Bounded Expectations: Resource Analysis for Probabilistic Programs

VAN CHAN NGO, Carnegie Mellon University
QUENTIN CARBONNEAUX, Yale University
JAN HOFFMANN, Carnegie Mellon University

Following the increasing relevance of probabilistic programming, there is a renewed interest in addressing the challenges that probabilistic code bears for static reasoning. For example, there are successful techniques for automatic worst-case resource analysis but these techniques are not applicable to many probabilistic programs, which, for instance, only terminate almost surely. 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, which reduces bound inference to off-the-shelf LP solving. This design also allows to extend automatically-derived bounds with existing program logics if the automation fails. The effectiveness of the technique is demonstrated with a prototype implementation that is used to automatically analyze probabilistic programs and randomized algorithms that previously have been analyzed manually in the literature. Building on existing work, the soundness of the analysis is proved with respect to an operational semantics that is based on Markov decision processes.

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: porbabilistic programs, resource bound analysis, static analysis

ACM Reference format:
Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2017. Bounded Expectations: Resource Analysis for Probabilistic Programs. 1, 1, Article 1 (January 2017), 56 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Probabilistic programming [62, 75] is an increasingly popular method for implementing and analyzing Bayesian Networks and Markov Chains [40], randomized algorithms [8], cryptographic constructions [11], and privacy mechanisms [10]. 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, 58, 62, 67, 74], automatic probabilistic invariant generation [22, 25], abstract interpretation for probabilistic programs [29, 68, 69], symbolic inference [38], and probabilistic model checking [59].

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

2017. XXXX-XXXX/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

developed numerous tools that can be used to automatically or semi-automatically derive non-trivial symbolic resource bounds for imperative [16, 44, 61, 78] and functional programs [5, 32, 51, 83]. For example, when presented with a program like quick sort, existing techniques automatically derive a bound like $3 + 19n + 14n^2$ on the evaluation time in less than a second [48].

Existing techniques for resource bound analysis can be readily 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 upper bound on the resource consumption. Consider for example the function *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. We also say, it *almost surely terminates* [34].

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

```
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 numShares;
    numShares = unif(0,10);
    while (numShares > 0) {
        numShares = numShares - 1;
        cost = cost + s;
    }
}
```

(b)

Fig. 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. That cost of the program is modeled by the global variable *cost*. If $s \geq s_{min}$ then the expected cost is $5{\cdot}(s{-}s_{min}) + 10{\cdot}(s{-}s_{min}){\cdot}s_{min} + 5{\cdot}(s{-}s_{min})^2$.

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 [58, 74]. Using these derivations, the computed bounds can seamlessly be extended with manual proofs if the automation fails. It would certainly be preferable to automatically derive the exact distribution of the resource consumption in some cases but bounds on the expected resource consumption are a good compromise between precision and effectiveness.

For example, consider again the probabilistic programs in Figure 1. For the random walk implemented in the function *trade(s_{min},s)*, our technique automatically derives the bound $2{\cdot}\max(0, s{-}s_{min})$ on the expected number of loop iterations. For the total spending of the trader, 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. However, there is in general no guarantee that the bounds we derive are tight.

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. It is also one the few techniques that can automatically derive polynomial properties. As demonstrated by the examples in Figure 1, the analysis works for imperative programs that are extended with a probabilistic branching construct and the possibility of sampling from a discrete distribution. Different resource metrics can be defined either by using a variable or by the use of *tick* commands. The analysis is compositional, automatically tracks size changes, and derives whole program bounds. Derived time bounds also imply *positive termination* [34]. Compositionality, is particularly tricky for probabilistic programs since the composition of two positively (almost surely) terminating programs is not necessarily positively (almost surely) terminating. 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 [70]) 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 [20, 58, 67, 74] that is based on derivation rules for weakest preconditions (WPs) with an automatic resource analysis method that is known as automatic amortized resource analysis (AARA) [17, 19, 48, 51]. As a result, our method combines the best properties of both worlds. On the one hand, we have the conceptual simplicity of quantitative WP reasoning and a soundness proof of the analysis with respect to an operational semantics based on Markov Decision Processes (MDP). 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 with a Python-like syntax that features procedures, recursion, and loops. Absynth also has a C interface based on LLVM and we use C notation for the examples in this paper. We have performed experiments with 43 examples from the literature on probabilistic programs and randomized algorithms. We can often derive a precise bound on the expected resource cost and for smaller programs the inference takes usually only seconds.

Limitations. Deriving symbolic resource bounds is an undecidable problem and our technique only works if loops and recursive functions have a relatively simple control flow. Furthermore, our implementation currently only derives polynomial bounds. We only support 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.

Contributions. 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 [74].
- We show the effectiveness of the technique with a prototype implementation and by successfully analyzing 43 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 linear constraint solving, compatibility with manual techniques for deriving bounds, and a soundness proof with respect to an operational MDP cost semantics.

2 PROBABILISTIC PROGRAMS

In this section we present the syntax of our imperative probabilistic language and give its semantics in terms of pushdown Markov Decision Process (MDP) with rewards.

2.1 Essential notions and concepts from probability theory

We first recall some essential concepts and notations from probability theory that are used in this paper. The interested readers can find more detailed descriptions in reference textbooks [3].

Probability space. Consider a random experiment. The set Ω of all possible outcomes of the experiment is called the *sample space*. Every probabilistic statement is associated with an underlying *probability 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 $\mu_X(B) = \mathbb{P}(X^{-1}(B))$ is a probability measure for $\mathscr{B}$ and $(\mathbb{R}, \mathscr{B}, \mu_X)$ is a probability space. The measure μ_X is called the *probability distribution* of X. If μ_X measures on a countable set of reals, or the range of X is countable, then X is called a *discrete random variable*. If μ_X gives zero measure to every singleton set, then X is called a *continuous random variable*. The distribution μ_X is often characterized by the *cumulative distribution function* defined by $F_X(x) = \mathbb{P}(X \leq x) = \mu_X((-\infty, x])$.

Expectation. The *expected value* of a random variable X from a probability space $(\Omega, \mathscr{F}, \mathbb{P})$, denoted $\mathbb{E}(X)$, is defined as $\mathbb{E}(X) := \int_\Omega X d\mathbb{P}$ where $\int$ is the Lebesgue integral of X [3]. $\mathbb{E}(X)$ can be defined through the distribution μ_X as $\mathbb{E}(X) = \int_{\mathbb{R}} x d\mu_X$. To clearly mention the expectation of X w.r.t its distribution μ_X, we write $\mathbb{E}_{\mu_X}(X)$. For a discrete random variable, it is defined as a 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. One of the most important properties of the expectation is *linearity*: let X, X' be two random variables and λ, μ be two real numbers, 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 with C syntax. It is structured into expressions and commands. We only consider expressions that are free of side effects. Its non-probabilistic part is general enough to demonstrate our approach and it contains the standard commands for probabilistic branching and discrete sampling.

The abstract syntax of the language is given by the grammar in Figure 2. For use in the MDP-based semantics, we assume that every command c is labeled with a unique *location* ℓ. The command $\mathtt{id} = e \text{ bop } R$, where R is a (discrete) random variable whose probability distribution is μ_R (written as $R \sim \mu_R$), is a *random sampling* assignment. The command first samples independently from the probability distribution μ_R to obtain a sample value of R then evaluates the expression in which R is replaced by the sample. Finally, the evaluated value is assigned to the variable $\mathtt{id}$. The command $c_1 \oplus_p c_2$ is a *probabilistic branching*: With probability p command c_1 is executed, whereas with probability $(1 - p)$ command c_2 is executed.

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

Fig. 2. The abstract syntax of a simple probabilistic language.

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 P. In this article, we assume that the procedure P only manipulates 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. It helps to express assumptions on the inputs of the program. The primitive tick(q), where $q \in \mathbb{Q}_0^+$ is used to model resource consumption of the program 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.

Note that cost models can be probabilistic, that is, the resource consumption of the language constructs can be expressed as random variables. There are multiple ways to achieve this. First, if the cost is represented by a variable, then a sampled value can directly be added to it. Second, users can use probabilistic branching commands to express a probabilistic cost model. For example, one can use the following code segment to express a probabilistic cost model.

```
tick(1) ⊕1/3 { tick(2) ⊕1/4 tick(3) }
```

In this model, 1 resource unit is consumed with probability $\frac{1}{3}$, 2 resource units with probability $\frac{1}{6}$, and 3 with probability $\frac{1}{2}$. Thus, the expected cost is $\frac{1}{3}\cdot 1 + \frac{1}{6}\cdot 2 + \frac{1}{2}\cdot 3 = \frac{13}{6}$.

In the conditions we interpret 0 as false and all other values as true. Similarly, relation operators yield 0 for false and 1 for true. Note that our language only supports variables as the left-hand side of an assignment. Thus we do not consider operations that update the heap or the stack through a pointer.

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 3 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 $h > t$ of the loop). After each unit of time (expressed using the tick command, line 7), the tortoise goes one step forward (line 3). With probability $\frac{1}{2}$ (represented by a *probabilistic branching* command, line 5) the hare remains at its position (line 6) and with the same probability it advances a random number of steps between 0 and 10 (line 4) following the uniform distribution.

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

race

Fig. 3. A complete example of probabilistic program.

$$\text{(S:Term)}\ \frac{}{(\downarrow,\sigma) \xrightarrow{\tau,\epsilon,\epsilon,1} (\,\text{Term}\,,\sigma)} \qquad \text{(S:Skip)}\ \frac{\text{cmd}\,(\ell) = \text{skip} \qquad \text{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell',\sigma)} \qquad \text{(S:Abort)}\ \frac{\text{cmd}\,(\ell) = \text{abort}}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,1} (\ell,\sigma)}$$

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

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

$$\text{(S:Sample)}\ \frac{\text{cmd}\,(\ell) = \text{id} = e\ \text{bop}\ R \qquad \text{fcmd}\,(\ell) = \ell' \qquad [\![\mu_R : v]\!] = p > 0 \qquad \sigma' = \sigma[[\![e]\!]_\sigma\ \text{bop}\ v/\ \text{id}\,]}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma,p} (\ell',\sigma')} \qquad \text{(S:NonDetL)}\ \frac{\text{cmd}\,(\ell) = \text{if} \star c_1\ \text{else}\ c_2 \qquad \text{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\text{Th},\gamma,\gamma,1} (\ell',\sigma)}$$

$$\text{(S:NonDetR)}\ \frac{\text{cmd}\,(\ell) = \text{if} \star c_1\ \text{else}\ c_2 \qquad \text{scmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\text{El},\gamma,\gamma,1} (\ell',\sigma)} \qquad \text{(S:Call)}\ \frac{\text{cmd}\,(\ell) = \text{call}\ P \qquad \text{fcmd}\,(\ell) = \ell'}{(\ell,\sigma) \xrightarrow{\tau,\gamma,\gamma\cdot\ell',1} (\,\text{init}\,(\mathscr{D}),\sigma)}$$

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

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

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

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

Fig. 4. Rules of the probabilistic pushdown transition relation.

2.3 Operational cost semantics

Following existing work [74], 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 [7, 33]. Note that the main results of this article can be understood without working through this section.

A program state $\sigma : \text{VID} \to \mathbb{Z}$ 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 $\mathsf{init} : \mathsf{PID} \to L$ maps procedure names to the initial program location of their bodies, 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 4. 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, 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 procedure body. 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 4,
- $\mathscr{E} := L \cup \{\epsilon\}$
- 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 $\mathfrak{M}_\sigma^f[\![(c, \mathscr{D})]\!]$ eventually reaches the set of terminated states $(\mathsf{Term}, \_)$ from the starting state $s_0 = (\ell_0, \sigma)$. Formally,

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

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 $\mathfrak{M}_{\mathfrak{S},\sigma}^f[\![c, \mathscr{D}]\!]$, from $\mathfrak{M}_\sigma^f[\![c, \mathscr{D}]\!]$. $\Pi(s_0, \mathsf{Term})$ denotes the set of all finite paths from s_0 to some state (Term, σ') in $\mathfrak{M}_{\mathfrak{S},\sigma}^f[\![c, \mathscr{D}]\!]$, $\mathbb{P}^{\mathfrak{M}_\sigma^f[\![c,\mathscr{D}]\!]}(\widehat{\pi})$ is the probability of the finite path $\widehat{\pi}$. And $\mathfrak{Re}(\widehat{\pi})$ is the *cumulative reward* collected along $\widehat{\pi}$.

3 EXPECTED RESOURCE BOUND ANALYSIS

In this section, we show informally how automatic amortized resource analysis (AARA) [18, 48, 51] 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 of probabilistic code. 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 example. To build more intuition and showcase the flexibility of the technique, we then use the expected potential method to analyze several more challenging examples.

3.1 Manual analysis of a simple random walk

We demonstrate the classic approach to finding the expected runtime of a probabilistic program using recurrence relations. Consider the following program.

```
while (x > 0) {
    x = x - 1 ⊕3/4 x = x + 1;                                    (rdwalk1)
    tick(1);
}
```

Intuitively, it is clear that this program should terminate with high probability: most of the time (with probability $\frac{3}{4}$), the left assignment of the probabilistic branching is executed and therefore the variable x should get closer to 0 over time.

