)\cdot \operatorname{ert}[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](f)
\end{aligned}$$

Probabilistic branching

$$
\begin{array}{ll}
& \dagger \text{ By I.H on } c_1 \dagger \\
\Rightarrow & \min(1,r)\cdot \text{ert}[c_1,\mathscr{D}](f) \sqsubseteq \text{ert}[c_1,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[c_1,\mathscr{D}](f) \\
& \dagger \text{ Algebra } \dagger \\
\Leftrightarrow & \min(1,r)\cdot p\cdot \text{ert}[c_1,\mathscr{D}](f) \sqsubseteq p\cdot \text{ert}[c_1,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot p\cdot \text{ert}[c_1,\mathscr{D}](f) \\
& \dagger \text{ By I.H on } c_2 \dagger \\
\Rightarrow & \min(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f) \sqsubseteq \text{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f) \\
& \dagger \text{ Algebra } \dagger \\
\Leftrightarrow & \min(1,r)\cdot(1-p)\cdot \text{ert}[c_2,\mathscr{D}](f) \sqsubseteq (1-p)\cdot \text{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot(1-p)\cdot \text{ert}[c_2,\mathscr{D}](f) \\
& \dagger \text{ By composing } \dagger \\
\Leftrightarrow & \min(1,r)\cdot(p\cdot \text{ert}[c_1,\mathscr{D}](f)+(1-p)\cdot \text{ert}[c_2,\mathscr{D}](f)) \sqsubseteq p\cdot \text{ert}[c_1,\mathscr{D}](r\cdot f)+(1-p)\cdot \text{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot(p\cdot \text{ert}[c_1,\mathscr{D}](f)+(1-p)\cdot \text{ert}[c_2,\mathscr{D}](f)) \\
& \dagger \text{ Table 2 } \dagger \\
\Leftrightarrow & \min(1,r)\cdot \text{ert}[c_1 \oplus_p c_2,\mathscr{D}](f) \sqsubseteq \text{ert}[c_1 \oplus_p c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[c_1 \oplus_p c_2,\mathscr{D}](f)
\end{array}
$$

Sequence

$$
\begin{array}{ll}
& \dagger \text{ By I.H on } c_2 \dagger \\
\Rightarrow & \min(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f) \sqsubseteq \text{ert}[c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f) \\
& \dagger \text{ Monotonicity of ert } \dagger \\
\Rightarrow & \text{ert}[c_1,\mathscr{D}](\min(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f)) \sqsubseteq \text{ert}[c_1,\mathscr{D}](\text{ert}[c_2,\mathscr{D}](r\cdot f)) \sqsubseteq \text{ert}[c_1,\mathscr{D}](\max(1,r)\cdot \text{ert}[c_2,\mathscr{D}](f)) \\
& \dagger \text{ By I.H on } c_1 \dagger \\
\Rightarrow & \min(1,r)\cdot \text{ert}[c_1,\mathscr{D}](\text{ert}[c_2,\mathscr{D}](f)) \sqsubseteq \text{ert}[c_1,\mathscr{D}](\text{ert}[c_2,\mathscr{D}](r\cdot f)) \sqsubseteq \max(1,r)\cdot \text{ert}[c_1,\mathscr{D}](\text{ert}[c_2,\mathscr{D}](f)) \\
& \dagger \text{ Table 2 } \dagger \\
\Leftrightarrow & \min(1,r)\cdot \text{ert}[c_1;c_2,\mathscr{D}](f) \sqsubseteq \text{ert}[c_1;c_2,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[c_1;c_2,\mathscr{D}](f)
\end{array}
$$

Loop Consider the characteristic function w.r.t the expectation f

$$F_f := [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](X) + [\![e:\ \text{false}]\!]\cdot f$$

For all $n \in \mathbb{N}$, we first need to show that

$$\min(1,r)\cdot \sup_n F_f^n(\mathbf{0}) \sqsubseteq \sup_n F_{r\cdot f}^n(\mathbf{0}) \sqsubseteq \max(1,r)\cdot \sup_n F_f^n(\mathbf{0})$$

where $F_f^0 := \text{id}$ and $F_f^{k+1} := F_f \circ F_f^k$. Because F_f and $F_{r\cdot f}$ are monotone because of the monotonicity of ert , then using Kleene's Fixed Point Theorem, if holds that

$$\min(1,r)\cdot \text{ert}[\text{while } e\ c,\mathscr{D}](f) \sqsubseteq \text{ert}[\text{while } e\ c,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[\text{while } e\ c,\mathscr{D}](f)$$

To prove we need to show the following holds for all $n \in \mathbb{N}$

$$\min(1,r)\cdot F_f^n(\mathbf{0}) \sqsubseteq F_{r\cdot f}^n(\mathbf{0}) \sqsubseteq \max(1,r)\cdot F_f^n(\mathbf{0})$$

The proof is done by induction on the natural value n.

- *Base case.* For $n = 0$, we have

$$
\begin{array}{ll}
& \min(1,r)\cdot F_f^0(\mathbf{0}) \sqsubseteq F_{r\cdot f}^0(\mathbf{0}) \sqsubseteq \max(1,r)\cdot F_f^0(\mathbf{0}) \\
\Leftrightarrow & \min(1,r)\cdot \mathbf{0} \sqsubseteq \mathbf{0} \sqsubseteq \max(1,r)\cdot \mathbf{0} \\
\Leftrightarrow & \mathbf{0} \sqsubseteq \mathbf{0} \sqsubseteq \mathbf{0}
\end{array}
$$

- *Induction case.* Assume that

$$
\begin{array}{ll}
& \dagger \text{ By I.H on } n \dagger \\
& \min(1,r)\cdot F_f^n(\mathbf{0}) \sqsubseteq F_{r\cdot f}^n(\mathbf{0}) \sqsubseteq \max(1,r)\cdot F_f^n(\mathbf{0}) \\
& \dagger \text{ Monotonicity of ert } \dagger \\
\Rightarrow & \text{ert}[c,\mathscr{D}](\min(1,r)\cdot F_f^n(\mathbf{0})) \sqsubseteq \text{ert}[c,\mathscr{D}](F_{r\cdot f}^n(\mathbf{0})) \sqsubseteq \text{ert}[c,\mathscr{D}](\max(1,r)\cdot F_f^n(\mathbf{0})) \\
& \dagger \text{ By I.H on } c \dagger \\
\Rightarrow & \min(1,r)\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \sqsubseteq \text{ert}[c,\mathscr{D}](F_{r\cdot f}^n(\mathbf{0})) \sqsubseteq \max(1,r)\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \\
& \dagger \text{ Algebra } \dagger \\
\Leftrightarrow & \min(1,r)\cdot [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \sqsubseteq [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_{r\cdot f}^n(\mathbf{0})) \sqsubseteq \max(1,r)\cdot [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \\
& \dagger \text{ By I.H on } n \dagger \\
\Rightarrow & [\![e:\ \text{false}]\!]\cdot \min(1,r)\cdot F_f^n(\mathbf{0}) + \min(1,r)\cdot [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \sqsubseteq [\![e:\ \text{false}]\!]\cdot F_{r\cdot f}^n(\mathbf{0}) + [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_{r\cdot f}^n(\mathbf{0})) \sqsubseteq [\![e:\ \text{false}]\!]\cdot \max(1,r)\cdot F_f^n(\mathbf{0}) + \max(1,r)\cdot [\![e:\ \text{true}]\!]\cdot \text{ert}[c,\mathscr{D}](F_f^n(\mathbf{0})) \\
& \dagger \text{ Definitions of } F_f \text{ and } F_{r\cdot f} \dagger \\
\Leftrightarrow & \min(1,r)\cdot F_f^{n+1}(\mathbf{0}) \sqsubseteq F_{r\cdot f}^{n+1}(\mathbf{0}) \sqsubseteq \max(1,r)\cdot F_f^{n+1}(\mathbf{0})
\end{array}
$$

