Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs*

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo

Software Modeling and Verification Group, RWTH Aachen University
Ahornstraße 55, 52074 Aachen, Germany
`{benjamin.kaminski,katoen,matheja,federico.olmedo}@cs.rwth-aachen.de`

Abstract. This paper presents a wp–style calculus for obtaining bounds on the expected run–time of probabilistic programs. Its application includes determining the (possibly infinite) expected termination time of a probabilistic program and proving positive almost–sure termination—does a program terminate with probability one in finite expected time? We provide several proof rules for bounding the run–time of loops, and prove the soundness of the approach with respect to a simple operational model. We show that our approach is a conservative extension of Nielson's approach for reasoning about the run–time of deterministic programs. We analyze the expected run–time of some example programs including a one–dimensional random walk and the coupon collector problem.

Keywords: probabilistic programs · expected run–time · positive almost–sure termination · weakest precondition · program verification.

1 Introduction

Since the early days of computing, randomization has been an important tool for the construction of algorithms. It is typically used to convert a deterministic program with bad worst–case behavior into an efficient randomized algorithm that yields a correct output with high probability. The Rabin–Miller primality test, Freivalds' matrix multiplication, and the random pivot selection in Hoare's quicksort algorithm are prime examples. Randomized algorithms are conveniently described by probabilistic programs. On top of the usual language constructs, probabilistic programming languages offer the possibility of sampling values from a probability distribution. Sampling can be used in assignments as well as in Boolean guards.

The interest in probabilistic programs has recently been rapidly growing. This is mainly due to their wide applicability [10]. Probabilistic programs are

* This work was supported by the Excellence Initiative of the German federal and state government.

for instance used in security to describe cryptographic constructions and security experiments. In machine learning they are used to describe distribution functions that are analyzed using Bayesian inference. The sample program

$$C_{geo}\texttt{:}\quad b \mathrel{\texttt{:=}} 1;\ \texttt{while}\,(b = 1)\,\{b \mathrel{:\approx} {}^1\!/\!_2 \cdot \langle 0 \rangle + {}^1\!/\!_2 \cdot \langle 1 \rangle\}$$

for instance flips a fair coin until observing the first heads (i.e. 0). It describes a geometric distribution with parameter $^1/_2$.

The run–time of probabilistic programs is affected by the outcome of their coin tosses. Technically speaking, the run–time is a random variable, i.e. it is t_1 with probability p_1, t_2 with probability p_2 and so on. An important measure that we consider over probabilistic programs is then their *average* or *expected* run–time (over all inputs). Reasoning about the expected run–time of probabilistic programs is surprisingly subtle and full of nuances. In classical sequential programs, a single diverging program run yields the program to have an infinite run–time. This is not true for probabilistic programs. They may admit arbitrarily long runs while having a finite expected run–time. The program C_{geo}, for instance, does admit arbitrarily long runs as for any n, the probability of not seeing a heads in the first n trials is always positive. The expected run–time of C_{geo} is, however, finite.

In the classical setting, programs with finite run–times can be sequentially composed yielding a new program again with finite run–time. For probabilistic programs this does not hold in general. Consider the pair of programs

$$\begin{aligned} &C_1\texttt{:}\quad x \mathrel{\texttt{:=}} 1;\ b \mathrel{\texttt{:=}} 1;\ \texttt{while}\,(b = 1)\,\{b \mathrel{:\approx} {}^1\!/\!_2 \cdot \langle 0 \rangle + {}^1\!/\!_2 \cdot \langle 1 \rangle;\ x \mathrel{\texttt{:=}} 2x\} \quad \text{and} \\ &C_2\texttt{:}\quad \texttt{while}\,(x > 0)\,\{x \mathrel{\texttt{:=}} x - 1\}\ . \end{aligned}$$

The loop in C_1 terminates on average in two iterations; it thus has a finite expected run–time. From any initial state in which x is non–negative, C_2 makes x iterations, and thus its expected run–time is finite, too. However, the program $C_1; C_2$ has an *infinite* expected run–time—even though it almost–surely terminates, i.e. it terminates with probability one. Other subtleties can occur as program run–times are very sensitive to variations in the probabilities occurring in the program.