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. In this case, by expected runtime we mean the expected number of $\mathsf{tick}(1)$ statements executed. Then, for all $n \geq 1$, $T(n)$ satisfies the following equation

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

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

$$3D(n) = 4 + D(n+1). \tag{1}$$

One systematic way to find solutions for this equation is to use the generating function $G(z) = \sum_{n\geq 1} D(n)z^n$. With the equation (1), and writing D for $D(1)$, we get that the function $G(z)$ satisfies

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

And thus, using algebra and well-known generating functions

$$G(z) = \frac{4z^2}{(1-z)(3z-1)} - \frac{Dz}{3z-1} = \frac{6z^2 - Dz}{3z-1} + \frac{2z^2}{1-z} = \sum_{n\geq 1}(2 - D\cdot 3^{n-1} + 2\cdot 3^{n-1})z^n. \quad (2)$$

To finish the reasoning, we have to find the constant $D = T(1) - T(0)$ using a boundary condition. It is clear that $T(0) = 0$, because the loop is never entered when the program is started with $x = 0$. However, it is less clear how to find $T(1)$. The solution lies in the observation that any terminating execution of the loop started with $x = n$ has to first reach the state $x = n - 1$. 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, we have the new equation $T(n) = n \cdot T(1) = n \cdot D$ and consequently $D(n) = T(n) - T(n-1) = D$ for $n \geq 1$. But by (2), we also have $D(n) = 2 - D \cdot 3^{n-1} + 2 \cdot 3^{n-1}$, this entails that D is a solution of the equation

$$D = 2 - D \cdot 3^{n-1} + 2 \cdot 3^{n-1}.$$

Therefore we have $D = 2$ and thus the expected runtime of the loop started with $x = n \geq 0$ is $T(n) = 2 \cdot n$.

Limitations of the recurrence-relations method. Several limitations of the classic method make it hard to automate. First, inferring the recurrence relations is not always as simple as it was on our example. For example, more complex iteration patterns complicate this process. Additionally, there is currently no formal setting in which the correspondance between the program and the recurrence relation can be established.

Second, the method for solving the recurrence relation is fragile. In our reasoning, we used the hypothesis that the expected time to go from n to $n - 1$ is the same as the expected time to go from 1 to 0. Should the decrement be $x = x - 2$ instead of $x = x - 1$, this property would not be true anymore and the boundary condition would have to be found differently. Additionally, when bigger constants are used in the program, recurrence relations become of higher degrees and more difficult to solve. Similarly, the use of multiple variables complicates the solving process.

Finally, the classic method is not compositional. When programs become larger, the classic method does not provide a principled way to reason independently on smaller components.

3.2 The potential method for deterministic programs

It has been shown in the past decade that the potential method of automatic resource analysis provides an interesting alternative to classic resource analysis with recurrence relations that is particularly amenable to automation.

Assume that a program c executes with initial state σ and consumes n resource units as defined by the tick commands. To express this situation, we write $(c, \sigma) \Downarrow_n \sigma'$ where σ' is the program state after the execution. 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's execution. We thus use two potential functions Φ and Φ' that specify the available potential before and after the execution, respectively. The functions must satisfy the constraint $\Phi(\sigma) \geq n + \Phi'(\sigma')$ for all states σ and σ' such that $(c, \sigma) \Downarrow_n \sigma'$. Intuitively, $\Phi(\sigma)$ must be sufficient to pay for the

resource cost of the computation and for the potential $\Phi'(\sigma')$ on the resulting state σ'. Thus, if $(\sigma, c_1) \Downarrow_n \sigma'$ and $(\sigma', c_2) \Downarrow_m \sigma''$, we have $\Phi(\sigma) \geq n + \Phi'(\sigma')$ and $\Phi'(\sigma') \geq m + \Phi''(\sigma'')$ and therefore $\Phi(\sigma) \geq (n+m) + \Phi''(\sigma'')$. Note that the initial potential function Φ provides an upper bound on the resource consumption of the whole program. What we have observed is that, if we define $\{\Phi\}c\{\Phi'\}$ to mean

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

then the following familiar looking inference rule is valid.

$$\frac{\{\Phi\}c_1\{\Phi'\} \qquad \{\Phi'\}c_2\{\Phi''\}}{\{\Phi\}c_1; c_2\{\Phi''\}}$$

This rule already shows a departure from classical techniques that are based on recurrence relations or ranking functions. Reasoning with two potential functions promotes compositional reasoning by focusing on the sequential composition of programs.

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 (and expected) cost can be bounded by $|[x, y]| = \max(y-x, 0)$.

```
while (x<y) { x=x+1; tick(1); }
```
(*simple-det*)

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 $\{\Phi_0\}x = x + 1;\ \mathsf{tick}\,(1)\{\Phi_0\}$. 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 potential before and after the assignment *x=x+1* holds. This means that we have the potential $|[x, y]| + 1$ before the statement $\mathsf{tick}\,(1)$. Since $\mathsf{tick}\,(1)$ consumes one resource unit, we end up with potential $|[x, y]|$ after the loop body and have established the loop invariant again.

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 $\{\Phi\}c\{\Phi'\}$ 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' with respect the distribution over final states resulting from the program's execution. More precisely, using our MDP semantics from Section 2, the two functions must satisfy the constraint

$$\Phi(\sigma) \geq \mathsf{ExpRew}^{\mathfrak{M}^0_\sigma[\![c, \mathscr{D}]\!]}(\mathsf{Term}) + \mathbb{E}_{[\![c, \mathscr{D}]\!](\sigma)}(\Phi')$$

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. The intuitive meaning is that the potential $\Phi(\sigma)$ is sufficient for paying the expected resource cost of the execution from state σ and the expected potential with respect to the probability distribution over the final states.

Let $\Phi(\sigma) \geq \mathsf{ExpRew}^{\mathfrak{M}^0_\sigma[\![c, \mathscr{D}]\!]}(\mathsf{Term}) + \mathbb{E}_{[\![c, \mathscr{D}]\!](\sigma)}(\Phi')$ and $\Phi'(\sigma'_i) \geq \mathsf{ExpRew}^{\mathfrak{M}^0_{\sigma'_i}[\![c', \mathscr{D}]\!]}(\mathsf{Term}) + \mathbb{E}_{[\![c', \mathscr{D}]\!](\sigma'_i)}(\Phi'')$ for all sample states σ'_i from $[\![c, \mathscr{D}]\!](\sigma)$, where $\mathfrak{M}^0_{\sigma'_i}[\![c', \mathscr{D}]\!]$ is the MDP that represents

the execution of c' with initial program state σ_i'. Then we can show that for all states σ

$$\Phi(\sigma) \geq \mathsf{ExpRew}^{\mathfrak{M}^0_\sigma[\![c;c',\mathscr{D}]\!]}(\mathsf{Term}) + \mathbb{E}_{[\![c;c',\mathscr{D}]\!](\sigma)}(\Phi'')$$

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 $\Phi(\sigma) \geq \mathsf{ExpRew}^{\mathfrak{M}^0_\sigma[\![c,\mathscr{D}]\!]}(\mathsf{Term}) + \mathbb{E}_{[\![c,\mathscr{D}]\!](\sigma)}(\Phi')$ for all program states σ then we have again the familiar rule for compositional reasoning.

$$\frac{\{\Phi\}c_1\{\Phi'\} \qquad \{\Phi'\}c_2\{\Phi''\}}{\{\Phi\}c_1;c_2\{\Phi''\}}$$

Here, Φ is an upper-bound on the expected resource consumption of the program $c_1;c_2$ and yet c_1 and c_2 can be analyzed separately. Similarly, all the rules to reason on deterministic programs can be imported without changes to reason on the expected resource consumption of probabilistic programs.

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

Consider the simple random walk in the listing *rdwalk1* again. As we concluded after the classic analysis, the expected resource usage of the program is $\Phi := 2|[0,x]|$. This is the potential that we have available before the loop and it will also serve as a loop invariant. To prove that it is a loop invariant, 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 $2|[0,x]|+1$?

- In the first branch, 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 $2|[0,x]|-1 = 2|[0,x'+1]|-1 = 2|[0,x']|+1$.
- Similarly, the second branch needs the initial potential $\Phi_2(x) = 2|[0,x]|+3$.

Intuitively, since we enter the first branch with probability $\frac{3}{4}$ and the second branch only with probability $\frac{1}{4}$, we are tempted to derive the initial potential for the probabilistic branching with the weighted sum $\frac{3}{4}\Phi_1(x) + \frac{1}{4}\Phi_2(x) = 2|[0,x]|$. Doing so would restore the loop invariant and prove the desired bound.

The beauty of the potential method for expected resource consumption is that this reasoning is 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* of *rdwalk1* in Figure 5. The program simulates a general *one-dimensional random walk* [42] 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 $\mathsf{tick}(T)$, is bounded if and only if $(\star)$ $pK_1-(1-p)K_2>0$. In this case, the expected distance for "forward walking" is bigger than the expected distance for backward walking in each iteration.

The analysis for this example is very similar to the one of *rdwalk1*. It is only valid when the condition $(\star)$ is satisfied since the initial potential function would be negative otherwise. In the

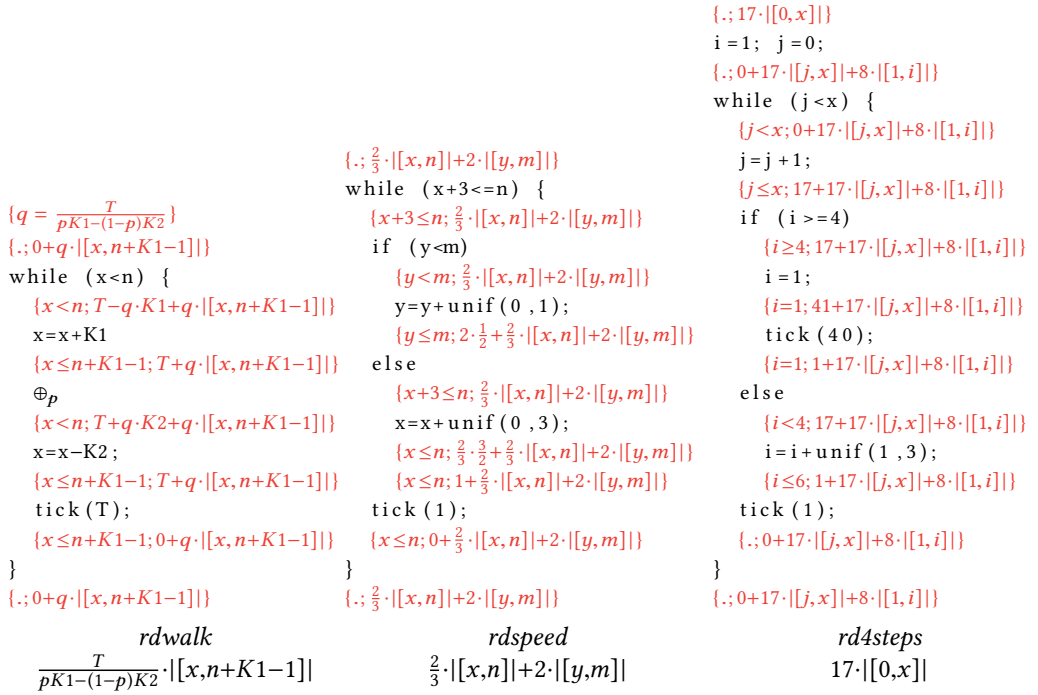

Fig. 5. Derivations of bounds on the expected value of ticks for probabilistic programs with a single loop and tricky iteration patterns.

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 $K2 + K1 + 1$ and if $K1 > 1$ the boundary condition argument we gave in Section 3.1 would not be valid anymore.

3.4 Bound derivations for challenging examples

In this section, we show how the expected potential method can derive polynomial bounds for challenging probabilistic programs with single, nested, and sequential loops, as well as recursive functions. All the presented bound derivations can be derived automatically by our tool Absynth. Absynth and the inference algorithm are discussed in Sections 5 and 7. In this subsection, we just demonstrate the expected potential method as a method for manual bound derivation. We point out again that this is very similar to existing methods that are based on weakest preconditions [58, 67].

Single loops. Examples *rdwalk*, *rdspeed* and *rd4steps* in Figure 5 demonstrate that our expected potential method can handle *tricky iteration patterns*. The exmaple *rdwalk*, has already been discussed in the previous subsection. Examples *rdspeed* and *rd4steps* are randomized versions of examples from papers on worst-case bound analysis [19, 44]. In *rdspeed*, the iteration first increases y by 0 or 1 until it reaches m, where 0 or 1 are picked randomly according to the uniform distribution. If $y \geq m$ 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 of the result. Then we compute the weighted sum on the resulting initial potentials where the weights are assigned following the uniform 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.

Example *rd4steps* is a loop that performs an expensive operation every time $i \geq 4$. In this case, i is reset to one. Otherwise, i is increase by a random variable drawn from uniform distribution. Absynth is able to infer the tight bound $17 \cdot |[0,x]|$ on the expected cost. The derivation of the bound is interesting but all the individual steps work as discussed before.