Procedure call Because $\text{call}_n^{\mathscr{D}} P$ (defined in Section C.4) is closed command for all $n \in \mathbb{N}$, by I.H we get

$$\min(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f) \sqsubseteq \text{ert}[\text{call}_n^{\mathscr{D}} P](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f)$$

On the other hand, by Theorem C.5, $\text{ert}[\text{call } P,\mathscr{D}](r\cdot f) = \sup_n \text{ert}[\text{call}_n^{\mathscr{D}} P](r\cdot f)$, where $\text{call}_n^{\mathscr{D}} P$ is the n^{th}-inlining of procedure call, we have

$$
\begin{array}{ll}
& \sup_n(\min(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \sqsubseteq \sup_n \text{ert}[\text{call}_n^{\mathscr{D}} P](r\cdot f) \sqsubseteq \sup_n(\max(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \\
\Leftrightarrow & \sup_n(\min(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \sqsubseteq \text{ert}[\text{call } P,\mathscr{D}](r\cdot f) \sqsubseteq \sup_n(\max(1,r)\cdot \text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \\
\Leftrightarrow & \min(1,r)\cdot \sup_n(\text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \sqsubseteq \text{ert}[\text{call } P,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \sup_n(\text{ert}[\text{call}_n^{\mathscr{D}} P](f)) \\
& \dagger \text{ Theorem C.5 } \dagger \\
\Leftrightarrow & \min(1,r)\cdot \text{ert}[\text{call } P,\mathscr{D}](f) \sqsubseteq \text{ert}[\text{call } P,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \text{ert}[\text{call } P,\mathscr{D}](f)
\end{array}
$$

E.2.6 Preservation of infinity

The proof is done by induction on the structure of the command c. Note that c is *abort* free, that is, c contains no abort commands.

Skip

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{skip}\,,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \infty \end{aligned}$$

Assert

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{assert}\ e,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& [\![e:\ \mathsf{true}\,]\!]\cdot\infty = \infty \end{aligned}$$

Weaken

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{weaken}\,,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \infty \end{aligned}$$

Tick

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{tick}\,(q),\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \mathbf{q} + \infty = \infty \end{aligned}$$

Assignment

$$\begin{aligned} & \mathrm{ert}\,[x = e,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \infty[e/x] = \infty \end{aligned}$$

Sampling

$$\begin{aligned} & \mathrm{ert}\,[x = e\ \mathsf{bop}\ R,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(\infty)(\sigma[e\ \mathsf{bop}\ v/x])) \\ =\;& \mathbb{E}_{\mu_R}(\infty) = \infty \end{aligned}$$

If

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& [\![e:\ \mathsf{true}\,]\!]\cdot\ \mathrm{ert}\,[c_1,\mathscr{D}](\infty) + \\ & [\![e:\ \mathsf{false}\,]\!]\cdot\ \mathrm{ert}\,[c_2,\mathscr{D}](\infty) \\ & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ =\;& [\![e:\ \mathsf{true}\,]\!]\cdot\infty + [\![e:\ \mathsf{false}\,]\!]\cdot\infty \\ =\;& \infty \end{aligned}$$

Nondeterministic branching

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \max\{\,\mathrm{ert}\,[c_1,\mathscr{D}](\infty),\ \mathrm{ert}\,[c_2,\mathscr{D}](\infty)\} \\ & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ =\;& \max\{\infty,\infty\} \\ =\;& \infty \end{aligned}$$

Probabilistic branching

$$\begin{aligned} & \mathrm{ert}\,[c_1 \oplus_p c_2,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& p\cdot\ \mathrm{ert}\,[c_1,\mathscr{D}](\infty) + (1-p)\cdot\ \mathrm{ert}\,[c_2,\mathscr{D}](\infty) \\ & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ =\;& p\cdot\infty + (1-p)\cdot\infty = 1\cdot\infty \\ =\;& \infty \end{aligned}$$

Sequence

$$\begin{aligned} & \mathrm{ert}\,[c_1;c_2,\mathscr{D}](\infty) \\ & \dagger \text{ Table 2 } \dagger \\ =\;& \mathrm{ert}\,[c_1,\mathscr{D}](\,\mathrm{ert}\,[c_2,\mathscr{D}](\infty)) \\ & \dagger \text{ By I.H on } c_2 \dagger \\ =\;& \mathrm{ert}\,[c_1,\mathscr{D}](\infty) \\ & \dagger \text{ By I.H on } c_1 \dagger \\ =\;& \infty \end{aligned}$$

Loop Consider the characteristic function w.r.t the expectation ∞

$$F_\infty := [\![e:\ \mathsf{true}\,]\!]\cdot\ \mathrm{ert}\,[c,\mathscr{D}](X) + [\![e:\ \mathsf{false}\,]\!]\cdot\infty$$

By Theorem C.2, we have $\mathrm{ert}\,[\,\mathsf{while}\ e\ c,\mathscr{D}](\infty) = \sup_n F_\infty^n(\mathbf{0})$, thus we show $\sup_n F_\infty^n(\mathbf{0}) = \infty$. The proof is done by contradiction. Assume that $\sup_n F_\infty^n(\mathbf{0}) < \infty$. Hence, there exists $M < \infty$ such that $\forall n \in \mathbb{N}.\sigma \in \Sigma.F_\infty^n(\mathbf{0})(\sigma) \le M$. By definition of $F_\infty^n(\mathbf{0})$, it is

$$[\![e:\ \mathsf{true}\,]\!](\sigma)\cdot\ \mathrm{ert}\,[c,\mathscr{D}](F_\infty^{n-1}(\mathbf{0}))(\sigma) + [\![e:\ \mathsf{false}\,]\!](\sigma)\cdot\infty$$

If $\sigma \not\models e$ then $[\![e:\ \mathsf{true}\,]\!](\sigma) = 0$ and $[\![e:\ \mathsf{false}\,]\!](\sigma) = 1$. We get $F_\infty^n(\mathbf{0})(\sigma) = \infty$. Therefore, the assumption is contradictory, or $\mathrm{ert}\,[\,\mathsf{while}\ e\ c,\mathscr{D}](\infty) = \infty$.

Procedure call Because $\mathsf{call}_n^{\mathscr{D}} P$ (defined in Section C.4) is closed command for all $n \in \mathbb{N}$, by I.H we get

$$\mathrm{ert}\,[\,\mathsf{call}_n^{\mathscr{D}} P](\infty) = \infty$$

On the other hand, by Theorem C.5, we have

$$\begin{aligned} & \mathrm{ert}\,[\,\mathsf{call}\ P,\mathscr{D}](\infty) \\ =\;& \sup_n \mathrm{ert}\,[\,\mathsf{call}_n^{\mathscr{D}} P](\infty) \\ & \dagger \text{ Observation above } \dagger \\ =\;& \sup_n \infty \\ =\;& \infty \end{aligned}$$

F Simulation-based comparison

We measured the expected numbers of ticks for the collected 39 challenging examples with different looping and recursion patterns that use probabilistic branching and sampling assignments. Then we compared the results to our computed bounds. The following figures show the complete list of plots of these comparisons. In the plots, we draw a candlestick chart for each example as well as the measured mean value and the corresponding inferred bound of the number of ticks. The candlesticks represent the highest (h), lowest (l) measured values, 75%, and 25% of the interval ($h - l$).

Our experiments indicate that the computed bounds are close to the measured expected values needed for both linear and polynomial expected bound programs. However, there is no guarantee that Absynth infers asymptotically tight bounds and there is many classes of bounds that Absynth cannot derive. For example, *C4B_t15* has expected logarithmic expected cost, thus the best bound that Absynth can derive is a linear bound. Similarly, $|[0,n]|+|[0,m]|$ is the best bound that can be inferred for *condand* whose expected cost defined by function $2\cdot\min\{|[0,n]|,|[0,m]|\}$. Another source of imprecise constant factors in the bounds is rounding. Program *robot* has an imprecise constant factor because it uses a profound depth of nested probabilistic choice.

Some programs whose worst-case resource usage is only triggered by some particular inputs, called *worst-case inputs*. Thus, this makes the difference between the measured values and the inferred bounds big. For instance, the worst-case inputs of *C4B_t30* are values of x and y such that they are equal. While we measured the expected values with x varying from 1000 to 5000 and y is fixed to be 3000.