Bounds on the expected run–time of randomized algorithms are typically obtained using a detailed analysis exploiting classical probability theory (on expectations or martingales) [9,22]. This paper presents an alternative approach, based on formal program development and verification techniques. We propose a wp–style calculus à la Dijkstra for obtaining bounds on the expected run–time of probabilistic programs. The core of our calculus is the transformer ert, a quantitative variant of Dijkstra's wp–transformer. For a program C, $\mathsf{ert}\,[C]\,(f)\,(\sigma)$ gives the expected run–time of C started in initial state σ under the assumption that f captures the run–time of the computation following C. In particular, $\mathsf{ert}\,[C]\,(\mathbf{0})\,(\sigma)$ gives the expected run–time of program C on input σ (where $\mathbf{0}$ is the constantly zero run–time). Transformer ert is defined inductively on the program structure. We prove that our transformer conservatively extends Nielson's approach [23] for reasoning about the run–time of deterministic programs.

In addition we show that $\mathsf{ert}\left[C\right]\left(f\right)\left(\sigma\right)$ corresponds to the expected run–time in a simple operational model for our probabilistic programs based on Markov Decision Processes (MDPs). The main contribution is a set of proof rules for obtaining (upper and lower) bounds on the expected run–time of loops. We apply our approach for analyzing the expected run–time of some example programs including a one–dimensional random walk and the coupon collector problem [20].

We finally point out that our technique enables determining the (possibly infinite) expected time until termination of a probabilistic program and proving (universal) *positive almost–sure termination*—does a program terminate with probability one in finite expected time (on all inputs)? It has been recently shown [16] that the universal positive almost–sure termination problem is Π_3^0–complete, and thus strictly harder to solve than the universal halting problem for deterministic programs. To the best of our knowledge, the formal verification framework in this paper is the first one that is proved sound and can handle both positive almost–sure termination and infinite expected run–times.

Related work. Several works apply wp–style– or Floyd–Hoare–style reasoning to study quantitative aspects of classical programs. Nielson [23,24] provides a Hoare logic for determining upper bounds on the run–time of deterministic programs. Our approach applied to such programs yields the tightest upper bound on the run–time that can be derived using Nielson's approach. Arthan *et al.* [1] provide a general framework for sound and complete Hoare–style logics, and show that an instance of their theory can be used to obtain upper bounds on the run–time of while programs. Hickey and Cohen [13] automate the average–case analysis of deterministic programs by generating a system of recurrence equations derived from a program whose efficiency is to be analyzed. They build on top of Kozen's seminal work [18] on semantics of probabilistic programs. Berghammer and Müller–Olm [3] show how Hoare–style reasoning can be extended to obtain bounds on the closeness of results obtained using approximate algorithms to the optimal solution. Deriving space and time consumption of deterministic programs has also been considered by Hehner [11]. Formal reasoning about probabilistic programs goes back to Kozen [18], and has been developed further by Hehner [12] and McIver and Morgan [19]. The work by Celiku and McIver [5] is perhaps the closest to our paper. They provide a wp–calculus for obtaining performance properties of probabilistic programs, including upper bounds on expected run–times. Their focus is on refinement. They do neither provide a soundness result of their approach nor consider lower bounds. We believe that our transformer is simpler to work with in practice, too. Monniaux [21] exploits abstract interpretation to automatically prove the probabilistic termination of programs using exponential bounds on the tail of the distribution. His analysis can be used to prove the soundness of experimental statistical methods to determine the average run–time of probabilistic programs. Brazdil *et al.* [4] study the run–time of probabilistic programs with unbounded recursion by considering probabilistic pushdown automata (pPDAs). They show (using martingale theory) that for every pPDA the probability of performing a long run decreases exponentially (polynomially) in the length of the run, iff the pPDA has a finite (infinite) expected runtime. As opposed to our program verification technique, [4] considers reasoning at the operational level. Fioriti and Hermanns [8] recently proposed a typing scheme for deciding almost-sure termination. They showed, amongst others, that if a program is well-typed, then it almost surely terminates. This result does not cover positive almost-sure-termination.