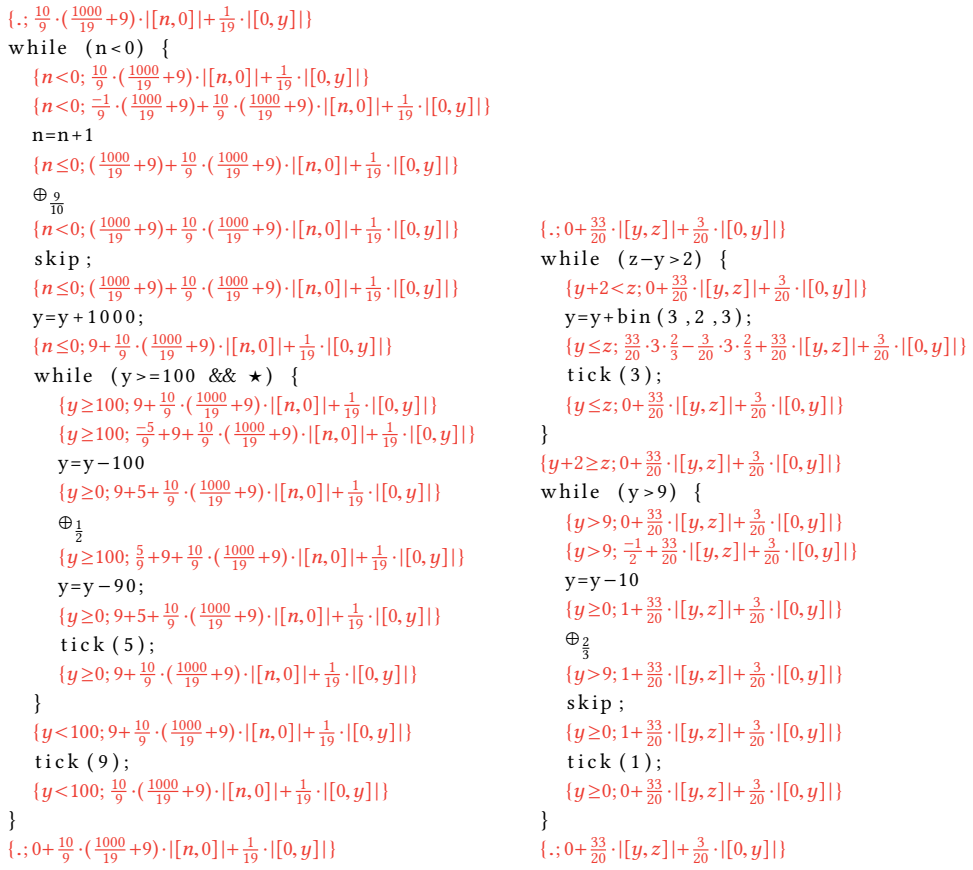

Fig. 6. Program *prnes* contains an interacting nested loop and example *prseq* contains a sequential loop so that the iterations of the second loop depend on the the first one.

Nested and sequential loops. The examples in Figure 6 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 && ⋆ 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 $n = 3$ and $p = \frac{2}{3}$. In the second loop, y is decreased by 10 with probability $\frac{2}{3}$ or left unchanged otherwise. The potential method accurately derives the expected value of the size change of y by transferring the potential $|[y,z]|$ to $|[0,y]|$.

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

```
trader(int s_min, int s) {
  {.; 0+inv(s, s_min)}
  assume (s_min >= 0);
  {s_min≥0; 0+inv(s, s_min)}
  while (s > s_min) {
    {s_min≥0∧s>s_min; 0+inv(s, s_min)}
    {s_min≥0∧s>s_min; 15+15·(|[0, s_min]|+|[s_min, s]|)+inv(s, s_min)}
    s=s+1
    {s_min≥0∧s≥s_min; 5·(|[0, s_min]|+|[s_min, s]|)+inv(s, s_min)}
    ⊕1/4
    {s_min≥0∧s>s_min; -5-5·(|[0, s_min]|+|[s_min, s]|)+inv(s, s_min)}
    s=s-1;
    {s_min≥0∧s≥s_min; 5·(|[0, s_min]|+|[s_min, s]|)+inv(s, s_min)}
    trade(s);
    {s_min≥0; 0+inv(s, s_min)}
  }
  {s_min≥0∧s≤s_min; 0+inv(s, s_min)}
}
```

rdbub: $3\cdot|[0,n]|^2$

trader: $10|[s_{\min},s]|\cdot|[0,s_{\min}]| + 10\binom{|[s_{\min},s]|}{2} + 10|[s_{\min},s]|$

Fig. 7. Derivations of polynomial bounds on the expected value of ticks for programs with nested loops and procedure calls. In the derivation of *trader* we derive the bound $\mathrm{inv}(s,s_{\min}) := 10|[s_{\min},s]|\cdot|[0,s_{\min}]| + 10\binom{|[s_{\min},s]|}{2} + 10|[s_{\min},s]|$ on the global variable *cost* from the stock price example in Figure 1. The procedure *trade(s)* has the bound $5\cdot|[0,s]|$ over the input parameter, which can be derived like in the previous examples.

Non-linear bounds. The examples in Figure 7 have non-linear bounds that demonstrate that our expected potential method can handle programs with nested loops and procedure calls which often have a super-linear expected resource consumption. 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 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 $3{\cdot}|[0,n]|^2$ 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 derivation for the program *sprdwalk*. However, the annotation carries an extra quadratic part which is invariant in the inner loop and just passed along in the derivation. The derived bound for *trader* is equal to the bound that has been mentioned in the introduction. The representation we use here should make the steps in the derivation more clear. In the derivation $\text{inv}(s, s_{\min})$ acts as a loop invariant. We assume that we already established the bound $5|[0,s]| = 5(|[0,s_{\min}]| + |[s_{\min},s]|)$ on the expected cost of the function *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 preconditions of the branches is equal to $\text{inv}(s, s_{\min})$ if the weights are $\frac{1}{4}$ and $\frac{3}{4}$. Second, verify that the potential in the post conditions is equal to the potential in the respective precondition if we substitute $s+1$ (or $s-1$) for s.

4 DERIVATION SYSTEM FOR PROBABILISTIC QUANTITATIVE ANALYSIS

In this section, we describe the inference system used by our analysis. The inference system is presented like a program logic and enables compositional reasoning. Additionally, 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 potential functions*. Finding the suitable base potential functions for a given program is discussed in Section 7. For now we assume a given list of N base functions. For convenience, base functions can be represented as a vector $B = (b_1, \ldots, b_N)$, where each $b_i : \Sigma \to \mathbb{Q}$ maps memory 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 it is convenient to store the coefficients q_i as a vector $Q = (q_1, \ldots, q_N)$. We call this vector a *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 bilinearity of the dot product, operations on the vector space of annotations correspond directly to operations on potential functions:

$$\Phi_{\lambda Q + \mu Q'} = \langle \lambda Q + \mu Q' \cdot B \rangle = \lambda \langle Q \cdot B \rangle + \mu \langle Q' \cdot B \rangle = \lambda \Phi_Q + \mu \Phi_{Q'}. \tag{3}$$

In the following, we assume that the constant base function $\mathbf{1}$ defined by $\mathbf{1}(\sigma) := 1$ is in the list of base functions B. This way, the constant potential function $K(\sigma) := k$ 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. The pre- and postcondition are pairs $\{\Gamma, Q\}$ where Γ is a logical context and Q is a potential annotation. The *logical context* $\Gamma \in \mathscr{P}(\Sigma)$ is a predicate on memory states (a set of states) inferred in our implementation using abstract interpretation.

$\boxed{\vdash \Delta}$ Validity of the context Δ

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

$\boxed{\Delta \vdash \{\Gamma; Q\}c\{\Gamma'; Q'\}}$ Derivability of the triple $\{\Gamma; Q\}c\{\Gamma'; Q'\}$ in the context Δ (elided for concision)

$$\text{(Q:NonDet)}\quad \frac{\vdash \{\Gamma; Q\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}\ \text{if } \star \text{ } c_1 \text{ else } c_2\ \{\Gamma'; Q'\}} \qquad \text{(Q:Skip)}\quad \frac{}{\vdash \{\Gamma; Q\}\ \text{skip}\ \{\Gamma; Q\}} \qquad \text{(Q:Abort)}\quad \frac{}{\vdash \{\Gamma; 0\}\ \text{abort}\ \{\Gamma'; Q'\}}$$

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

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

$$\text{(Q:PIf)}\quad \frac{0 \le p \le 1 \qquad Q = p{\cdot}Q_1 + (1-p){\cdot}Q_2 \qquad \vdash \{\Gamma; Q_1\}c_1\{\Gamma'; Q'\} \qquad \vdash \{\Gamma; Q_2\}c_2\{\Gamma'; Q'\}}{\vdash \{\Gamma; Q\}c_1 \oplus_p c_2\{\Gamma'; Q'\}}$$

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

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

$$\text{(Q:Call)}\quad \frac{P : (\Gamma; Q, \Gamma'; Q') \in \Delta \qquad x \in \mathbb{Q}_0^+}{\vdash \{\Gamma; Q + x\}\ \text{call}\ P\ \{\Gamma'; Q' + x\}} \qquad \text{(Relax)}\quad \frac{F = (F_1, \ldots, F_N) \qquad \vec{u} = (u_1, \ldots, u_N)^\intercal \qquad \forall i.\, \Gamma \models \Phi_{F_i} \ge 0 \qquad \forall i.\, u_i \ge 0 \qquad Q' = Q - F\vec{u}}{Q \ge_\Gamma Q'}$$

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

Fig. 8. Inference rules of the probabilistic quantitative analysis.

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 $\mathtt{skip}$ as the command c' and derive a triple where $\Phi_{Q'} = 0$. In that case, Φ_Q is a bound on the expected cost of the command c. Section 6 gives a formal definition of the semantics of triples.

To handle procedure calls, the derivability 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. The judgement $\vdash \Delta$ is defined in Figure 8 and 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 body. Note that a context Δ can contain multiple specifications for the same procedure. This allows a context-sensitive analysis of procedures.

To find a bound on the expected resource use of a command c, one has to use the two judgements: first, derive a triple $\Delta \vdash \{\Gamma; Q\}c\{.; .\}$ in a specification context Δ; and second, prove that this context is valid $\vdash \Delta$.

Notations and conventions. A logical context Γ is a set of permitted memory states at a given program point. In the rules, we use the following notations for contexts. The entailment relation on logical contexts $\Gamma \models \Gamma'$ means that Γ is stronger than Γ', i.e., $\Gamma \subseteq \Gamma'$. For a memory state σ, we write $\sigma \models \Gamma$ when $\sigma \in \Gamma$. For an arbitrary 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 $\{\sigma \mid \sigma[e/x] \models \Gamma\}$. Finally, we also use the notation $[\![x = e]\!]$ for the function on states defined by $[\![x = e]\!](\sigma) := \sigma[[\![e]\!]_\sigma/x]$.

When two potential annotations $Q = (q_i)$ and $Q' = (q'_i)$ are constrained using a relation $Q \diamond Q'$ for $\diamond \in \{<, =, \ldots\}$, it means that their components are constrained point-wise $\bigwedge_i q_i \diamond q'_i$. Additionally, for $c \in \mathbb{Q}$, we write $Q \pm c$ for the potential annotation Q' obtained from Q by changing the coefficient q'_i of the base function $\mathbf{1}$ to $q_i \pm c$ and leaving all 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 should become clearer when we explain the meaning of $\geq_.$ in the next section.

4.3 Inference rules

The complete set of inference rules is given in Figure 8. We informally describe some important rules and justify their validity. Section 6 gives details about the formal soundness proof.

The Q:Tick rule is the only one that accounts for the effect of consuming resources on potential functions. Using the informal semantics for a triple, we justify this rule as follows. First, the $\mathtt{tick}$ command does not change the memory state, so we require the logical context Γ in the pre- and postcondition to be the same. Now, let $Q' = Q - q$ and c' be a command with expected cost $\Phi_{Q'} = \Phi_{Q-q} = \Phi_Q - q$. Because the cost of $\mathtt{tick}\,(q)$ is exactly q units of time, the cost of the compound command $\mathtt{tick}\,(q); c'$ is exactly $\Phi_{Q'} + q$. But by definition of Q' we have $\Phi_{Q'} - q = \Phi_Q$. Thus, the triple $\{\Gamma, Q\}\ \mathtt{tick}\,(q)\ \{\Gamma, Q - q\}$ is sound; justifying the validity of the rule Q:Tick.

The rule Q:Pif accounts for probabilistic branching. Again, we justify it using the informal semantics of triples. Let c' be a continuation command with an expected resource bound $\Phi_{Q'}$. Our goal is to find an expected resource bound T_c of the command $c := (c_1 \oplus_p c_2); c'$. In proportion, p of the executions of c are executions of $c_1; c'$; and $1 - p$ of the executions of c are executions of $c_2; c'$. Let T_1 and T_2 be the execution resource cost of $c_1; c'$ and $c_2; c'$, respectively. Now, using the hypothesis triples for c_1 and c_2, we know that a bound on T_1 is given by Φ_{Q_1} and a bound on T_2 is

given by Φ_{Q_2}. Thus, using the linearity of the expected value, a have $T_c = p \cdot T_1 + (1-p) \cdot T_2 \leq p \cdot \Phi_{Q_1} + (1-p)\Phi_{Q_2} = \Phi_{p \cdot Q_1 + (1-p) \cdot Q_2}$, where the last equality is justified by the equation (3) in Section 4.1.

The second probabilistic rule Q:Sample deals with sampling assignments. Recall that R in the sampling assignment is a random variable following a distribution μ_R. The essence of the Q:Sample rule is that, since we assumed that R is bounded, we can treat a sampling assignment as a 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 Q:Assign rule for the command $x = e$, 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 $\Phi_{Q'} \circ [\![x = e]\!]$ is a bound on the expected cost of $x = e; c'$. (This is similar to the usual backward presentation of the Hoare logic rule for state updates.) Thus, we are looking for a potential annotation Q such that $\Phi_Q = \Phi_{Q'} \circ [\![x = e]\!]$. To encode this constraint as a linear program (this is necessary to enable automation using LP solving), we find all the *stable* base functions; that is, all the base functions b_j for which there exists coefficients $(a_{i,j})_i$ such that $b_j \circ [\![x = e]\!] = \sum_i a_{i,j} \cdot b_i$. The set of stable base functions for an assignment $x = e$ is noted $\mathscr{S}_{x=e}$ in the rule. Finally, to ensure that the transformation $[\![x = e]\!]$ 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 $\Phi_Q = \Phi_{Q'} \circ [\![x = e]\!] = \Phi_{AQ'}$ where A is the matrix with coefficients $(a_{i,j})$; hence justifying the validity of the rule.