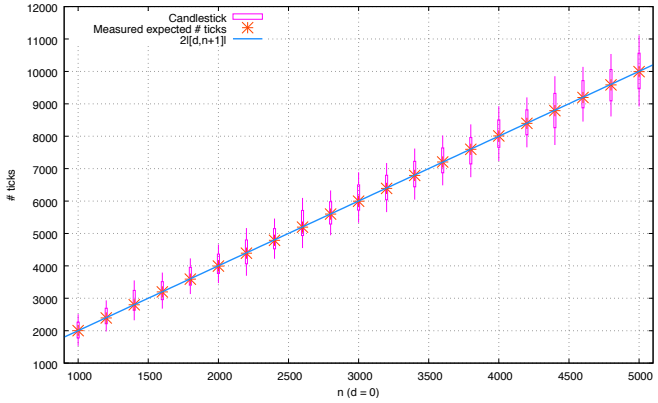


Figure 10. Example *2drwalk*.

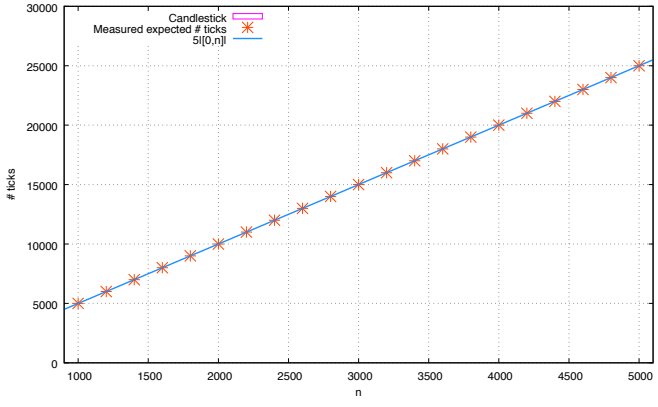


Figure 11. Example *bayesian*.

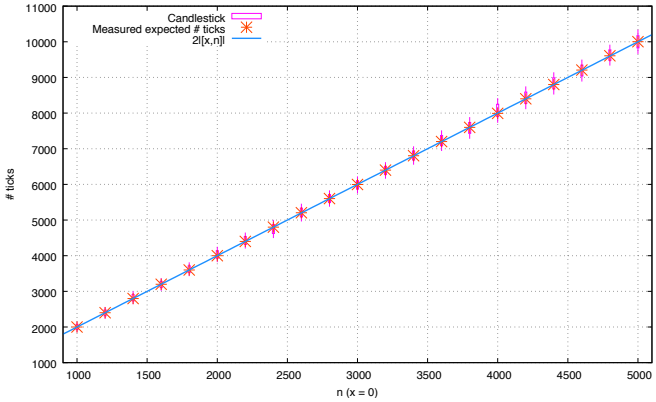


Figure 12. Example *ber*.

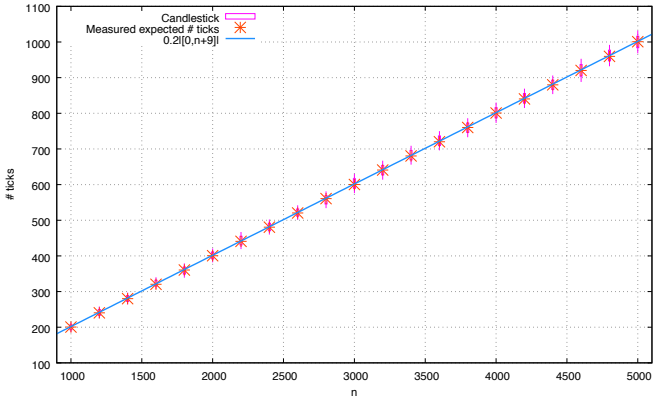


Figure 13. Example *bin*.

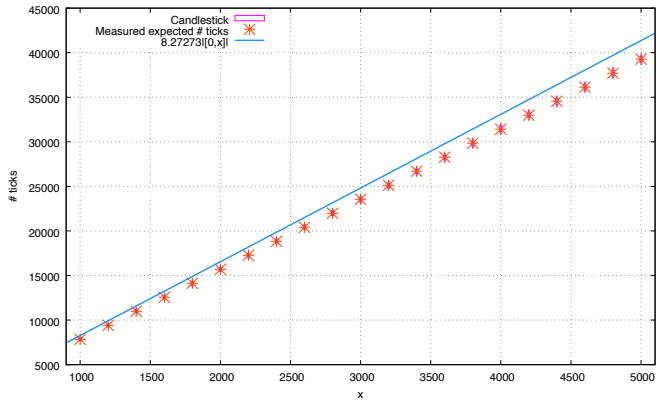


Figure 14. Example *C4B_t09*.

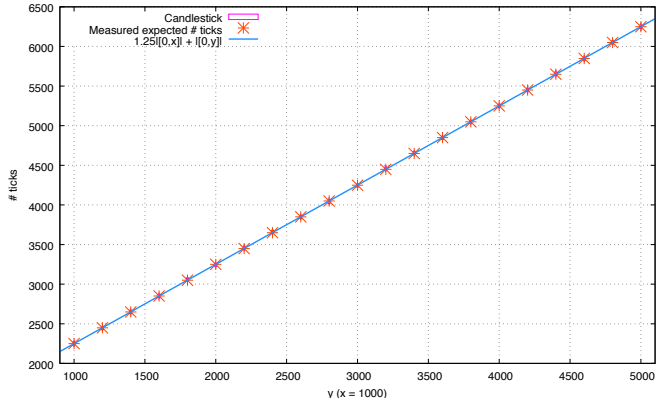


Figure 15. Example *C4B_t13*.

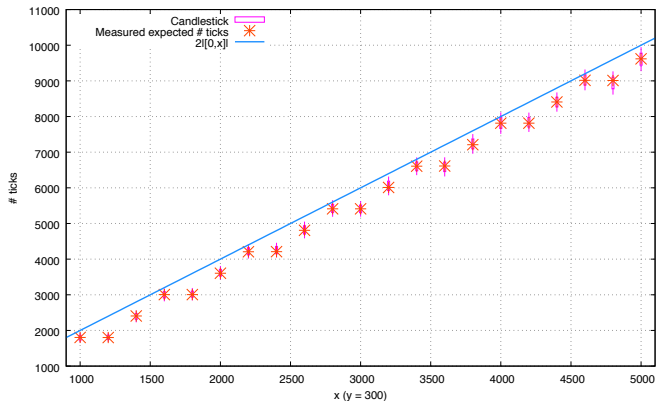


Figure 16. Example *C4B_t15*.

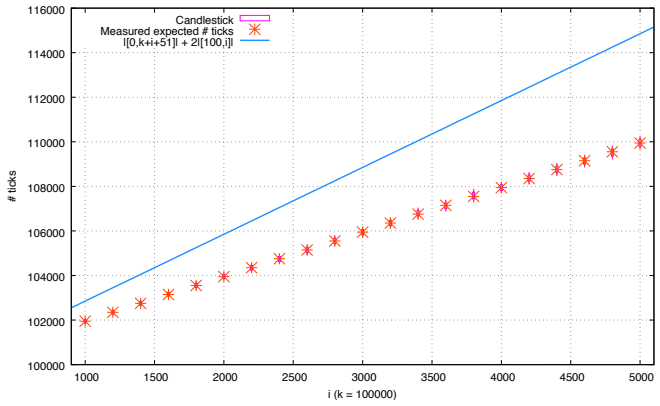


Figure 17. Example *C4B_t19*.