Organization of the paper. Section 2 defines our probabilistic programming language. Section 3 presents the transformer ert and studies its elementary properties such as continuity. Section 4 shows that the ert transformer coincides with the expected run–time in an MDP that acts as operational model of our programs. Section 5 presents two sets of proof rules for obtaining upper and lower bounds on the expected run–time of loops. In Section 6, we show that the ert transformer is a conservative extension of Nielson's approach for obtaining upper bounds on deterministic programs. Section 7 discusses two case studies in detail. Section 8 concludes the paper.

The proofs of the main facts are included in the body of the paper. The remaining proofs and the calculations omitted in Section 7 are included in an extended version of the paper [17].

2 A Probabilistic Programming Language

In this section we present the probabilistic programming language used throughout this paper, together with its run–time model. To model probabilistic programs we employ a standard imperative language à la Dijkstra's Guarded Command Language [7] with two distinguished features: we allow distribution expressions in assignments and guards to be probabilistic. For instance, we allow for probabilistic assignments like

$$y :\approx \texttt{Unif}[1 \ldots x]$$

which endows variable y with a uniform distribution in the interval $[1 \ldots x]$. We allow also for a program like

$$x := 0;\ \texttt{while}\ \big(p \cdot \langle \mathsf{true} \rangle + (1{-}p) \cdot \langle \mathsf{false} \rangle\big)\ \{x := x + 1\}$$

which uses a probabilistic loop guard to simulate a geometric distribution with success probability p, i.e. the loop guard evaluates to true with probability p and to false with the remaining probability $1{-}p$.

Formally, the set of *probabilistic programs* pProgs is given by the grammar

C ::=	empty	empty program
	$\mid$ skip	effectless operation
	$\mid$ halt	immediate termination
	$\mid$ $x :\approx \mu$	probabilistic assignment
	$\mid$ $C; C$	sequential composition
	$\mid$ $\{C\} \,\Box\, \{C\}$	non–deterministic choice
	$\mid$ if (ξ) $\{C\}$ else $\{C\}$	probabilistic conditional
	$\mid$ while (ξ) $\{C\}$	probabilistic while loop

Here x represents a *program variable* in Var, μ a *distribution expression* in DExp, and ξ a distribution expression over the truth values, i.e. a *probabilistic guard*, in DExp. We assume distribution expressions in DExp to represent discrete probability distributions with a (possibly *infinite*) support of total probability mass 1. We use $p_1 \cdot \langle a_1 \rangle + \cdots + p_n \cdot \langle a_n \rangle$ to denote the distribution expression that assigns probability p_i to a_i. For instance, the distribution expression $1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle$ represents the toss of a fair coin. Deterministic expressions over program variables such as $x - y$ or $x - y > 8$ are special instances of distribution expressions—they are understood as Dirac probability distributions[1].

To describe the different language constructs we first present some preliminaries. A *program state* σ is a mapping from program variables to values in Val. Let $\Sigma \triangleq \{\sigma \mid \sigma\colon \mathsf{Var} \to \mathsf{Val}\}$ be the set of program states. We assume an interpretation function $[\![\,\cdot\,]\!]\colon \mathsf{DExp} \to (\Sigma \to \mathcal{D}(\mathsf{Val}))$ for distribution expressions, $\mathcal{D}(\mathsf{Val})$ being the set of discrete probability distributions over Val. For $\mu \in \mathsf{DExp}$, $[\![\mu]\!]$ maps each program state to a probability distribution of values. We use $[\![\mu\colon v]\!]$ as a shorthand for the function mapping each program state σ to the probability that distribution $[\![\mu]\!](\sigma)$ assigns to value v, i.e. $[\![\mu\colon v]\!](\sigma) \triangleq \mathsf{Pr}_{[\![\mu]\!](\sigma)}(v)$, where Pr denotes the probability operator on distributions over values.