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 $\Phi_{Q'}(\sigma) \geq \Phi_Q(\sigma)$; this result is formally stated in Lemma C.1. 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. In the rule Q:Weaken, the logical contexts used for the weakening of the pre- and postconditions are chosen to be the strongest available; that is, Γ for the precondition and Γ'_0 for the postcondition. This ensures that the strongest assumptions are available when proving the non-negativity of the rewrite functions.

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* $x \in \mathbb{Q}_0^+$ 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 expected runtime transformer [74] used in our formal soundness proof.

5 AUTOMATIC CONSTRAINT GENERATION AND SOLVING USING LP SOLVERS

Overview. The implementation of our automated tool automatically infers derivations in the system of Figure 8. The search process is split in two phases. First, a derivation template is created by applying inductively the derivation rules on the input program. During this first phase, the coefficients of the potential annotations are left as symbolic names. Each symbolic name corresponds

to a variable in a linear program that is constrained according to the rules of Figure 8. Second, we feed the linear program to an off-the-shelf LP solver. If the LP solver returns a satisfying assignment for the linear program, we obtain a valid derivation and extract the expected resource bound. To do so, the symbolic names used in the initial potential annotation are replaced with the coefficients returned by the LP solver. If the LP solver fails to find a solution, an error is reported to the user.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{. \vdash \{x \geq 2; Q^{d1}\} x = x - 1 \{.; P^{d1}\}}\,\text{Q:Assign}_1}{. \vdash \{x \geq 2; Q^{w1}\} x = x - 1 \{.; P^{w1}\}}\,\text{Q:Weaken}_1 \quad \dfrac{\dfrac{}{. \vdash \{x \geq 2; Q^{d2}\} x = x - 2 \{.; P^{d2}\}}\,\text{Q:Assign}_2}{. \vdash \{x \geq 2; Q^{w2}\} x = x - 2 \{.; P^{w2}\}}\,\text{Q:Weaken}_2}{. \vdash \{x \geq 2; Q^{pi}\} x = x - 1 \oplus_{\frac{1}{3}} x = x - 2 \{.; P^{pi}\}}\,\text{Q:If}$$

$$\vdots \qquad \dfrac{}{. \vdash \{.; Q^{ti}\}\ \text{tick}\,(1) \{.; P^{ti}\}}\,\text{Q:Tick}$$

$$\dfrac{. \vdash \{x \geq 2; Q^{sq}\}\ x = x - 1 \oplus_{\frac{1}{3}} x = x - 2;\ \text{tick}\,(1) \{.; P^{sq}\}}{. \vdash \{.; Q\}\ \ \text{while}\,(x >= 2) \{x = x - 1 \oplus_{\frac{1}{3}} x = x - 2;\ \text{tick}\,(1)\}\ \{x < 2; P\}}\,\text{Q:Loop}$$

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:If
$Q^{w1} \succeq_{(x \geq 2)} Q^{d1} \wedge P^{d1} \succeq_{(x \geq 2)} P^{w1}$	Q:Weaken$_1$
$q_1^{d1} = p_1^{d1} \wedge q_{x0}^{d1} = 0 \wedge q_{x1}^{d1} = p_{x0}^{d1} \wedge q_{x2}^{d1} = p_{x1}^{d1} \wedge p_{x2}^{d1} = 0$	Q:Assign$_1$
$Q^{w2} \succeq_{(x \geq 2)} Q^{d2}; P^{d2} \succeq_{(x \geq 2)} P^{w2}$	Q:Weaken$_2$
$q_1^{d2} = p_1^{d2} \wedge q_{x0}^{d2} = 0 \wedge q_{x1}^{d2} = 0 \wedge q_{x2}^{d2} = p_{x0}^{d2} \wedge p_{x1}^{d2} = 0 \wedge p_{x2}^{d1} = 0$	Q:Assign$_2$
$Q^{ti} = P^{ti} + 1$	Q:Tick

Fig. 9. Inference of a derivation using linear constraint solving.

Generating linear constraints. A detailed example of this process is shown in Figure 9. Both the template derivation and the constraints inferred by the first phase of the analysis are shown at the top of the figure. 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. As explained earlier, the individual coefficients in those annotations are symbolic names, each associated with an LP variable. For example Q, the precondition of the whole program, represents the potential function

$$q_1 \cdot \mathbf{1} + q_{x0} \cdot |[0, x]| + q_{x1} \cdot |[1, x]| + q_{x2} \cdot |[2, x]| \tag{4}$$

As in the examples, we use the notation $|[a, b]|$ to denote $\max(0, b - a)$. For this example, the only base functions we consider are the constant function $\mathbf{1}$ and the three interval sizes $|[i, x]|$ for $i \in \{0, 1, 2\}$. We will see that they are sufficient to infer a bound. To apply weakenings, we are going to need some rewrite functions; in this example we pick

$$F_0 = 1 \qquad F_1 = |[0, x]| - |[1, x]| - 1 \qquad F_2 = |[0, x]| - |[2, x]| - 2\ .$$

The rewrite function F_1 is applicable (i.e., non-negative) if and only if $x \geq 1$. For example $F_1 = -1$, when x is 0. Similarly, the rewrite function F_2 is applicable if and only if $x \geq 2$. This means, in particular, that both rewrite functions can be used at the beginning of the loop body, when $x \geq 2$ can be proved because of the loop condition.

The constraints given in the table in Figure 9 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 in this derivation 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 weigths $\frac{1}{3}$ 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 $Q^{w1} \succeq_{(x \geq 2)} Q^{d1}$. This relation is defined by the RELAX rule of Figure 8 and involves finding all the applicable rewrite functions non-negative 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.

$$\begin{pmatrix} q_1^{d1} \\ q_{x0}^{d1} \\ q_{x1}^{d1} \\ q_{x2}^{d1} \end{pmatrix} = \begin{pmatrix} q_1^{w1} \\ q_{x0}^{w1} \\ q_{x1}^{w1} \\ q_{x2}^{w1} \end{pmatrix} - \begin{pmatrix} 1 & -1 & -2 \\ 0 & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \quad \wedge \quad \begin{pmatrix} u_0 \\ u_1 \\ u_2 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

In this system, 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.

Finally, we discuss the first assignment Q:ASSIGN$_1$. For this assignment, the stable set discussed in Section 4.3 is $\mathscr{S}_{x=x-1} = \{1, x0, x1\}$. Indeed, the only unstable base function is $|[2, x]|$ that becomes $|[3, x]|$ when composed with the assignment $x = x - 1$. Since the assignment leaves $\mathbf{1}$ unchanged and changes $|[0, x]|$ into $|[1, x]|$ and $|[1, x]|$ into $|[2, x]|$, the system of constraints generated is

$$\begin{pmatrix} q_1^{d1} \\ q_{x0}^{d1} \\ q_{x1}^{d1} \\ q_{x2}^{d1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} p_1^{d1} \\ p_{x0}^{d1} \\ p_{x1}^{d1} \\ p_{x2}^{d1} \end{pmatrix} \quad \wedge \quad p_{x2}^{d1} = 0 \,, \tag{6}$$

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

Solving the constraints. When passing the constraints to the LP solver for solving, we have the opportunity to ask the solver to optimize a certain linear 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 bound to optimize is given by (4) above. 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 second objective function for the only constant coefficient is simply q_1. In our example, the final solution returned by the LP solver has $q_{x0} = \frac{3}{5}$ 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 the operational semantics of Section 2. It leverages previous work on probabilistic programs by relying on the soundness of a weakest

pre-expectation calculus [58, 74]. This calculus provides a way to compute the expected reward of the MDP derived from a probabilistic program.

6.1 Weakest pre-expectation transformer

A weakest pre-expectation calculus [41, 67] 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 1; it operates on the set of functions $\mathbb{T} := \{f \mid f : \Sigma \to \mathbb{R} \cup \{\infty\}\}$, 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. 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 [41, 67].

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.

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

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

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

Table 1. Definition of the expected cost transformer 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 functional F.

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 6.1 (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 [58, 74]. □

6.2 Soundness of the automatic analysis

To prove the soundness of our automated analysis, we first interpret the pre- and postconditions of our triples as expectations. This interpretation is defined as 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 where the logical context Γ 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 by using the expected cost transformer of the previous section.

Theorem 6.2 (Soundness of the automatic analysis). *Let c be a command in a larger program $(\_, \mathscr{D})$. If $\Delta \vdash \{\Gamma; Q\}c\{\Gamma'; Q'\}$ is derivable, then for every program state $\sigma \in \Sigma$, the following holds*

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

Proof. The proof is done by induction on the program structure and the derivation using the inference rules. See Appendix C.2 for details. □

7 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

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. [1]

7.1 Implementation

Absynth is implemented in OCaml and consists of about 5000 lines of code. The tool currently works on imperative integer programs written in a Python-like syntax that supports recursive procedures. Absynth has also a C interface based on LLVM.

To discovery the bounds on expected resource usage automatically, we limit the format of the potential functions that we are looking for. In this work, we focus on inferring polynomial potential functions that are *linear combinations* of *base potential functions* picked among the monomials. Formally, they are defined as follows.

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

The abstract interpretation procedure used in Absynth to infer logical contexts derives linear inequalities between program variables. It uses a Presburger decision procedure that is also used to prove the non-negativity of rewrite functions.

In our implementation, we use the invariants produced using a simple abstract interpretation 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.

When the heuristic adds a new base function, a set of rewrite functions is also added to allow transfers of potential to and from the new base function. For instance, for the base function $\max(0, n - x)$ the heuristic adds the rewrite function $F = \max(0, n - x) - \max(0, n - x - 1) - 1$. The rewrite function F can be used before 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.

Absynth supports four common distributions for random sampling assignments: 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.

7.2 Experimental evaluation

Evaluation setup. To evaluated the practicality of our framework, we have designed and collected more than 40 challenging examples with different looping and recursion patterns that use probabilistic branching and sampling assignments. In total, the benchmark consists of more than 1000 lines of code.

The programs *bayesian_net* [40], *geo*, *coupon*, *filling_vol* [76], *race*, *2drwalk*, *robot*, *roulette* [22], and *rejection_sampling* [58] have been described in the literature on probabilistic programs. In which the expected resource consumption has been analyzed manually or the programs have been used in a different context. The programs *C4B_**, *prseq*, *prseq*, *preseq_bin* (replace unif by bin), *prspeed*, *rdseql*, *rdspeed*, *recursive* are probabilistic versions of deterministic examples from previous work [18, 19, 44]. 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.

[1] The source code of the examples and Absynth have been submitted as auxiliary material.

Program	Expected bound	LP time	Time(s)
Linear programs			
2drwalk	$2\cdot\lvert[d, n+1]\rvert$	77%	1.929
bayesian_net	$5\cdot\lvert[0, n]\rvert$	51%	0.223
ber	$2\cdot\lvert[x, n]\rvert$	64%	0.008
bin	$0.2\cdot\lvert[0, n+9]\rvert$	83%	0.245
C4B_t09	$17\cdot\lvert[0, x]\rvert$	56%	0.051
C4B_t13	$1.25\cdot\lvert[0, x]\rvert + \lvert[0, y]\rvert$	63%	0.021
C4B_t15	$1.33333\cdot\lvert[0, x]\rvert$	58%	0.043
C4B_t19	$\lvert[0, k+i+51]\rvert + 2\cdot\lvert[0, i]\rvert$	63%	0.035
C4B_t30	$0.5\cdot\lvert[0, x+2]\rvert + 0.5\cdot\lvert[0, y+2]\rvert$	58%	0.021
C4B_t61	$0.5\cdot\lvert[0, l-1]\rvert + \lvert[0, l]\rvert$	70%	0.027
condand	$\lvert[0, m]\rvert + \lvert[0, n]\rvert$	63%	0.010
cooling	$0.42\cdot\lvert[0, t+5]\rvert + \lvert[\text{st}, \text{mt}]\rvert$	59%	0.064
coupon	21	62%	0.070
fcall	$2\cdot\lvert[x, n]\rvert$	47%	0.008
filling_vol	$0.017264\cdot\lvert[0, \text{vol}+2]\rvert + 0.333333\cdot\lvert[0, \text{vol}+10]\rvert + 0.316069\cdot\lvert[0, \text{vol}+11]\rvert$	64%	0.475
geo	5	65%	0.009
hyper	$5\cdot\lvert[x, n]\rvert$	62%	0.013
linear01	$0.6\cdot\lvert[0, x]\rvert$	55%	0.010
prdwalk	$1.14286\cdot\lvert[x, n+4]\rvert$	74%	0.053
prnes	$68.4795\cdot\lvert[0, -n]\rvert + 0.052631\cdot\lvert[0, y]\rvert$	68%	0.059
prseq	$1.65\cdot\lvert[y, x]\rvert + 0.15\cdot\lvert[0, y]\rvert$	69%	0.040
prseq_bin	$1.65\cdot\lvert[y, x]\rvert + 0.15\cdot\lvert[0, y]\rvert$	72%	0.051
prspeed	$2\cdot\lvert[y, m]\rvert + 0.666667\cdot\lvert[x, n]\rvert$	62%	0.046
race	$0.666667\cdot\lvert[h, t+9]\rvert$	83%	0.208
rdseql	$2.25\cdot\lvert[0, x]\rvert + \lvert[0, y]\rvert$	62%	0.022
rdspeed	$2\cdot\lvert[y, m]\rvert + 0.666667\cdot\lvert[x, n]\rvert$	54%	0.033
rdwalk	$2\cdot\lvert[x, n+1]\rvert$	69%	0.012
rejection_sampling	$2\cdot\lvert[0, n]\rvert$	69%	2.635
rfind_mc	$\lvert[0, k]\rvert$	50%	0.028
robot	$0.384615\cdot\lvert[0, N+6]\rvert$	60%	1.996
roulette	$4.93333\cdot\lvert[n, 20]\rvert$	83%	0.737
sprdwalk	$2\cdot\lvert[x, n]\rvert$	70%	0.010
trapped_miner	$7.5\cdot\lvert[0, n]\rvert$	56%	0.059
Polynomial programs			
multirace	$2\cdot\lvert[0, m]\rvert\cdot\lvert[0, n]\rvert + 4\cdot\lvert[0, n]\rvert$	82%	7.825
pol04	$4.5\cdot\lvert[0, x]\rvert^2 + 7.5\cdot\lvert[0, x]\rvert$	61%	0.437
pol05	$\lvert[0, x+2]\rvert\cdot\lvert[0, x]\rvert + 0.125\cdot\lvert[0, x]\rvert^2$	77%	1.287
pol06	$0.624998\cdot\lvert[\min, s]\rvert + 2\cdot\lvert[\min, s]\rvert\cdot\lvert[0, \min]\rvert + 0.625\cdot\lvert[\min, s]\rvert^2$	81%	6.619
pol07	$1.5\cdot\lvert[0, n-2]\rvert\cdot\lvert[0, n-1]\rvert$	89%	4.477
rdbub	$3\cdot\lvert[0, n]\rvert^2$	50%	0.152
recursive	$0.25\cdot\lvert[\text{arg\_l}, \text{arg\_h}]\rvert^2 + 1.75\cdot\lvert[\text{arg\_l}, \text{arg\_h}]\rvert$	53%	3.330
trader	$5\cdot\lvert[s_{\min}, s]\rvert^2 + 10\cdot\lvert[s_{\min}, s]\rvert\cdot\lvert[0, s_{\min}]\rvert + 5\cdot\lvert[s_{\min}, s]\rvert$	78%	7.262

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

The experiments were run on a machine with an Intel Core i5 2.4 GHz processor and 8GB of RAM under Mac OS X 10.12.5. The LP solver we use in the experiments is CoinOr's CLP.