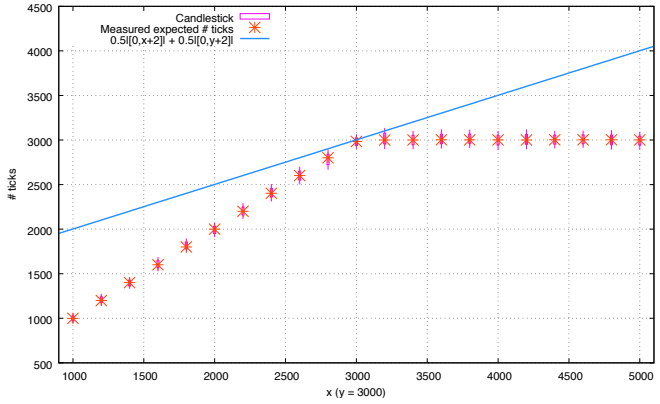


Figure 18. Example *C4B_t30*.

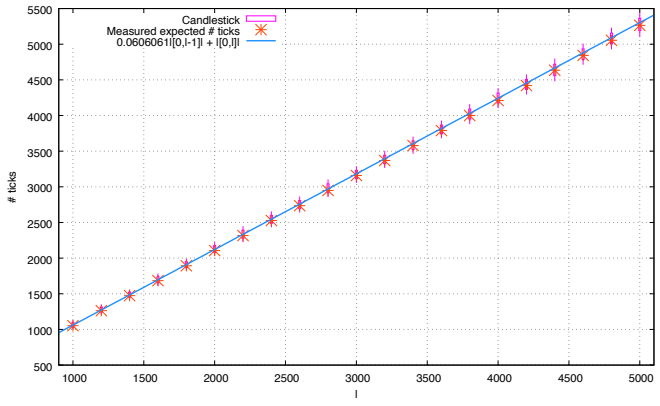


Figure 19. Example *C4B_t61*.

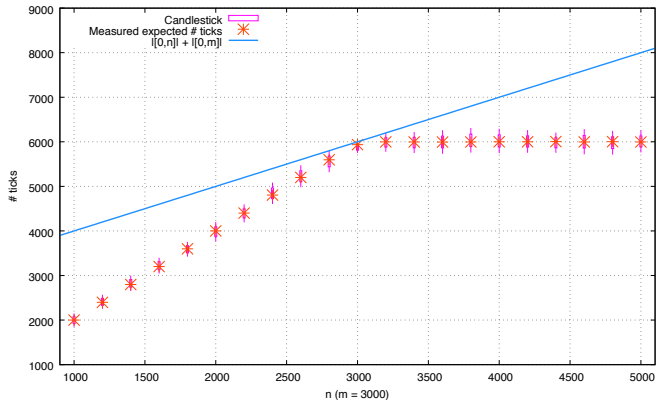


Figure 20. Example *condand*.

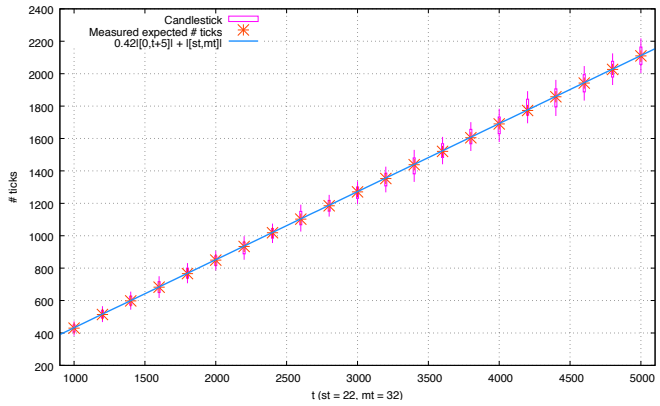


Figure 21. Example *cooling*.

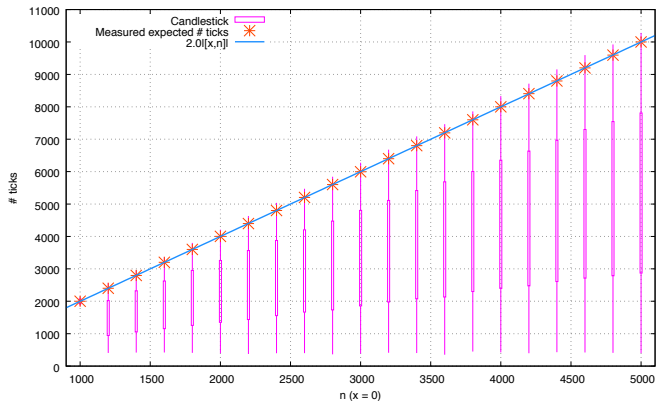


Figure 22. Example *fcall*.

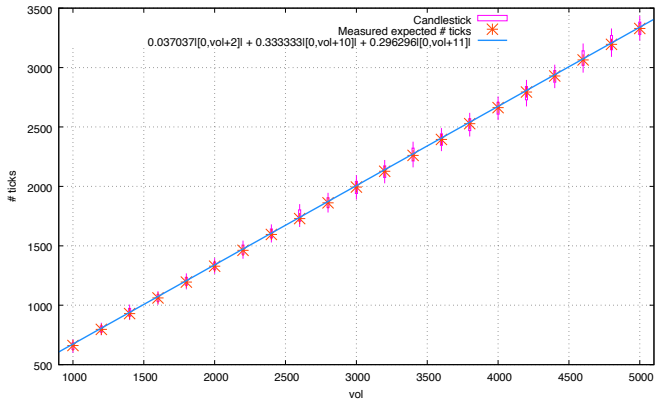


Figure 23. Example *filling*.

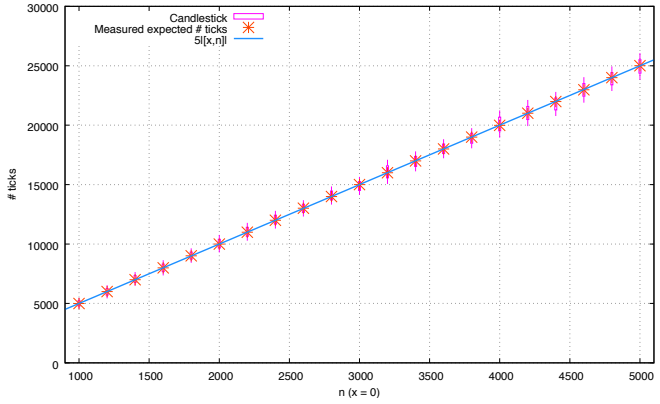


Figure 24. Example *hyper*.

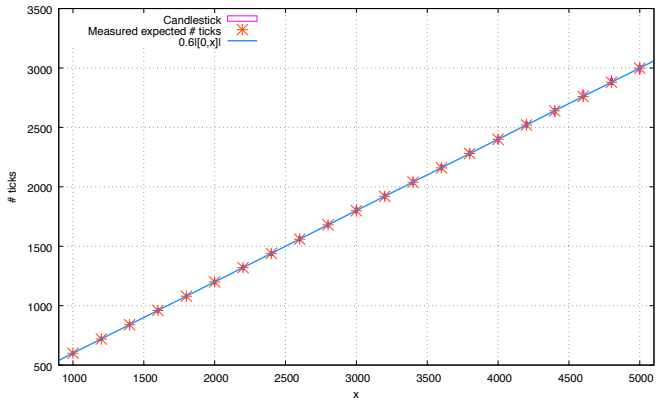


Figure 25. Example *linear01*.

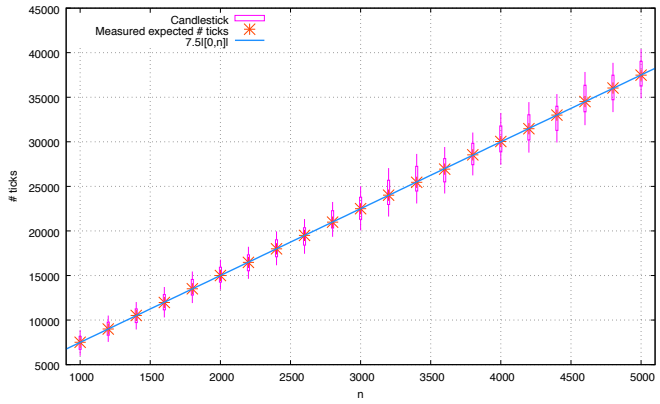


Figure 26. Example *miner*.