We now present the effects of pProgs programs and the run–time model that we adopt for them. `empty` has no effect and its execution consumes no time. `skip` has also no effect but consumes, in contrast to `empty`, one unit of time. `halt` aborts any further program execution and consumes no time. $x :\approx \mu$ is a probabilistic assignment that samples a value from $[\![\mu]\!]$ and assigns it to variable x; the sampling and assignment consume (altogether) one unit of time. $C_1; C_2$ is the sequential composition of programs C_1 and C_2. $\{C_1\} \,\Box\, \{C_2\}$ is a non–deterministic choice between programs C_1 and C_2; we take a demonic view where we assume that out of C_1 and C_2 we execute the program with the greatest run–time. $\mathtt{if}\,(\xi)\,\{C_1\}\,\mathtt{else}\,\{C_2\}$ is a probabilistic conditional branching: with probability $[\![\xi\colon \mathsf{true}]\!]$ program C_1 is executed, whereas with probability $[\![\xi\colon \mathsf{false}]\!] = 1 - [\![\xi\colon \mathsf{true}]\!]$ program C_2 is executed; evaluating (or more rigorously, sampling a value from) the probabilistic guard requires an additional unit of time. $\mathtt{while}\ (\xi)\ \{C\}$ is a probabilistic while loop: with probability $[\![\xi\colon \mathsf{true}]\!]$ the loop body C is executed followed by a recursive execution of the loop, whereas with probability $[\![\xi\colon \mathsf{false}]\!]$ the loop terminates; as for conditionals, each evaluation of the guard consumes one unit of time.

Example 1 (Race between tortoise and hare). The probabilistic program

```
$h :\approx 0$; $t :\approx 30$;
while ($h \leq t$) {
    if ($1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle$) {$h :\approx h + \mathtt{Unif}[0 \ldots 10]$}
    else {empty};
    $t :\approx t + 1$
} ,
```

[1] A Dirac distribution assigns the total probability mass, i.e. 1, to a single point.

adopted from [6], illustrates the use of the programming language. It models a race between a hare and a tortoise (variables h and t represent their respective positions). The tortoise starts with a lead of 30 and in each step advances one step forward. The hare with probability $1/2$ advances a random number of steps between 0 and 10 (governed by a uniform distribution) and with the remaining probability remains still. The race ends when the hare passes the tortoise. $\triangle$

We conclude this section by fixing some notational conventions. To keep our program notation consistent with standard usage, we use the standard symbol `:=` instead of $:\approx$ for assignments whenever μ represents a Dirac distribution given by a deterministic expressions over program variables. For instance, in the program in Example 1 we write t `:=` $t+1$ instead of $t \;:\approx\; t+1$. Likewise, when ξ is a probabilistic guard given as a deterministic Boolean expression over program variables, we use $[\![\xi]\!]$ to denote $[\![\xi\colon \mathsf{true}]\!]$ and $[\![\neg\xi]\!]$ to denote $[\![\xi\colon \mathsf{false}]\!]$. For instance, we write $[\![b=0]\!]$ instead of $[\![b=0\colon \mathsf{true}]\!]$.

3 A Calculus of Expected Run–Times

Our goal is to associate to any program C a function that maps each state σ to the average or expected run–time of C started in initial state σ. We use the functional space of *run–times*

$$\mathbb{T} \;\triangleq\; \left\{f \;\middle|\; f\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty}\right\}$$

to model such functions. Here, $\mathbb{R}_{\geq 0}^{\infty}$ represents the set of non–negative real values extended with ∞. We consider run–times as a mapping from program states to real numbers (or ∞) as the expected run–time of a program may depend on the initial program state.

We express the run–time of programs using a continuation–passing style by means of the transformer

$$\mathsf{ert}[\,\cdot\,]\colon \mathsf{pProgs} \to (\mathbb{T} \to \mathbb{T}) \ .$$

Concretely, $\mathsf{ert}\left[C\right](f)(\sigma)$ gives the expected run–time of program C from state σ assuming that f captures the run–time of the computation that follows C. Function f is usually referred to as *continuation* and can be thought of as being evaluated in the final states that are reached upon termination of C. Observe that, in particular, if we set f to the constantly zero run–time, $\mathsf{ert}\left[C\right](\mathbf{0})(\sigma)$ gives the expected run–time of program C on input σ.