Results. The results of the evaluation are compiled in Table 2. The table is split into linear and non-linear bounds and contains the inferred bounds, the total time taken by Absynth, and the proportion of the time that was spent by the LP solver.

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. One could speed up the analysis with a control-flow analysis that excludes variables that do not influence the control flow from base functions. Currently, 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 price we pay for the precision is a linear constraint set that is proportional to the range of the sampling distribution. As expected, most of the analysis time is spent on LP solving.

As argued in Section 3, the derived bounds are often not only asymptotically tight but also contain the exact constant factors of the expected resource consumption. The bounds of all presented examples seem to be asymptotically tight. However, there is no guarantee that Absynth infers tight bounds and there is a large class of bounds that Absynth cannot derive. For example, the best bound that Absynth can derive for a program with expected logarithmic cost is a linear bound.

8 RELATED WORK

Our work is a confluence of ideas from automatic resource bound analysis and analysis of probabilistic programs. These two areas have been extensively studied but developed largely 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. Considering automatic resource bound analysis, most closely related to our work is prior work on automatic amortized resource analysis (AARA) for deterministic programs. AARA has been introduced by Hofmann and Jost [51] for automatically deriving linear worst-case bounds for first-order functional programs with lists. The technique has been generalized to derive polynomial bounds [47, 49, 50, 53, 54], lower bounds [70], and to handle (strictly evaluated) programs with arrays and references [65], higher-order functions [48, 56], and user defined data types [48, 57]. A distinctive common theme that we share with prior work is compositionality and automatic bound inference via LP solving.

AARA has also been used to derive linear bounds for lazy functional programs [77, 80] and object-oriented programs [52, 55]. Moreover, the technique has been integrated into separation logic [4] to derive bounds that depend on mutable data-structures and to manually verify the complexity of algorithms and data-structures using proof assistants [24, 72]. 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, 71]. Our contribution is the application of existing techniques for automation [19] and inference of polynomial bounds [18] to derive 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 [79], linear dependent types [63, 64], refinement types [28, 83], annotated type systems [30, 31], defunctionalization [5], recurrence relations [1, 2, 12, 32, 36, 43, 61], abstract interpretation [13, 21, 44, 78, 82], template based assume-guarantee reasoning [66], measure functions [26], and techniques from term rewriting [6, 16, 37, 73]

Obviously, 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 resource usage (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. [58, 74]. Like in our work, the goal of this line of research is to characterize the expected runtime of probabilistic programs. However, they use a weakest precondition 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, 62, 67]. It has been automated using constraint generation [60] and abstract interpretation [23] to derive quantitative invariants. While such invariants can be used to derive facts about the expected resource usage, it is unclear how to use them to automatically derive symbolic (polynomial) bounds like in this work. Hickey et al. [46] rely on Kozen's foundations to address the derivation of recurrence relations for probabilistic programs. In contrast to this work, they are not focusing on closed form bounds and it is unclear how practical the approach is.

Another body of research relies probabilistic pushdown automata and martingale theory to analyze the termination time [15]. They give more fine grained information on the distribution of the termination time and use existing methods for deriving bounds on the expected number of steps [33].

The use of martingale theory to automatically analyze probabilistic programs has been pioneered by Sankaranarayanan et al. [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 but also focus on termination.

Abstract interpretation has also been applied to probabilistic programs [29, 68, 69] but we are not aware of an application of an abstract interpretation to derive bounds on the expected resource usage. An approach to automatically analyze programs with bounded loops is based on symbolic inference [38]. Another static technique analyzes execution paths to derive probability bounds on integer assertion [76]. Other analyses are based on statistical techniques [14, 39].

In the context of analyzing differential privacy, there are works that focus on deriving bounds on the privacy budget for probabilistic programs [9, 45]. While these systems can be seen as a form of resource analysis, they are concerned with worst-case bounds and automation is limited.

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.

While bounds on the expected resource usage are useful for many applications (e.g., rejection sampling), it is sometimes necessary to have more fine-grained information on the resource usage. 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. Another direction we are currently studying is the adaption of the new analysis to derive average-case bounds for deterministic programs. 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 [48] to apply the expected potential method to (higher-order) functional programs.

REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] G. Barthe, M. Gaboardi, J. Hsu, and B. Pierce. Programming Language Techniques for Differential Privacy. *ACM SIGLOG News*, 3(1):34–53, Feb. 2016.

[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] 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.
[41] 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.
[42] G. Grimmett and D. Stirzaker. *Probability and Random Processes.* Oxford University Press, 1992.
[43] B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.
[44] 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.
[45] 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.
[46] T. J. Hickey and J. Cohen. Automating Program Analysis. *J. ACM*, 35(1):185–220, 1988.
[47] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.
[48] 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.

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

[50] 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.

[51] 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.

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

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

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

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

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

[65] 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.

[66] 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.

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

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

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

[70] 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.

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

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

[73] 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.

[74] 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.

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

[76] 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.

[77] 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.

[78] 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.
[79] P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.
[80] 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.
[81] W. Wechler. Universal Algebra for Computer Scientists. *EATCS Monographs on Theoretical Computer Science*, 25, 1992.
[82] 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.
[83] 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 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 $\sigma \in \Sigma$.

A.1 The ω-complete partial order

Lemma A.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 as follows.

$$\sup\, S = \lambda\sigma.\ \sup\{f(\sigma) \mid f \in S\}$$

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.

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

Reflexivity.

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

Transitivity.

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

□

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 A.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.*

$$
\begin{array}{rl}
\textit{Continuity:} & \sup_n \operatorname{ert}[c,\mathscr{D}](f_n) = \operatorname{ert}[c,\mathscr{D}](\sup_n f_n) \\
\textit{Monotonicity:} & f \sqsubseteq g \Rightarrow \operatorname{ert}[c,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[c,\mathscr{D}](g) \\
\textit{Propagation of constants:} & \operatorname{ert}[c,\mathscr{D}](\mathbf{k}+f) \sqsubseteq \mathbf{k} + \operatorname{ert}[c,\mathscr{D}](f) \\
\textit{Sub-additivity:} & \operatorname{ert}[c,\mathscr{D}](f+g) \sqsubseteq \operatorname{ert}[c,\mathscr{D}](f) + \operatorname{ert}[c,\mathscr{D}](g) \\
 & \textit{if } c \textit{ is non-determinism-free} \\
\textit{Scaling:} & \min(1,r)\cdot \operatorname{ert}[c,\mathscr{D}](f) \sqsubseteq \operatorname{ert}[c,\mathscr{D}](r\cdot f) \\
 & \operatorname{ert}[c,\mathscr{D}](r\cdot f) \sqsubseteq \max(1,r)\cdot \operatorname{ert}[c,\mathscr{D}](f) \\
\textit{Preservation of } \infty\textit{:} & \operatorname{ert}[c,\mathscr{D}](\infty) = \infty \quad \textit{if } c \textit{ is abort-free}
\end{array}
$$

A.2 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.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[\,\mathsf{skip}\,,\mathscr{D}](f_n) & & \dagger \text{ Table 1 } \dagger \\
 & = & \sup_n f_n \\
 & & \dagger \text{ Table 1 } \dagger \\
 & = & \operatorname{ert}[\,\mathsf{skip}\,,\mathscr{D}](\sup_n f_n)
\end{array}
$$

Abort.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[\,\mathsf{abort}\,,\mathscr{D}](f_n) & & \dagger \text{ Table 1 and algebra } \dagger \\
 & = & \mathbf{0} \\
 & = & \operatorname{ert}[\,\mathsf{abort}\,,\mathscr{D}](\sup_n f_n)
\end{array}
$$

Assert.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[\,\mathsf{assert}\ e,\mathscr{D}](f_n) & & \dagger \text{ Table 1 and algebra } \dagger \\
 & = & [\![e:\ \mathsf{true}\,]\!]\cdot(\sup_n f_n) \\
 & = & \operatorname{ert}[\,\mathsf{assert}\ e,\mathscr{D}](\sup_n f_n)
\end{array}
$$

Weaken.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](f_n) & & \dagger \text{ Table 1 } \dagger \\
 & = & \sup_n f_n \\
 & & \dagger \text{ Table 1 } \dagger \\
 & = & \operatorname{ert}[\,\mathsf{weaken}\,,\mathscr{D}](\sup_n f_n)
\end{array}
$$

Tick.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[\,\mathsf{tick}\,(q),\mathscr{D}](f_n) & & \dagger \text{ Table 1 } \dagger \\
 & = & \sup_n(\mathbf{q}+f_n) \\
 & & \dagger \text{ Algebra } \dagger \\
 & = & \mathbf{q} + \sup_n f_n \\
 & & \dagger \text{ Table 1 } \dagger \\
 & = & \operatorname{ert}[\,\mathsf{tick}\,(q),\mathscr{D}](\sup_n f_n)
\end{array}
$$

Assignment.

$$
\begin{array}{rll}
\sup_n \operatorname{ert}[x=e,\mathscr{D}](f_n) & & \dagger \text{ Table 1 } \dagger \\
 & = & \sup_n(f_n[e/x]) = (\sup_n f_n)[e/x] \\
 & & \dagger \text{ Table 1 } \dagger \\
 & = & \operatorname{ert}[x=e,\mathscr{D}](\sup_n f_n)
\end{array}
$$

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

$$\begin{array}{lll} & & \dagger \text{ Table 1 } \dagger \\ & = & \sup_n(\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f_n(\sigma[e \text{ bop } v/x]))) \\ & & \dagger \text{ LMCT } \dagger \\ & = & \lambda\sigma.\mathbb{E}_{\mu_R}(\sup_n \lambda v.f_n(\sigma[e \text{ bop } v/x])) \\ & = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.\sup_n f_n(\sigma[e \text{ bop } v/x])) \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \text{ert}[x = e \text{ bop } R, \mathscr{D}](\sup_n f_n) \end{array}$$

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

$$\begin{array}{lll} & & \dagger \text{ Table 1 } \dagger \\ & = & \sup_n([\![e : \text{true}]\!] \cdot \text{ert}[c_1, \mathscr{D}](f_n) + \\ & & [\![e : \text{false}]\!] \cdot \text{ert}[c_2, \mathscr{D}](f_n)) \\ & & \dagger \text{ MST } \dagger \\ & = & \lim_{n\to\infty}([\![e : \text{true}]\!] \cdot \text{ert}[c_1, \mathscr{D}](f_n) + \\ & & [\![e : \text{false}]\!] \cdot \text{ert}[c_2, \mathscr{D}](f_n)) \\ & = & [\![e : \text{true}]\!] \cdot \lim_{n\to\infty} \text{ert}[c_1, \mathscr{D}](f_n) + \\ & & [\![e : \text{false}]\!] \cdot \lim_{n\to\infty} \text{ert}[c_2, \mathscr{D}](f_n) \\ & & \dagger \text{ MST } \dagger \\ & = & [\![e : \text{true}]\!] \cdot \sup_n \text{ert}[c_1, \mathscr{D}](f_n) + \\ & & [\![e : \text{false}]\!] \cdot \sup_n \text{ert}[c_2, \mathscr{D}](f_n) \\ & & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ & = & [\![e : \text{true}]\!] \cdot \text{ert}[c_1, \mathscr{D}](\sup_n f_n) + \\ & & [\![e : \text{false}]\!] \cdot \text{ert}[c_2, \mathscr{D}](\sup_n f_n) \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \text{ert}[\text{ if } e \ c_1 \text{ else } c_2, \mathscr{D}](\sup)_n f_n \end{array}$$

Nondeterministic branching.