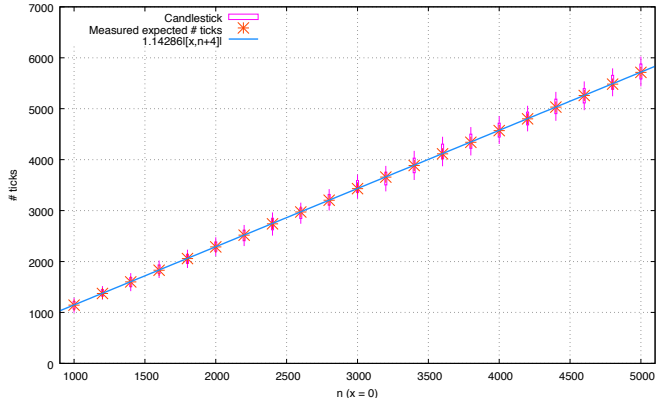


Figure 27. Example *prdwalk*.

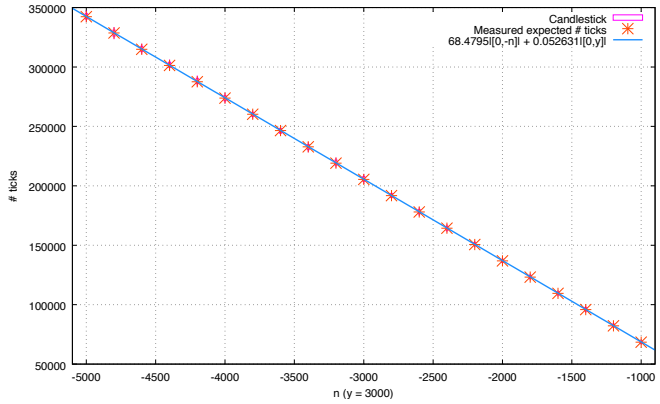


Figure 28. Example *prnes*.

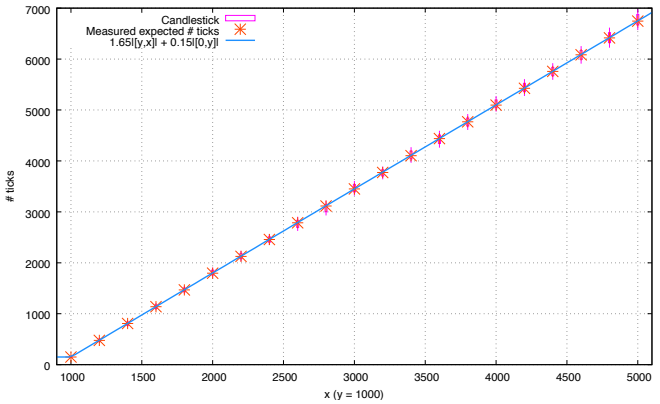


Figure 29. Example *prseq*.

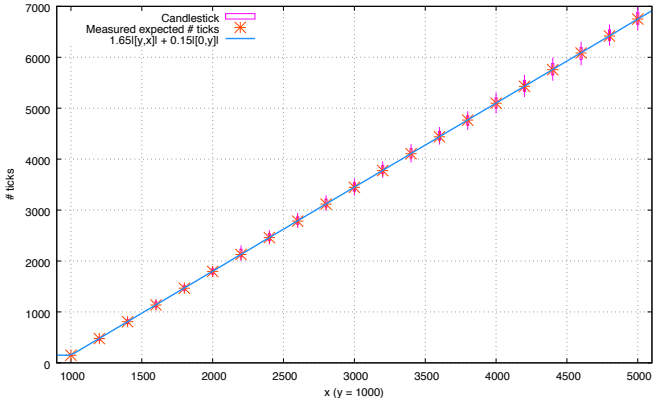


Figure 30. Example *prseq_bin*.

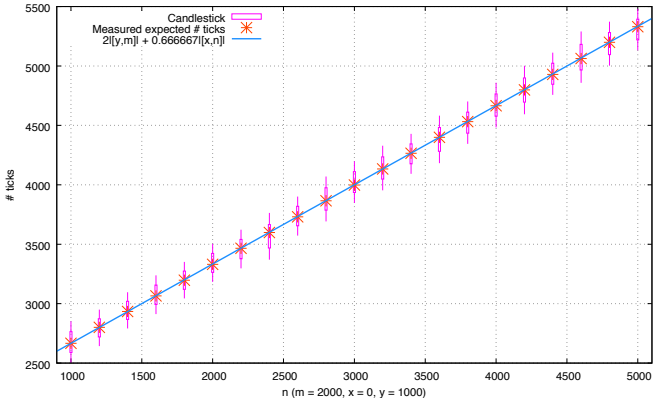


Figure 31. Example *prspeed*.

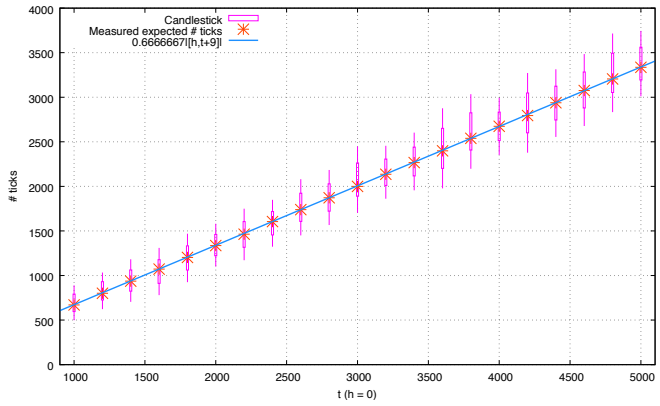


Figure 32. Example *race*.

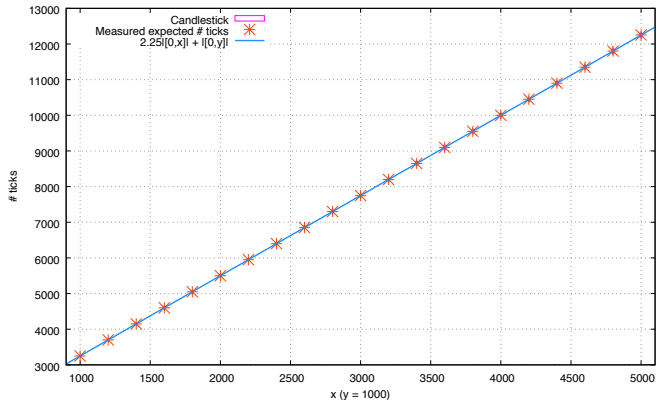


Figure 33. Example *rdseql*.

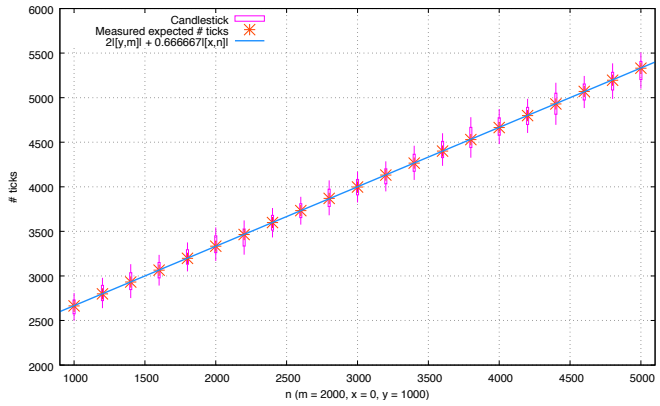


Figure 34. Example *rdspeed*.

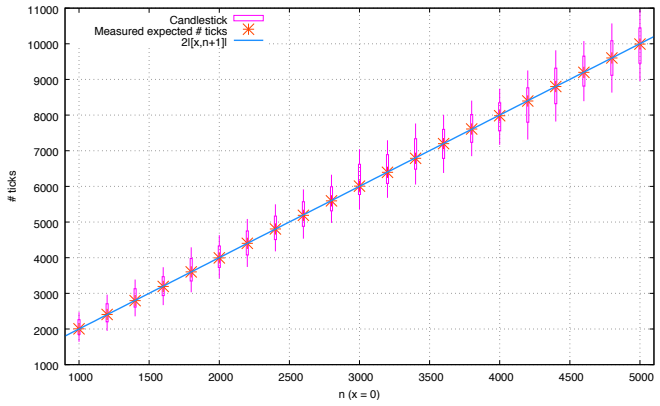


Figure 35. Example *rdwalk*.