The transformer `ert` is defined by induction on the structure of C following the rules in Table 1. The rules are defined so as to correspond to the run–time model introduced in Section 2. That is, $\mathsf{ert}\left[C\right](\mathbf{0})$ captures the expected number of assignments, guard evaluations and `skip` statements. Most rules in Table 1 are self–explanatory. $\mathsf{ert}[$`empty`$]$ behaves as the identity since `empty` does not modify the program state and its execution consumes no time. On the other hand, $\mathsf{ert}[$`skip`$]$ adds one unit of time since this is the time required by the execution of `skip`. $\mathsf{ert}[$`halt`$]$ yields always the constant run–time $\mathbf{0}$ since `halt`

C	$\mathsf{ert}\,[C]\,(f)$
`empty`	f
`skip`	$\mathbf{1} + f$
`halt`	$\mathbf{0}$
$x :\approx \mu$	$\mathbf{1} + \lambda\sigma\text{.}\ \mathsf{E}_{[\![\mu]\!](\sigma)}\,(\lambda v.\, f\,[x/v]\,(\sigma))$
$C_1\,;\,C_2$	$\mathsf{ert}\,[C_1]\,(\mathsf{ert}\,[C_2]\,(f))$
$\{C_1\} \,\Box\, \{C_2\}$	$\max\{\mathsf{ert}\,[C_1]\,(f)\,,\,\mathsf{ert}\,[C_2]\,(f)\}$
`if` $(\xi)\ \{C_1\}$ `else` $\{C_2\}$	$\mathbf{1} + [\![\xi : \mathsf{true}]\!] \cdot \mathsf{ert}\,[C_1]\,(f) + [\![\xi : \mathsf{false}]\!] \cdot \mathsf{ert}\,[C_2]\,(f)$
`while` $(\xi)\ \{C'\}$	$\mathsf{lfp}\,X\text{.}\ \mathbf{1} + [\![\xi : \mathsf{false}]\!] \cdot f + [\![\xi : \mathsf{true}]\!] \cdot \mathsf{ert}\,[C']\,(X)$

Table 1. Rules for defining the expected run–time transformer ert. $\mathbf{1}$ is the constant run–time $\lambda\sigma.1$. $\mathsf{E}_\eta\,(h) \triangleq \sum_v \mathsf{Pr}_\eta(v) \cdot h(v)$ represents the expected value of (random variable) h w.r.t. distribution η. For $\sigma \in \Sigma$, $f\,[x/v]\,(\sigma) \triangleq f(\sigma\,[x/v])$, where $\sigma\,[x/v]$ is the state obtained by updating in σ the value of x to v. $\max\{f_1, f_2\} \triangleq \lambda\sigma.\max\{f_1(\sigma), f_2(\sigma)\}$ represents the point–wise lifting of the max operator over $\mathbb{R}^{\infty}_{\geq 0}$ to the function space of run–times. $\mathsf{lfp}\,X\text{.}\ F(X)$ represents the least fixed point of the transformer $F\colon \mathbb{T} \to \mathbb{T}$.

aborts any subsequent program execution (making their run–time irrelevant) and consumes no time. The definition of ert on random assignments is more involved: $\mathsf{ert}\,[x :\approx \mu]\,(f)\,(\sigma) = 1 + \sum_v \mathsf{Pr}_{[\![\mu]\!](\sigma)}(v) \cdot f(\sigma\,[x/v])$ is obtained by adding one unit of time (due to the distribution sampling and assignment of the value sampled) to the sum of the run–time of each possible subsequent execution, weighted according to their probabilities. $\mathsf{ert}[C_1\,;\,C_2]$ applies $\mathsf{ert}[C_1]$ to the expected run–time obtained from the application of $\mathsf{ert}[C_2]$. $\mathsf{ert}[\{C_1\} \,\Box\, \{C_2\}]$ returns the maximum between the run–time of the two branches. $\mathsf{ert}[$`if` $(\xi)\ \{C_1\}$ `else` $\{C_2\}]$ adds one unit of time (on account of the guard evaluation) to the weighted sum of the run–time of the two branches. Lastly, the ert of loops is given as the least fixed point of a run–time transformer defined in terms of the run–time of the loop body.