$$\begin{array}{lll} & & \dagger \text{ Table 1 } \dagger \\ & = & \sup_n(\max\{\text{ert}[c_1, \mathscr{D}](f_n), \text{ert}[c_2, \mathscr{D}](f_n)\}) \\ & \sqsupseteq & \max\{\sup_n \text{ert}[c_1, \mathscr{D}](f_n), \sup_n \text{ert}[c_2, \mathscr{D}](f_n)\}\} \\ & & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ & = & \max\{\text{ert}[c_1, \mathscr{D}](\sup_n f_n), \text{ert}[c_2, \mathscr{D}](\sup)_n f_n\}\} \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \text{ert}[\text{ if } \star \ c_1 \text{ else } c_2, \mathscr{D}](\sup_n f_n) \end{array}$$

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

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

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

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

Therefore, we get

$$\begin{array}{rcl} A & = & \max\{\sup_n \mathrm{ert}\,[c_1, \mathscr{D}](f_n),\ \sup_n \mathrm{ert}\,[c_2, \mathscr{D}](f_n)\} \\ & = & \mathrm{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](\sup_n f_n) \\ & \sqsupseteq & \sup_n(\max\{\mathrm{ert}\,[c_1, \mathscr{D}](f_n),\ \mathrm{ert}\,[c_2, \mathscr{D}](f_n)\}) \\ & = & \sup_n \mathrm{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f_n) \end{array}$$

By observation above, it follows

$$\sup_n \mathrm{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f_n) = \mathrm{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](\sup_n f_n)$$

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

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

Sequence.

$$\begin{array}{rcl} \sup_n \mathrm{ert}\,[c_1; c_2, \mathscr{D}](f_n) & & \dagger\ \text{Table 1}\ \dagger \\ & = & \sup_n(\mathrm{ert}\,[c_1, \mathscr{D}](\mathrm{ert}\,[c_2, \mathscr{D}](f_n))) \\ & & \dagger\ \text{By I.H on } c_1\ \dagger \\ & = & \mathrm{ert}\,[c_1, \mathscr{D}](\sup_n \mathrm{ert}\,[c_2, \mathscr{D}](f_n)) \\ & & \dagger\ \text{By I.H on } c_2\ \dagger \\ & = & \mathrm{ert}\,[c_1, \mathscr{D}](\mathrm{ert}\,[c_2, \mathscr{D}](\sup_n f_n)) \\ & & \dagger\ \text{Table 1}\ \dagger \\ & = & \mathrm{ert}\,[c_1; c_2, \mathscr{D}](\sup_n f_n) \end{array}$$

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

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

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

$$\begin{array}{rcl} \mathrm{ert}\,[\,\mathsf{while}\ e\ c, \mathscr{D}](\sup_n f_n) & = & \sup_k \mathrm{ert}\,[\,\mathsf{while}^k e\ c, \mathscr{D}](\sup_n f_n) \\ & & \dagger\ \text{Observation above}\ \dagger \\ & = & \sup_k(\sup_n \mathrm{ert}\,[\,\mathsf{while}^k e\ c, \mathscr{D}](f_n)) \\ & = & \sup_n(\sup_k \mathrm{ert}\,[\,\mathsf{while}^k e\ c, \mathscr{D}](f_n)) \\ & & \dagger\ \text{By Theorem } B.2\ \dagger \\ & = & \sup_n(\mathrm{ert}\,[\,\mathsf{while}\ e\ c, \mathscr{D}](f_n)) \end{array}$$

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

$$\sup_n \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P](f_n) = \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P](\sup_n f_n)$$

On the other hand, by Theorem B.5, $\mathsf{ert}\,[\,\mathsf{call}\,P, \mathscr{D}] = \sup_k \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P]$, where $\mathsf{call}\,\overset{\mathscr{D}}{k}P$ is the k^{th}-inlining of procedure call, we have

$$\begin{array}{rcl}
\mathsf{ert}\,[\,\mathsf{call}\,P, \mathscr{D}](\sup_n f_n) & = & \sup_k \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P](\sup_n f_n) \\
& & \dagger \text{ Observation above } \dagger \\
& = & \sup_k(\sup_n \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P](f_n)) \\
& = & \sup_n(\sup_k \mathsf{ert}\,[\,\mathsf{call}\,\overset{\mathscr{D}}{k}P](f_n)) \\
& & \dagger \text{ By Theorem } B.5 \dagger \\
& = & \sup_n(\mathsf{ert}\,[\,\mathsf{call}\,P, \mathscr{D}](f_n))
\end{array}$$

A.3 Monotonicity

The monotonicity follows from the continuity of ert as follows.

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

A.4 Propagation of constants

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

Skip.

$$\begin{array}{rcl}
\mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](\mathbf{k} + f) & & \dagger \text{ Table 1 } \dagger \\
& = & \mathbf{k} + f \\
& & \dagger \text{ Table 1 } \dagger \\
& = & \mathbf{k} + \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](f)
\end{array}$$

Abort.

$$\begin{array}{rcl}
\mathsf{ert}\,[\,\mathsf{abort}\,, \mathscr{D}](\mathbf{k} + f) & & \dagger \text{ Table 1 and algebra } \dagger \\
& = & \mathbf{0} \\
& \sqsubseteq & \mathbf{k} + \mathsf{ert}\,[\,\mathsf{abort}\,, \mathscr{D}](f)
\end{array}$$

Assert.

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

Weaken.

$$\begin{array}{rcl}
\mathsf{ert}\,[\,\mathsf{weaken}\,, \mathscr{D}](\mathbf{k} + f) & & \dagger \text{ Table 1 } \dagger \\
& = & \mathbf{k} + f \\
& & \dagger \text{ Table 1 } \dagger \\
& = & \mathbf{k} + \mathsf{ert}\,[\,\mathsf{weaken}\,, \mathscr{D}](f)
\end{array}$$

Tick.

$$\begin{array}{rll} \mathrm{ert}\,[\,\mathrm{tick}\,(q), \mathscr{D}](\mathbf{k}+f) & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{q}+\mathbf{k}+f \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+\ \mathrm{ert}\,[\,\mathrm{tick}\,(q), \mathscr{D}](f) \end{array}$$

Assignment.

$$\begin{array}{rll} \mathrm{ert}\,[x=e, \mathscr{D}](\mathbf{k}+f) & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+f[e/x] \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+\ \mathrm{ert}\,[x=e, \mathscr{D}](f) \end{array}$$

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

$$\begin{array}{rll} \mathrm{ert}\,[x=e\ \mathrm{bop}\ R, \mathscr{D}](\mathbf{k}+f) & & \dagger \text{ Table 1 } \dagger \\ & = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(\mathbf{k}+f)(\sigma[e\ \mathrm{bop}\ v/x])) \\ & & \dagger \text{ LPE and } \mathbf{k}(\sigma[e\ \mathrm{bop}\ v/x]) = \mathbf{k}(\sigma) \dagger \\ & = & \mathbf{k}+\lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e\ \mathrm{bop}\ v/x])) \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+\ \mathrm{ert}\,[x=e\ \mathrm{bop}\ R, \mathscr{D}](f) \end{array}$$

If.

$$\begin{array}{rll} \mathrm{ert}\,[\,\mathrm{if}\ e\ c_1\ \mathrm{else}\ c_2, \mathscr{D}](\mathbf{k}+f) & & \dagger \text{ Table 1 } \dagger \\ & = & [\![e:\mathrm{true}]\!]\cdot\ \mathrm{ert}\,[c_1, \mathscr{D}](\mathbf{k}+f)+ \\ & & [\![e:\mathrm{false}]\!]\cdot\ \mathrm{ert}\,[c_2, \mathscr{D}](\mathbf{k}+f) \\ & & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ & \sqsubseteq & [\![e:\mathrm{true}]\!]\cdot(\mathbf{k}+\ \mathrm{ert}\,[c_1, \mathscr{D}](f))+ \\ & & [\![e:\mathrm{false}]\!]\cdot(\mathbf{k}+\ \mathrm{ert}\,[c_2, \mathscr{D}](f)) \\ & & \dagger \text{ Algebra } \dagger \\ & = & \mathbf{k}+[\![e:\mathrm{true}]\!]\cdot\ \mathrm{ert}\,[c_1, \mathscr{D}](f)+ \\ & & [\![e:\mathrm{false}]\!]\cdot\ \mathrm{ert}\,[c_2, \mathscr{D}](f) \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+\ \mathrm{ert}\,[\,\mathrm{if}\ e\ c_1\ \mathrm{else}\ c_2, \mathscr{D}](f) \end{array}$$

Nondeterministic branching.

$$\begin{array}{rll} \mathrm{ert}\,[\,\mathrm{if}\ \star\ c_1\ \mathrm{else}\ c_2, \mathscr{D}](\mathbf{k}+f) & & \dagger \text{ Table 1 } \dagger \\ & = & \max\{\,\mathrm{ert}\,[c_1, \mathscr{D}](\mathbf{k}+f),\ \mathrm{ert}\,[c_2, \mathscr{D}](\mathbf{k}+f)\} \\ & & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\ & \sqsubseteq & \max\{\mathbf{k}+\ \mathrm{ert}\,[c_1, \mathscr{D}](f), \mathbf{k}+\ \mathrm{ert}\,[c_2, \mathscr{D}](f)\} \\ & & \dagger \text{ Algebra } \dagger \\ & = & \mathbf{k}+\ \max\{\,\mathrm{ert}\,[c_1, \mathscr{D}](f),\ \mathrm{ert}\,[c_2, \mathscr{D}](f)\} \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathbf{k}+\ \mathrm{ert}\,[\,\mathrm{if}\ \star\ c_1\ \mathrm{else}\ c_2, \mathscr{D}](f) \end{array}$$

Probabilistic branching.

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

Sequence.

$$\begin{aligned}
\mathsf{ert}\,[c_1;c_2,\mathscr{D}](\mathbf{k}+f) &\quad \dagger \text{ Table 1 } \dagger \\
&= \mathsf{ert}\,[c_1,\mathscr{D}](\mathsf{ert}\,[c_2,\mathscr{D}](\mathbf{k}+f)) \\
&\quad \dagger \text{ By I.H on } c_2 \dagger \\
&\sqsubseteq \mathsf{ert}\,[c_1,\mathscr{D}](\mathbf{k}+\mathsf{ert}\,[c_2,\mathscr{D}](f)) \\
&\quad \dagger \text{ By I.H on } c_1 \dagger \\
&\sqsubseteq \mathbf{k}+\mathsf{ert}\,[c_1,\mathscr{D}](\mathsf{ert}\,[c_2,\mathscr{D}](f)) \\
&\quad \dagger \text{ Table 1 } \dagger \\
&= \mathbf{k}+\mathsf{ert}\,[c_1;c_2,\mathscr{D}](f)
\end{aligned}$$

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

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

We first need to show that $F_{\mathbf{k}+f}(\mathbf{k}+\mathsf{lfp}\ F_f) \sqsubseteq \mathbf{k}+\mathsf{lfp}\ F_f$. Then following Park's Theorem[2] [81], we get $\mathsf{lfp}\ F_{\mathbf{k}+f} \sqsubseteq \mathbf{k}+\mathsf{lfp}\ F_f$.

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

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

$$\mathsf{ert}\,[\,\mathsf{call}\,{}^{\mathscr{D}}_{n}P](\mathbf{k}+f) \sqsubseteq \mathbf{k}+\mathsf{ert}\,[\,\mathsf{call}\,{}^{\mathscr{D}}_{n}P](f)$$

[2]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 $\mathsf{lfp}\ H \sqsubseteq d$ for all $d \in \mathscr{D}$.

On the other hand, by Theorem B.5, $\mathrm{ert}\,[\,\mathrm{call}\ P, \mathscr{D}] = \sup_n \mathrm{ert}\,[\,\mathrm{call}\,{}^{\mathscr{D}}_{n}P]$, where $\mathrm{call}\,{}^{\mathscr{D}}_{n}P$ is the n^{th}-inlining of procedure call, we have

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

A.5 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 $\mathrm{non-deterministic\ commands}$.