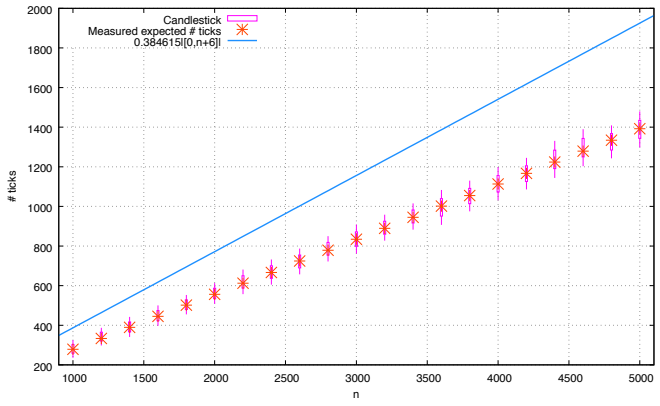


Figure 36. Example *robot*.

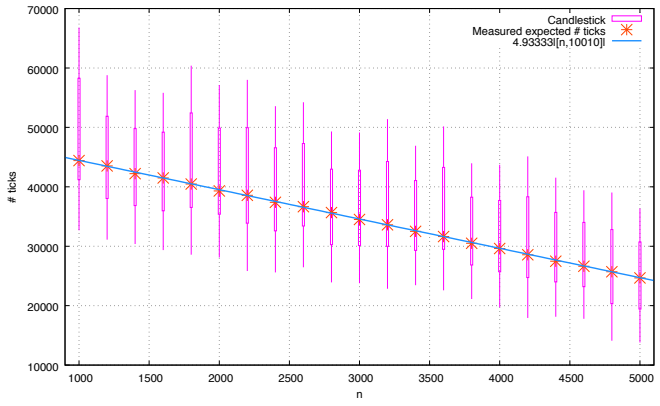


Figure 37. Example *roulette*.

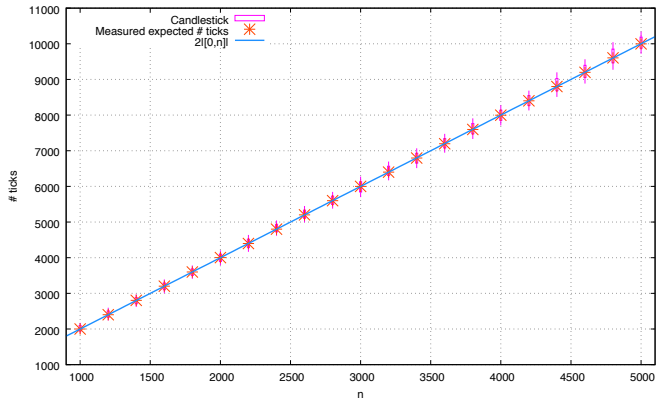


Figure 38. Example *sampling*.

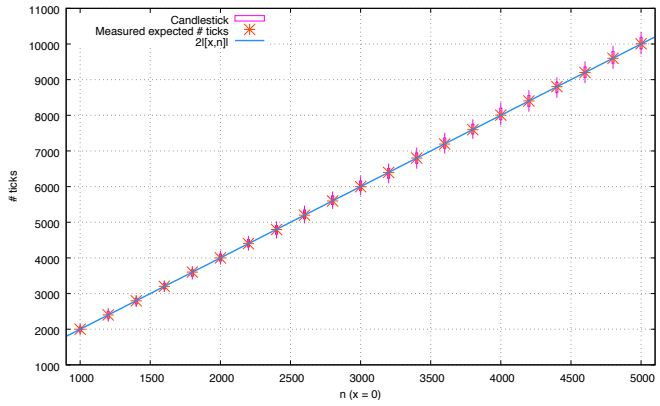


Figure 39. Example *sprdwalk*.

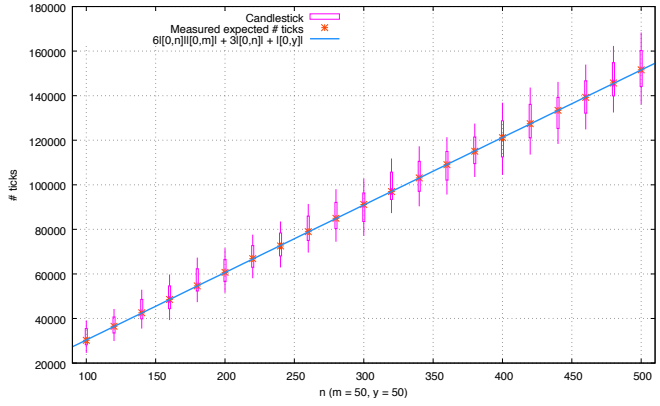


Figure 40. Example *complex*.

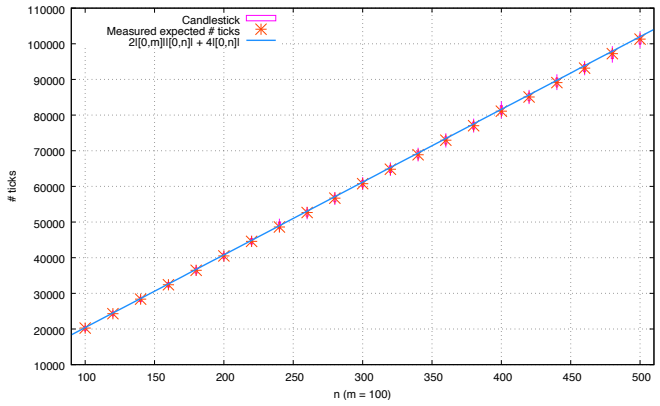


Figure 41. Example *multirace*.

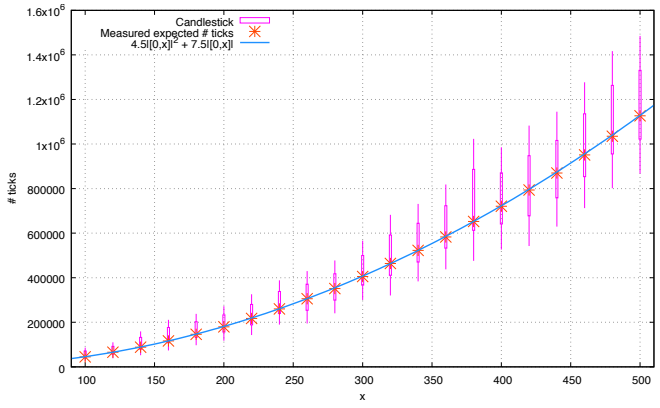


Figure 42. Example *pol04*.

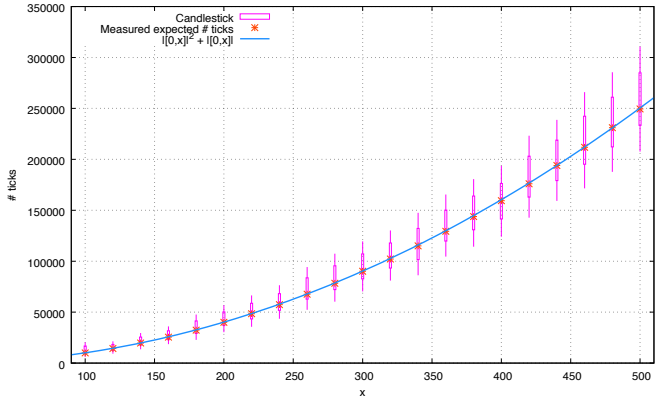


Figure 43. Example *pol05*.