Remark. We stress that the above run–time model is a design decision for the sake of concreteness. All our developments can easily be adapted to capture alternative models. These include, for instance, the model where only the number of assignments in a program run or the model where only the number of loop iterations are of relevance. We can also capture more fine–grained models, where for instance the run–time of an assignment depends on the *size* of the distribution expression being sampled.

Example 2 (Truncated geometric distribution). To illustrate the effects of the ert transformer consider the program in Figure 1. It can be viewed as modeling a truncated geometric distribution: we repeatedly flip a fair coin until observing

```
$C_{trunc}$:  if $\big(1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle\big)$ $\{succ := \mathsf{true}\}$ else {
                  if $\big(1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle\big)$ $\{succ := \mathsf{true}\}$
                  else $\{succ := \mathsf{false}\}$
              }
```

Fig. 1. Program modeling a truncated geometric distribution

the first heads or completing the second unsuccessful trial. The calculation of the expected run–time $\mathsf{ert}\,[C_{trunc}]\,(\mathbf{0})$ of program C_{trunc} goes as follows:

$$\begin{aligned}
&\mathsf{ert}\,[C_{trunc}]\,(\mathbf{0})\\
&\quad= \mathbf{1} + \tfrac{1}{2} \cdot \mathsf{ert}\,[succ := \mathsf{true}]\,(\mathbf{0})\\
&\qquad\quad + \tfrac{1}{2} \cdot \mathsf{ert}\,[\mathtt{if}\ (\ldots)\ \{succ := \mathsf{true}\}\ \mathtt{else}\ \{succ := \mathsf{false}\}]\,(\mathbf{0})\\
&\quad= \mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1} + \tfrac{1}{2} \cdot \Big(\mathbf{1} + \tfrac{1}{2} \cdot \mathsf{ert}\,[succ := \mathsf{true}]\,(\mathbf{0}) + \tfrac{1}{2} \cdot \mathsf{ert}\,[succ := \mathsf{false}]\,(\mathbf{0})\Big)\\
&\quad= \mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1} + \tfrac{1}{2} \cdot \Big(\mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1}\Big) \;=\; \tfrac{\mathbf{5}}{\mathbf{2}}
\end{aligned}$$

Therefore, the execution of C_{trunc} takes, on average, 2.5 units of time. △

Note that the calculation of the expected run–time in the above example is straightforward as the program at hand is loop–free. Computing the run–time of loops requires the calculation of least fixed points, which is generally not feasible in practice. In Section 5, we present invariant–based proof rules for reasoning about the run–time of loops.

The ert transformer enjoys several algebraic properties. To formally state these properties we make use of the point–wise order relation "$\preceq$" between run–times: given $f, g \in \mathbb{T}$, $f \preceq g$ iff $f(\sigma) \leq g(\sigma)$ for all states $\sigma \in \Sigma$.

Theorem 1 (Basic properties of the ert transformer). *For any program $C \in \mathsf{pProgs}$, any constant run–time $\mathbf{k} = \lambda\sigma.k$ for $k \in \mathbb{R}_{\geq 0}$, any constant $r \in \mathbb{R}_{\geq 0}$, and any two run–times $f, g \in \mathbb{T}$ the following properties hold:*

Monotonicity:	$f \preceq g \implies \mathsf{ert}\,[C]\,(f) \preceq \mathsf{ert}\,[C]\,(g)$;
Propagation of constants:	$\mathsf{ert}\,[C]\,(\mathbf{k} + f) = \mathbf{k} + \mathsf{ert}\,[C]\,(f)$ *provided C is* `halt`*–free;*
Preservation of ∞*:*	$\mathsf{ert}\,[C]\,(\infty) = \infty$ *provided C is* `halt`*–free;*
Sub–additivity:	$\mathsf{ert}\,[C]\,(f + g) \preceq \mathsf{ert}\,[C]\,(f) + \mathsf{ert}\,[C]\,(g)$; *provided C is fully probabilistic*[2]*;*
Scaling:	$\mathsf{ert}\,[C]\,(r \cdot f) \succeq \min\{1, r\} \cdot \mathsf{ert}\,[C]\,(f)$; $\mathsf{ert}\,[C]\,(r \cdot f) \preceq \max\{1, r\} \cdot \mathsf{ert}\,[C]\,(f)$.