Skip.

$$\begin{aligned}
\mathrm{ert}\,[\,\mathrm{skip}\,, \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 } \dagger\\
&= f+g\\
&\quad \dagger \text{ Table 1 } \dagger\\
&= \mathrm{ert}\,[\,\mathrm{skip}\,, \mathscr{D}](f) + \mathrm{ert}\,[\,\mathrm{skip}\,, \mathscr{D}](g)
\end{aligned}$$

Abort.

$$\begin{aligned}
\mathrm{ert}\,[\,\mathrm{abort}\,, \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 and algebra } \dagger\\
&= \mathbf{0}\\
&= \mathrm{ert}\,[\,\mathrm{abort}\,, \mathscr{D}](f) + \mathrm{ert}\,[\,\mathrm{abort}\,, \mathscr{D}](g)
\end{aligned}$$

Assert.

$$\begin{aligned}
\mathrm{ert}\,[\,\mathrm{assert}\ e, \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 and algebra } \dagger\\
&= [\![e:\mathrm{true}]\!]\cdot(f+g) = [\![e:\mathrm{true}]\!]\cdot f + [\![e:\mathrm{true}]\!]\cdot g\\
&\quad \dagger \text{ Table 1 } \dagger\\
&= \mathrm{ert}\,[\,\mathrm{assert}\ e, \mathscr{D}](f) + \mathrm{ert}\,[\,\mathrm{assert}\ e, \mathscr{D}](g)
\end{aligned}$$

Weaken.

$$\begin{aligned}
\mathrm{ert}\,[\,\mathrm{weaken}\,, \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 } \dagger\\
&= f+g\\
&\quad \dagger \text{ Table 1 } \dagger\\
&= \mathrm{ert}\,[\,\mathrm{weaken}\,, \mathscr{D}](f) + \mathrm{ert}\,[\,\mathrm{weaken}\,, \mathscr{D}](g)
\end{aligned}$$

Tick.

$$\begin{aligned}
\mathrm{ert}\,[\,\mathrm{tick}\,(q), \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 } \dagger\\
&= \mathbf{q}+f+g\\
&\quad \dagger \text{ Algebra } \dagger\\
&\sqsubseteq \mathbf{q}+f+\mathbf{q}+g\\
&\quad \dagger \text{ Table 1 } \dagger\\
&= \mathrm{ert}\,[\,\mathrm{tick}\,(q), \mathscr{D}](f) + \mathrm{ert}\,[\,\mathrm{tick}\,(q), \mathscr{D}](g)
\end{aligned}$$

Assignment.

$$\begin{aligned}
\mathrm{ert}\,[x=e, \mathscr{D}](f+g) &\quad \dagger \text{ Table 1 } \dagger\\
&= f[e/x]+g[e/x]\\
&\quad \dagger \text{ Table 1 } \dagger\\
&= \mathrm{ert}\,[x=e, \mathscr{D}](f) + \mathrm{ert}\,[x=e, \mathscr{D}](g)
\end{aligned}$$

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

$$
\begin{array}{lll}
\mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](f+g) & & \dagger \text{ Table 1 } \dagger \\
& = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(f+g)(\sigma[e \text{ bop } v/x])) \\
& & \dagger \text{ Linearity of expectations } \dagger \\
& = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.f(\sigma[e \text{ bop } v/x])) + \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.g(\sigma[e \text{ bop } v/x])) \\
& & \dagger \text{ Table 1 } \dagger \\
& = & \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](f) + \mathsf{ert}\,[x = e \text{ bop } R, \mathscr{D}](g)
\end{array}
$$

If.

$$
\begin{array}{lll}
\mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f+g) & & \dagger \text{ Table 1 } \dagger \\
& = & [\![e : \mathsf{true}]\!]\cdot \mathsf{ert}\,[c_1, \mathscr{D}](f+g) + \\
& & [\![e : \mathsf{false}]\!]\cdot \mathsf{ert}\,[c_2, \mathscr{D}](f+g) \\
& & \dagger \text{ By I.H on } c_1 \text{ and } c_2 \dagger \\
& \sqsubseteq & [\![e : \mathsf{true}]\!]\cdot(\mathsf{ert}\,[c_1, \mathscr{D}](f) + \mathsf{ert}\,[c_1, \mathscr{D}](g)) + \\
& & [\![e : \mathsf{false}]\!]\cdot(\mathsf{ert}\,[c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_2, \mathscr{D}](g)) \\
& & \dagger \text{ Table 1 and algebra } \dagger \\
& = & \mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](f) + \mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](g)
\end{array}
$$

Probabilistic branching.

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

Sequence.

$$
\begin{array}{lll}
\mathsf{ert}\,[c_1; c_2, \mathscr{D}](f+g) & & \dagger \text{ Table 1 } \dagger \\
& = & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f+g)) \\
& & \dagger \text{ By I.H on } c_2 \dagger \\
& \sqsubseteq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_2, \mathscr{D}](g)) \\
& & \dagger \text{ By I.H on } c_1 \dagger \\
& \sqsubseteq & \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](f)) + \mathsf{ert}\,[c_1, \mathscr{D}](\mathsf{ert}\,[c_2, \mathscr{D}](g)) \\
& & \dagger \text{ Table 1 } \dagger \\
& = & \mathsf{ert}\,[c_1; c_2, \mathscr{D}](f) + \mathsf{ert}\,[c_1; c_2, \mathscr{D}](g)
\end{array}
$$

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

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

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 [81], we get $\mathsf{lfp}\ F_{\mathbf{k}+f} \sqsubseteq \mathsf{lfp}\ F_f + \mathsf{lfp}\ F_g$.

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

Procedure call. Because $\mathsf{call}\,{}^{\mathscr{D}}_{n}P$ (defined in Section B.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 B.5, $\mathsf{ert}\,[\,\mathsf{call}\,P, \mathscr{D}] = \sup_n \mathsf{ert}\,[\,\mathsf{call}\,{}^{\mathscr{D}}_{n}P]$, where $\mathsf{call}\,{}^{\mathscr{D}}_{n}P$ is the n^{th}-inlining of procedure call, we have

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

A.6 Scaling

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

Skip.

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

Abort.

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

Assert.

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

Weaken.

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

Tick.

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

Assignment.

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

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

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

If.

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

Nondeterministic branching.

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

Probabilistic branching.

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

Sequence.

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

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

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

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 $F_f^0 := \mathsf{id}$ 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

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

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

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

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

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

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

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

A.7 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.

$$
\begin{array}{rl}
\mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & \infty
\end{array}
$$

Assert.

$$
\begin{array}{rl}
\mathsf{ert}\,[\,\mathsf{assert}\ e, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & [\![e : \mathsf{true}]\!]\cdot\infty = \infty
\end{array}
$$

Weaken.

$$
\begin{array}{rl}
\mathsf{ert}\,[\,\mathsf{weaken}\,, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & \infty
\end{array}
$$

Tick.

$$
\begin{array}{rl}
\mathsf{ert}\,[\,\mathsf{tick}\,(q), \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & \mathbf{q} + \infty = \infty
\end{array}
$$

Assignment.

$$
\begin{array}{rl}
\mathsf{ert}\,[x = e, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & \infty[e/x] = \infty
\end{array}
$$

Sampling.

$$
\begin{array}{rl}
\mathsf{ert}\,[x = e\ \mathsf{bop}\ R, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.(\infty)(\sigma[e\ \mathsf{bop}\ v/x])) \\
= & \mathbb{E}_{\mu_R}(\infty) = \infty
\end{array}
$$

If.

$$
\begin{array}{rl}
\mathsf{ert}\,[\,\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](\infty) & \dagger\ \text{Table 1}\ \dagger \\
= & [\![e : \mathsf{true}]\!]\cdot\ \mathsf{ert}\,[c_1, \mathscr{D}](\infty) + \\
 & [\![e : \mathsf{false}]\!]\cdot\ \mathsf{ert}\,[c_2, \mathscr{D}](\infty) \\
 & \dagger\ \text{By I.H on } c_1 \text{ and } c_2\ \dagger \\
= & [\![e : \mathsf{true}]\!]\cdot\infty + [\![e : \mathsf{false}]\!]\cdot\infty \\
= & \infty
\end{array}
$$

Nondeterministic branching.

$$\begin{aligned}\mathsf{ert}\,[\,\mathsf{if}\ \star\ c_1\ \mathsf{else}\ c_2, \mathscr{D}](\infty) &\quad \dagger\ \text{Table 1}\ \dagger\\ &= \max\{\,\mathsf{ert}\,[c_1, \mathscr{D}](\infty),\ \mathsf{ert}\,[c_2, \mathscr{D}](\infty)\}\\ &\quad \dagger\ \text{By I.H on } c_1 \text{ and } c_2\ \dagger\\ &= \max\{\infty, \infty\}\\ &= \infty\end{aligned}$$

Probabilistic branching.

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

Sequence.

$$\begin{aligned}\mathsf{ert}\,[c_1; c_2, \mathscr{D}](\infty) &\quad \dagger\ \text{Table 1}\ \dagger\\ &= \mathsf{ert}\,[c_1, \mathscr{D}](\,\mathsf{ert}\,[c_2, \mathscr{D}](\infty))\\ &\quad \dagger\ \text{By I.H on } c_2\ \dagger\\ &= \mathsf{ert}\,[c_1, \mathscr{D}](\infty)\\ &\quad \dagger\ \text{By I.H on } c_1\ \dagger\\ &= \infty\end{aligned}$$

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

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

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

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

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

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

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

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

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

B BOUNDED LOOPS AND RECURSIVE PROCEDURE CALLS

B.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 B.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

$$\begin{aligned}
&\mathsf{ert}\,[\,\mathsf{while}\ e\ c, \mathscr{D}](f) && \dagger \text{ Table 1 } \dagger\\
&= && \mathsf{lfp}\ F_f\\
& && \dagger \text{ Definition of fixed point } \dagger\\
&= && F_f(\,\mathsf{lfp}\ F_f) = [\![e :\ \mathsf{true}\,]\!] \cdot \mathsf{ert}\,[c, \mathscr{D}](\,\mathsf{lfp}\ F_f) + [\![e :\ \mathsf{false}\,]\!] \cdot f\\
& && \dagger \text{ Table 1 } \dagger\\
&= && [\![e :\ \mathsf{true}\,]\!] \cdot \mathsf{ert}\,[c, \mathscr{D}](\,\mathsf{ert}\,[\,\mathsf{while}\ e\ c, \mathscr{D}](f)) + [\![e :\ \mathsf{false}\,]\!] \cdot f\\
& && \dagger \text{ Table 1 } \dagger\\
&= && [\![e :\ \mathsf{true}\,]\!] \cdot \mathsf{ert}\,[c; \mathsf{while}\ e\ c, \mathscr{D}](f) + [\![e :\ \mathsf{false}\,]\!] \cdot f\\
& && \dagger\ \ \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](f) = f\ \dagger\\
&= && [\![e :\ \mathsf{true}\,]\!] \cdot \mathsf{ert}\,[c; \mathsf{while}\ e\ c, \mathscr{D}](f) + [\![e :\ \mathsf{false}\,]\!] \cdot \mathsf{ert}\,[\,\mathsf{skip}\,, \mathscr{D}](f)\\
& && \dagger \text{ Table 1 } \dagger\\
&= && \mathsf{ert}\,[\,\mathsf{if}\ e\ \{c;\ \mathsf{while}\ e\ c\}\ \mathsf{else}\ \ \mathsf{skip}\,, \mathscr{D}](f)
\end{aligned}$$

□

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

$$\begin{aligned}
\mathsf{while}^0 e\ c &:= \mathsf{abort}\\
\mathsf{while}^{n+1} e\ c &:= \mathsf{if}\ e\ \{c;\ \mathsf{while}^n e\ c\}\ \mathsf{else}\ \ \mathsf{skip}
\end{aligned}$$

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

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

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

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

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

Let $C_n := \mathsf{while}^n e\ c$, we first prove that for every $n \in \mathbb{N}$, $\mathsf{ert}\,[C_n, \mathscr{D}](f) = F_f^n(\mathbf{0})$, where $F_f^0 := \mathsf{id}$ 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

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

- *Induction case.* Assume that $\mathsf{ert}\,[C_n, \mathscr{D}](f) = F_f^n(\mathbf{0})$, we reason as follows

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

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

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

□

B.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

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

Table 3. Characterization of procedure call.

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 $\mathsf{call}\ P$.

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

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

Proof. The proof is done by induction on the structure of c. If c has the form that is different from $\mathsf{call}\ P$, 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 $\mathsf{if}\ e\ c_1\ \mathsf{else}\ c_2$. For all $f \in \mathbb{T}$, we reason as follows.

$$\begin{aligned}
\mathrm{ert}\,[c]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(f) \quad & \dagger\ \text{Table 3}\ \dagger \\
= \quad & [\![e:\ \mathsf{true}\,]\!]\cdot \mathrm{ert}\,[c_1]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(f) + [\![e:\ \mathsf{false}\,]\!]\cdot \mathrm{ert}\,[c_2]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(f) \\
& \dagger\ \text{By I.H for } \mathrm{ert}\,[c_1]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]} \text{ and } \mathrm{ert}\,[c_2]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}\ \dagger \\
= \quad & [\![e:\ \mathsf{true}\,]\!]\cdot \mathrm{ert}\,[c_1,\mathscr{D}](f) + [\![e:\ \mathsf{false}\,]\!]\cdot \mathrm{ert}\,[c_2,\mathscr{D}](f) \\
& \dagger\ \text{Table 1}\ \dagger \\
= \quad & \mathrm{ert}\,[c,\mathscr{D}](f)
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{ert}\,[c]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(f) \quad & \dagger\ \text{Table 3}\ \dagger \\
= \quad & \mathrm{lfp}\, Y.([\![e:\ \mathsf{true}\,]\!]\cdot \mathrm{ert}\,[c_1]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(Y) + [\![e:\ \mathsf{false}\,]\!]\cdot f) \\
& \dagger\ \text{By I.H for } \mathrm{ert}\,[c_1]^{\sharp}_{\mathrm{ert}\,[c',\mathscr{D}]}(Y)\ \dagger \\
= \quad & \mathrm{lfp}\, Y.([\![e:\ \mathsf{true}\,]\!]\cdot \mathrm{ert}\,[c_1,\mathscr{D}](Y) + [\![e:\ \mathsf{false}\,]\!]\cdot f) \\
& \dagger\ \text{Table 1}\ \dagger \\
= \quad & \mathrm{ert}\,[c,\mathscr{D}](f)
\end{aligned}$$

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

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

□

B.3 Syntactic replacement of procedure call

Table 4 gives the formal inductive definition of the syntactic replacement of procedure calls $c[c'/\,\mathsf{call}\,P]$ on the structure of the command c. The following lemma says the property of the

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

Table 4. Syntactic replacement of procedure call.

replacement by a closed command w.r.t the expected cost transformer ert , where the declaration $\mathscr{D}(P) = c'$.