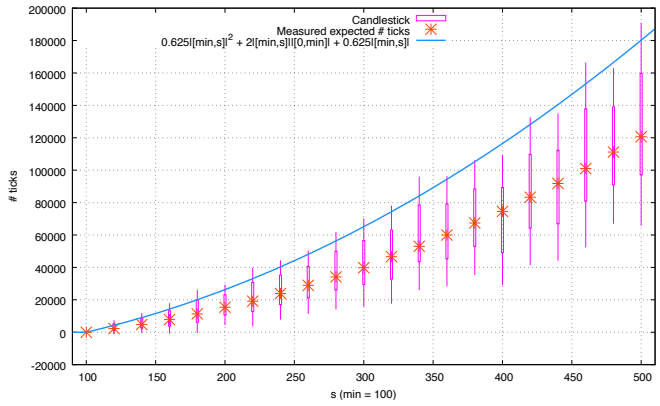


Figure 44. Example *pol06*.

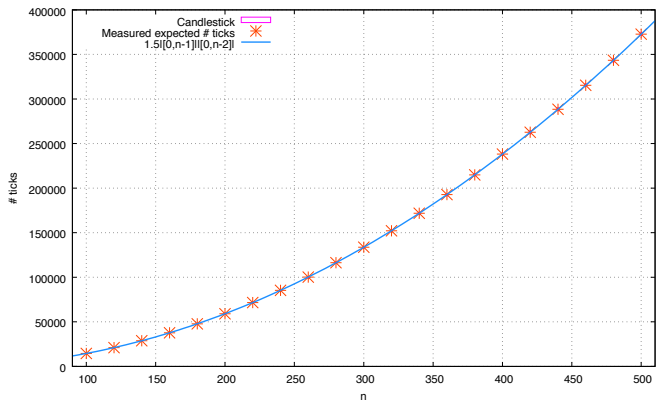


Figure 45. Example *pol07*.

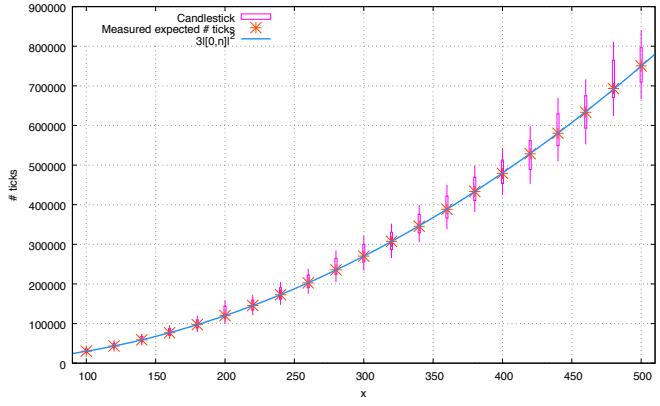


Figure 46. Example *rdbub*.

G More examples of derivation

In this section, we show more examples of derivation for challenging probabilistic programs. We also demonstrate how Absynth handles Boolean values to get the tightest

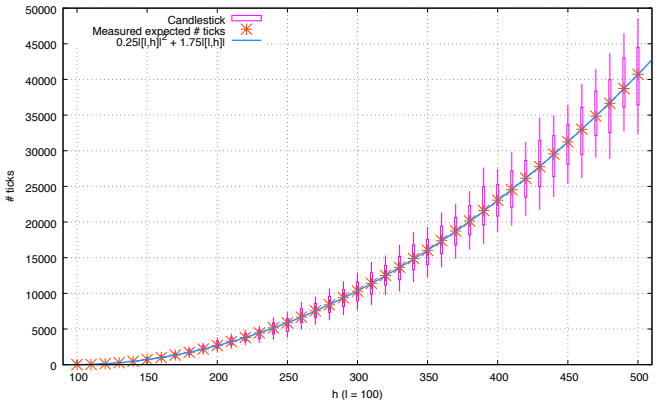


Figure 47. Example *recursive*.

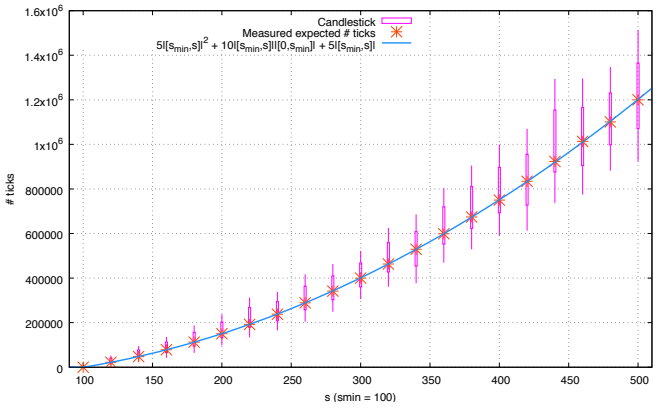


Figure 48. Example *trader*.

expected bounds. All the presented bound derivations can be derived automatically by our tool Absynth.

The derivation on the left side of Figure 49 infers the bound $\Phi = \frac{2T}{pK_1+(1-p)K_2}\cdot|[x, n+K_2-1]|$ on the number of ticks for a generalized version of *rdwalk*, in which both probabilistic branching and sampling commands are used. The value of x is added by a uniformly-distributed random variable from 0 to K_1 with probability p. With probability $(1-p)$, it is incremented by a uniformly-distributed random variable from 0 to K_2. The reasoning is similar to the ones of previous examples, in which we obtain the following post-potential after the samplings.

$$\Phi' = \frac{2T}{pK_1+(1-p)K_2}\cdot|[x, n+K_2-1]|+T$$

The corresponding pre-potentials w.r.t the samplings are given as follows.

$$\Phi_1 = (T-\frac{TK_1}{pK_1+(1-p)K_2}+\frac{2T}{pK_1+(1-p)K_2}\cdot|[x, n+K_2-1]|)$$
$$\Phi_2 = (T-\frac{TK_2}{pK_1+(1-p)K_2}+\frac{2T}{pK_1+(1-p)K_2}\cdot|[x, n+K_2-1]|)$$

Thus, the pre-potential before the probabilistic branching that also serves as a loop invariant is $\Phi = p\cdot\Phi_1+(1-p)\cdot\Phi_2$.