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

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

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 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.

$$\begin{aligned}
\text{ert}\,[c[c'/\,\text{call}\,P], \mathscr{D}] \quad & \dagger \text{ Table 4 } \dagger \\
= \quad & \text{ert}\,[\,\text{if } e\ c_1[c'/\,\text{call}\,P]\ \text{else}\ c_2[c'/\,\text{call}\,P], \mathscr{D}] \\
& \dagger \text{ Table 1 } \dagger \\
= \quad & [\![e :\ \text{true}\,]\!] \cdot \text{ert}\,[c_1[c'/\,\text{call}\,P], \mathscr{D}] + [\![e :\ \text{false}\,]\!] \cdot \text{ert}\,[c_2[c'/\,\text{call}\,P], \mathscr{D}] \\
& \dagger \text{ By I.H for ert } [c_1[c'/\,\text{call}\,P], \mathscr{D}] \text{ and ert } [c_2[c'/\,\text{call}\,P], \mathscr{D}] \ \dagger \\
= \quad & [\![e :\ \text{true}\,]\!] \cdot \text{ert}\,[c_1, \mathscr{D}] + [\![e :\ \text{false}\,]\!] \cdot \text{ert}\,[c_2, \mathscr{D}] \\
& \dagger \text{ Table 1 } \dagger \\
= \quad & \text{ert}\,[c, \mathscr{D}]
\end{aligned}$$

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

$$\begin{aligned}
\text{ert}\,[c[c'/\,\text{call}\,P], \mathscr{D}](f) \quad & \dagger \text{ Table 4 } \dagger \\
= \quad & \text{ert}\,[\,\text{while}\ e\ c_1[c'/\,\text{call}\,P]](f) \\
& \dagger \text{ Table 1 } \dagger \\
= \quad & \text{lfp}\,X.([\![e :\ \text{true}\,]\!] \cdot \text{ert}\,[c_1[c'/\,\text{call}\,P], \mathscr{D}](X) + [\![e :\ \text{false}\,]\!] \cdot f) \\
& \dagger \text{ By I.H for ert } [c_1[c'/\,\text{call}\,P], \mathscr{D}] \ \dagger \\
= \quad & \text{lfp}\,X.([\![e :\ \text{true}\,]\!] \cdot \text{ert}\,[c_1, \mathscr{D}](X) + [\![e :\ \text{false}\,]\!] \cdot f) \\
& \dagger \text{ Table 1 } \dagger \\
= \quad & \text{ert}\,[c, \mathscr{D}](f)
\end{aligned}$$

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

$$\begin{aligned}
\text{ert}\,[\,\text{call}\ P, \mathscr{D}](f) \quad & \dagger \text{ Theorem } B.5 \ \dagger \\
= \quad & \sup_n \text{ert}\,[\,\text{call}\,\overset{\mathscr{D}}{n}P](f) \\
& \dagger \ \text{ert}\,[\,\text{call}\,\overset{\mathscr{D}}{0}P](f) = \mathbf{0};\ \text{call}\,\overset{\mathscr{D}}{n+1}P = c' \ \dagger \\
= \quad & \sup_n \text{ert}\,[\,\text{call}\,\overset{\mathscr{D}}{n+1}P](f) = \sup_n \text{ert}\,[c', \mathscr{D}](f) \\
& \dagger \text{ The supremum of constant sequence } \dagger \\
= \quad & \text{ert}\,[c', \mathscr{D}](f) \\
& \dagger \ \text{call}\,P[c'/\,\text{call}\ P] = c' \ \dagger \\
= \quad & \text{ert}\,[\,\text{call}\ P[c'/\,\text{call}\ P], \mathscr{D}](f)
\end{aligned}$$

□

B.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 [74] 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 $\text{call}\,\overset{\mathscr{D}}{n}P$, is

defined as follows.

$$\mathsf{call}\,{}^{\mathscr{D}}_{0}P := \mathsf{abort}$$
$$\mathsf{call}\,{}^{\mathscr{D}}_{n+1}P := \mathscr{D}(P)[\,\mathsf{call}\,{}^{\mathscr{D}}_{n}P/\,\mathsf{call}\;P]$$

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 B.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

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

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

$$\sup_n \mathsf{ert}\,[\,\mathsf{call}\,{}^{\mathscr{D}}_{n}P] = \mathsf{lfp}\;X.(\,\mathsf{ert}\,[\mathscr{D}(P)]^{\sharp}_{X}) = \mathsf{ert}\,[\,\mathsf{call}\;P,\mathscr{D}]$$

□

C OMITTED PROOFS

C.1 Potential relax

Lemma C.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 C.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

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

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

$$\begin{array}{lll} (q'_1, \cdots, q'_n) & \dagger\ Q' = Q - F\vec{u} \text{ and matrix multiplication } \dagger \\ = & (q_1, \cdots, q_n) - (\sum_{k=1}^N f_1^k{\cdot}u_k, \cdots, \sum_{k=1}^N f_n^k{\cdot}u_k) \\ & \dagger \text{ Vector subtraction } \dagger \\ = & ((q_1 - \sum_{k=1}^N f_1^k{\cdot}u_k), \cdots, (q_n - \sum_{k=1}^N f_n^k{\cdot}u_k)) \end{array}$$

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

□

C.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

$$\begin{array}{lll} \text{ert}\,[\,\text{skip}\,,\mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) & \dagger \text{ Table 1 } \dagger \\ = & \mathscr{T}(\Gamma;Q)(\sigma) \leq \mathscr{T}(\Gamma;Q)(\sigma) \end{array}$$

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

$$\begin{array}{lll} \text{ert}\,[\,\text{abort}\,,\mathscr{D}](\mathscr{T}(\Gamma;Q))(\sigma) & \dagger \text{ Table 1 } \dagger \\ = & \mathbf{0}(\sigma) \leq \mathscr{T}(\Gamma;Q)(\sigma) \end{array}$$

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

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

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

$$\begin{array}{lll} \text{ert}\,[\,\text{assert}\ e,\mathscr{D}](\mathscr{T}(\Gamma \wedge e;Q))(\sigma) & \dagger \text{ Table 1 } \dagger \\ = & [\![e : \text{true}]\!]{\cdot}\mathscr{T}(\Gamma;Q)(\sigma) \\ \leq & \mathscr{T}(\Gamma;Q)(\sigma) \end{array}$$

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 $Q_2' \geq_{\Gamma_2'} Q_1'$, it holds that

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

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

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

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

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

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

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

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

$$\begin{array}{rll} \text{ert}\,[\,\text{tick}\,(q), \mathscr{D}](\mathscr{T}(\Gamma; Q - q))(\sigma) & & \dagger \text{ Table 1 } \dagger \\ & = & q + \mathscr{T}(\Gamma; Q - q)(\sigma) \\ & & \dagger \text{ Definition of translation function } \dagger \\ & = & q + \Phi_Q(\sigma) - q \leq \mathscr{T}(\Gamma; Q)(\sigma) \end{array}$$

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 $\sigma \not\models \Gamma[e/x]$,

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

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

$$\begin{array}{rll} \text{ert}\,[x = e, \mathscr{D}](\mathscr{T}(\Gamma; Q'))(\sigma) & & \dagger \text{ Table 1 } \dagger \\ & = & \mathscr{T}(\Gamma; Q')(\sigma[e/x]) \\ & & \dagger \text{ Definition of translation function } \dagger \\ & = & \max(\Gamma[e/x](\sigma), \Phi_{Q'}(\sigma[e/x])) = \Phi_{Q'}(\sigma[e/x]) \\ & & \dagger \text{ Definition of potential function and rule Q:Assign } \dagger \\ & = & \sum_{j \in X_{x=e}} q_j' \cdot b_j[e/x](\sigma) + \sum_{j \notin X_{x=e}} 0 \cdot b_j[e/x](\sigma) \\ & & \dagger \text{ Rule Q:Assign } \dagger \\ & = & \sum_{j \in X_{x=e}} q_j' \cdot \sum_i a_{i,j} \cdot b_i(\sigma) = \Phi_Q(\sigma) \\ & & \dagger \text{ Definition of translation function } \dagger \\ & = & \mathscr{T}(\Gamma[e/x]; Q) \end{array}$$

Sampling. Suppose c is of the form $x = e \text{ bop } R$, 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 $\sigma \not\models \Gamma$,

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

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

$$\begin{array}{rll} \mathscr{T}(\Gamma; Q_i)(\sigma) & & \dagger \text{ Definition of translation function } \dagger \\ & = & \Phi_{Q_i}(\sigma) \\ & & \dagger \text{ By I.H for assignment } \dagger \\ & \geq & \text{ert}\,[x = e \text{ bop } v_i, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) \\ & & \dagger \text{ Table 1 } \dagger \\ & = & \mathscr{T}(\Gamma'; Q')(\sigma[e \text{ bop } v_i/x]) \\ & & \dagger \text{ Definition of translation function } \dagger \\ & = & \max(\Gamma^i(\sigma), \Phi_{Q'}(\sigma[e \text{ bop } v_i/x])) = \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]) \end{array}$$

$$\begin{array}{rll} \text{ert}\,[x = e \text{ bop } R, \mathscr{D}](\mathscr{T}(\Gamma'; Q'))(\sigma) & & \dagger \text{ Table 1 } \dagger \\ & = & \lambda\sigma.\mathbb{E}_{\mu_R}(\lambda v.\mathscr{T}(\Gamma'; Q')(\sigma[e \text{ bop } v/x])) \\ & & \dagger \text{ Definition of expectation } \dagger \\ & = & \sum_i [\![\mu_R : v_i]\!] \cdot (\max(\Gamma^i(\sigma), \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]))) \\ & & \dagger \text{ Definition of translation function } \dagger \\ & = & \sum_i p_i \cdot \Phi_{Q'}(\sigma[e \text{ bop } v_i/x]) \\ & & \dagger \text{ Observation above } \dagger \\ & \leq & \sum_i p_i \cdot \Phi_{Q_i}(\sigma) \\ & & \dagger \text{ By rule Q:Sample } \dagger \\ & = & \Phi_Q(\sigma) = \mathscr{T}(\Gamma; Q)(\sigma) \end{array}$$

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathscr{T}(\Gamma;Q)(\sigma) && \dagger \text{ By I.H for } c_1 \dagger\\
&\geq && \mathrm{ert}\,[c_1,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma)\\
& && \dagger \text{ By I.H for } c_2 \dagger\\
&\geq && \mathrm{ert}\,[c_2,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma)\\
& && \dagger \text{ Algebra } \dagger\\
&\geq && \max\{\,\mathrm{ert}\,[c_1,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma),\ \mathrm{ert}\,[c_2,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma)\}\\
& && \dagger \text{ Table 1 } \dagger\\
&= && \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma)
\end{aligned}$$

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

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

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

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

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

$$\begin{aligned}
&\mathscr{T}(\Gamma;Q)(\sigma) && \dagger \text{ By I.H for } c_1 \dagger\\
&\geq && \mathrm{ert}\,[c_1,\mathscr{D}](\mathscr{T}(\Gamma';Q'))(\sigma)\\
& && \dagger \text{ By I.H for } c_2 \text{ and monotonicity of ert } \dagger\\
&\geq && \mathrm{ert}\,[c_1,\mathscr{D}](\,\mathrm{ert}\,[c_2,\mathscr{D}](\mathscr{T}(\Gamma'';Q'')))(\sigma)\\
& && \dagger \text{ Table 1 } \dagger\\
&= && \mathrm{ert}\,[c,\mathscr{D}](\mathscr{T}(\Gamma'';Q''))(\sigma)
\end{aligned}$$

Loop. Suppose c is of the form $\text{while } e\ c_1$, thus the automatic analysis ends with an application of the rule Q:Loop. For any program state σ, we consider the following characteristic function

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

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

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

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

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

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

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

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

$$\begin{array}{rrl} & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q))(\sigma) & \leq \mathscr{T}(\Gamma;Q)(\sigma) \\ & \dagger \text{ Definition of } \sqsubseteq \dagger & \\ \Leftrightarrow & F_{\mathscr{T}(\Gamma\wedge\neg e;Q)}(\mathscr{T}(\Gamma;Q)) & \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \dagger \text{ Park's Theorem}^3\ [81] \dagger & \\ \Rightarrow & \text{lfp}\ F_{\mathscr{T}(\Gamma\wedge\neg e;Q)} & \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \dagger \text{ Table 1} \dagger & \\ \Leftrightarrow & \text{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma \wedge \neg e;Q)) & \sqsubseteq \mathscr{T}(\Gamma;Q) \\ & \dagger \text{ Definition of } \sqsubseteq \dagger & \\ \Leftrightarrow & \text{ert}\,[c, \mathscr{D}](\mathscr{T}(\Gamma \wedge \neg e;Q'))(\sigma) & \leq \mathscr{T}(\Gamma;Q)(\sigma) \end{array}$$

Following Theorem B.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 $\forall n.\, \text{ert}\,[\,\text{while}^n\ e\ c_1, \mathscr{D}](\mathscr{T}(\Gamma \wedge \neg e;Q))(\sigma) \leq \mathscr{T}(\Gamma;Q)(\sigma)$ for all program states σ.

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 B.5, $\text{ert}\,[\,\text{call}\ P, \mathscr{D}] = \sup_n \text{ert}\,[\,\text{call}\,{}^{\mathscr{D}}_{n} P]$, where the n^{th}-inlining of procedure call $\text{call}\,{}^{\mathscr{D}}_{n} P$ is defined in Section B.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

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

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

[3] 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 $d \in \mathscr{D}$.

- *Base case.* It is intermediately satisfied because

$$\begin{aligned} & \mathsf{ert}\,[\,\mathsf{call}^{\mathscr{D}}_{0}\,P](\mathscr{T}(\Gamma';Q'+c))(\sigma) \\ & \qquad \dagger \text{ Definition of } \mathsf{call}^{\mathscr{D}}_{n}P \dagger \\ = \;& \mathsf{ert}\,[\,\mathsf{abort}\,](\mathscr{T}(\Gamma';Q'+c))(\sigma) \\ & \qquad \dagger \text{ Table 1 } \dagger \\ = \;& \mathbf{0}(\sigma) \le \mathscr{T}(\Gamma;Q+c)(\sigma) \end{aligned}$$

- *Induction case.* We reason as follows

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