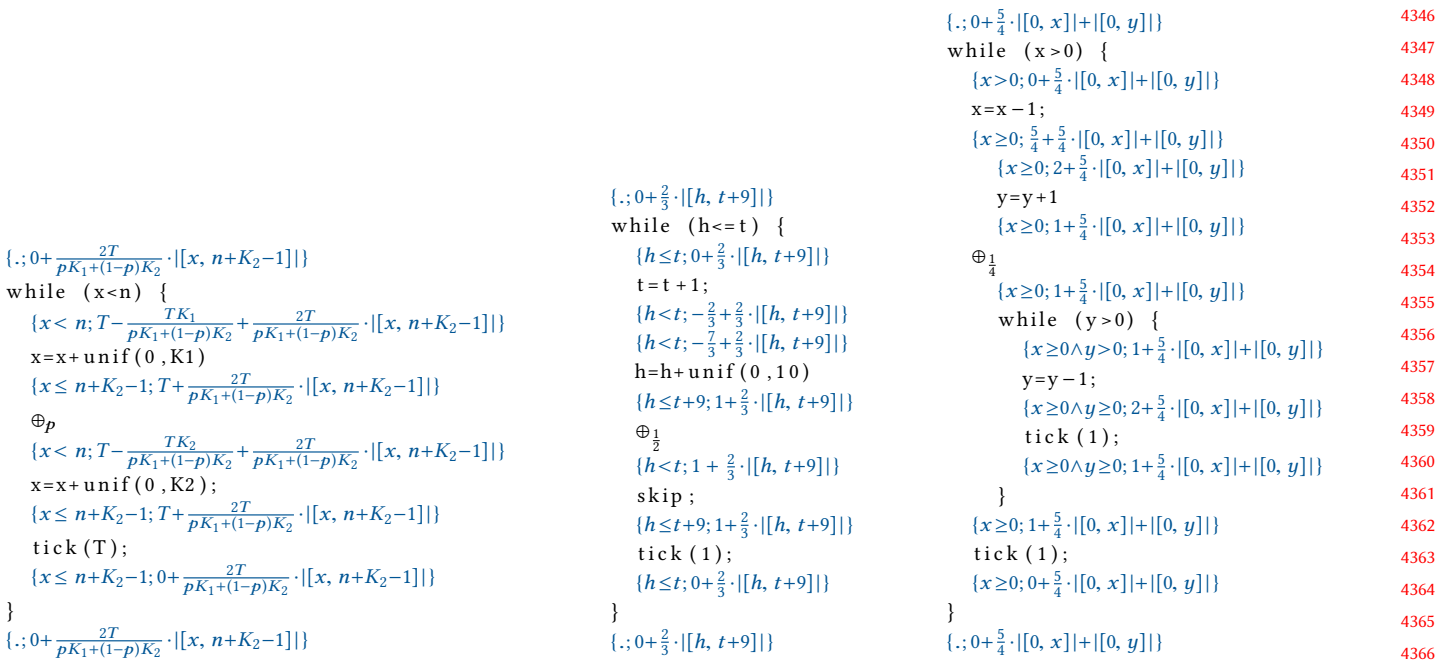


Figure 49. More derivations of bounds on the number of ticks for loopy probabilistic programs using both probabilistic branching and random sampling assignment. The parameters $K_2 \geq K_1 > 0$, $T > 0$, and $p \geq 0$ denote concrete constants.

Example *race* introduced in Section 2 models a race between a hare (variable h) and a tortoise (variable t). With the same reasoning for *prdwalk*, we can infer the bound $\frac{2}{3}\cdot|[h, t+9]|$ on the expected value of the elapsed time.

Example *C4B_t13* adapted from [19] shows how expected potential method can be used to find linear expected bound for nested loop. The outer loop iterates $|[0, x]|$ times. We increase y by 1 with probability $\frac{1}{4}$, otherwise the inner loop is executed until y is 0 with probability $\frac{3}{4}$. Absynth computes a tight bound by discovering that only one run of the inner loop depends on the initial value of y, that is $|[0, y]|$, for the others runs, the expected value of total number of iterations is $\frac{1}{4}\cdot|[0, x]|$.

In Figure 50, example *miner* simulates a miner who is trapped in a mine. The miner is sent to the mine for n times independently, for each time, with probability $\frac{1}{2}$ he is trapped and with the same probability he is safe. When he is trapped, there are 3 doors in the mine for using. The first door leads to a tunnel that will take him to safety after 3 hours (representing by 3 ticks). The second door leads to a tunnel that returns him to the mine after 5 hours. And the third door leads to a tunnel that returns him to the mine after 7 hours. At all times, the miner is equally likely to choose any one of the doors, meaning that he chooses any door with equivalent probability $\frac{1}{3}$.

Let's consider the manual way to compute expected time the miner reaches safety for n independent times. Let X_i be the random variable representing the miner escape time in the i^{th} time and let $X = X_1 + \cdots + X_n$, we are asking for $\mathbb{E}(X)$. We first need to use the *law of total expectation* to calculate $\mathbb{E}(X_i)$. Let A be the event that the miner is trapped then $\mathbb{E}(X_i) = \mathbb{E}(X_i|A)\cdot\mathbb{P}(A) + 0\cdot\mathbb{P}(\neg A) = \frac{1}{2}\cdot\mathbb{E}(X_i|A)$. Now use again the law of total expectation to calculate $\mathbb{E}(X_i|A)$, let B be the event that the miner reaches safety in the first try, then we have $\mathbb{E}(X_i|A) = \mathbb{E}((X_i|A)|B)\cdot\mathbb{P}(B) + \mathbb{E}((X_i|A)|\neg B)\cdot\mathbb{P}(\neg B) = 3\cdot\frac{1}{3} + (5\cdot\frac{1}{2} + 7\cdot\frac{1}{2} + \mathbb{E}(X_i|A))\cdot\frac{2}{3} = 5 + \frac{2}{3}\cdot\mathbb{E}(X_i|A)$, thus $\mathbb{E}(X_i|A) = 15$. Finally, by linearity of expectation, we get $\mathbb{E}(X) = \sum_{i=1}^{n}\mathbb{E}(X_i) = \frac{15}{2}\cdot n$ if $n > 0$, 0 otherwise.

As we have shown in the above reasoning process, the manual method involves many heuristic theorems and techniques in probability theory and algebra. As a result, it is very hard or impossible to automate. This example demonstrates how Absynth handles Boolean values (the variable flag) to get the tightest expected bound $\frac{15}{2}\cdot|[0, n]|$. It assigns the potential $15\cdot|[0, \text{flag}]|$ to the variable flag for paying the cost of the inner loop, in which $\frac{15}{2}\cdot|[0, n]| - \frac{15}{2} + 15\cdot|[0, \text{flag}]|$ is the loop invariant.

Example *rdbub* models a probabilistic bubble sort algorithm. For each iteration of the inner loop, a pair of adjacent elements in the input array can be swapped. However, the

```
{q:=15/2·|[0, n]|−15/2}
{.; 15/2·|[0, n]|}
while (n>0) {
  {n>0; 15/2+q}
  flag = 1;
    {flag=1; 15·|[0, flag]|+q}
    while (flag > 0) {
      {flag=1; 3+q}
        flag=0;
        {flag=0; 3+15·|[0, flag]|+q}
        tick(3)
        {flag=0; 15·|[0, flag]|+q}
      ⊕1/3
      {flag=1; −3/2+45/2+q}
          {flag=1; −5/2+45/2+q}
          flag=1;
          {flag=1; 5+15·|[0, flag]|+q}
          tick(5)
            {flag=1; 15·|[0, flag]|+q}
        ⊕1/2
          {flag=1; −1/2+45/2+q}
          flag=1;
          {flag=1; 7+15·|[0, flag]|+q}
          tick(7);
          {flag=1; 15·|[0, flag]|+q}
    }
    {flag ≤0; 15·|[0, flag]|+q}
    {n>0; q}
  ⊕1/2
    {n>0; 15/2·|[0, n]|−15/2}
    skip;
  {n>0; q}
  n=n−1;
  {n≥0; 15/2·|[0, n]|}
}
```

```
{inv:=6·(|[0,n]|2)+3·|[0, m]|}
{.; 3·|[0, n]|²}
while (n>0) {
  {n>0; 3·|[0, n]|²}
  {n>0; 6·(|[0,n]|2)}
  n=n−unif(0,1);
  {n≥0; 6·(|[0,n]|2)+3·|[0, n]|}
  m=n
  {n=m; inv}
  while (m>0) {
    {m=n∧m>0; −2+inv}
    m=m−1
    {m=n∧m≥0; 1+inv}
    ⊕1/3
    {m=n∧m>0; 1+inv}
    skip;
    {m=n∧m≥0; 1+inv}
    tick(1);
    {m=n∧m≥0; 0+inv}
  }
  {m≤0; inv}
}
```

miner	*rdbub*
$\frac{15}{2}\cdot\lvert[0,n]\rvert$	$3\cdot\lvert[0,n]\rvert^2$

Figure 50. More representative examples where probabilistic branching is used with compound statements and polynomial bound.

algorithm performs the swapping only with probability $\frac{1}{3}$ and skips the swap with probability $\frac{2}{3}$. The derivation proves the polynomial bound $3\cdot|[0,n]|^2$ on the expected number of iterations. This example shows the linear potential $3\cdot|[0,m]|$ is spilled from the a quadratic potential $6\cdot\binom{|[0,n]|}{2}$ after the sampling in the outer loop. This linear potential is used to pay for the ticks in the inner loop. The derivation of the bound for the inner loop is similar to the bound derivations for the previous program. However, the annotation carries an extra quadratic part which is invariant in the inner loop and just passed along in the derivation. Note that the derivation of the potential $3\cdot|[0,n]|^2$ from the potential $6\cdot\binom{|[0,n]|}{2}$ reflects the use of the rule Q:Weaken in the derivation system.