[2] A program is called *fully probabilistic* if it contains no non–deterministic choices.

Proof. Monotonicity follows from continuity (see Lemma 1 below). The remaining proofs proceed by induction on the program structure; see [17] for details. □

We conclude this section with a technical remark regarding the well–definedness of the ert transformer. To guarantee that ert is well–defined, we must show the existence of the least fixed points used to define the run–time of loops. To this end, we use a standard denotational semantics argument (see e.g. [26, Ch. 5]): First we endow the set of run–times $\mathbb{T}$ with the structure of an ω–complete partial order (ω–cpo) with bottom element. Then we use a continuity argument to conclude the existence of such fixed points.

Recall that $\preceq$ denotes the point–wise comparison between run–times. It easily follows that $(\mathbb{T}, \preceq)$ defines an ω–cpo with bottom element $\mathbf{0} = \lambda\sigma.0$ where the supremum of an ω–chain $f_1 \preceq f_2 \preceq \cdots$ in $\mathbb{T}$ is also given point–wise, i.e. as $\sup_n f_n \triangleq \lambda\sigma.\ \sup_n f_n(\sigma)$. Now we are in a position to establish the continuity of the ert transformer:

Lemma 1 (Continuity of the ert transformer). *For every program C and every ω–chain of run–times $f_1 \preceq f_2 \preceq \cdots$,*

$$\mathsf{ert}\,[C]\,(\sup_n f_n) \;=\; \sup_n \mathsf{ert}\,[C]\,(f_n) \ .$$

Proof. By induction on the program structure; see [17] for details. □

Lemma 1 implies that for each program $C \in \mathsf{pProgs}$, guard $\xi \in \mathsf{DExp}$, and run–time $f \in \mathbb{T}$, function $F_f(X) = \mathbf{1} + [\![\xi : \mathsf{false}]\!] \cdot f + [\![\xi : \mathsf{true}]\!] \cdot \mathsf{ert}\,[C]\,(X)$ is also continuous. The Kleene Fixed Point Theorem then ensures that the least fixed point $\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) = \mathsf{lfp}\,F_f$ exists and the expected run–time of loops is thus well-defined.

Finally, as the aforementioned function F_f is frequently used in the remainder of the paper, we define:

Definition 1 (Characteristic functional of a loop). *Given program $C \in \mathsf{pProgs}$, probabilistic guard $\xi \in \mathsf{DExp}$, and run–time $f \in \mathbb{T}$, we call*

$$F_f^{\langle \xi, C\rangle} \colon \mathbb{T} \to \mathbb{T},\ X \mapsto \mathbf{1} + [\![\xi : \mathsf{false}]\!] \cdot f + [\![\xi : \mathsf{true}]\!] \cdot \mathsf{ert}\,[C]\,(X)$$

the characteristic functional *of loop $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f.*

When C and ξ are understood from the context, we usually omit them and simply write F_f for the characteristic functional associated to $\mathtt{while}\,(\xi)\,\{C\}$ with respect to run–time f. Observe that under this definition, the ert of loops can be recast as

$$\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) \;=\; \mathsf{lfp}\,F_f^{\langle \xi, C\rangle} \ .$$

This concludes our presentation of the ert transformer. In the next section we validate the transformer's definition by showing a soundness result with respect to an operational model of programs.

4 An Operational Model for Expected Run–Times

We prove the soundness of the expected run–time transformer with respect to a simple operational model for our probabilistic programs. This model will be given in terms of a Markov Decision Process (MDP, for short) whose collected reward corresponds to the run–time. We first briefly recall all necessary notions. A more detailed treatment can be found in [2, Ch. 10]. A *Markov Decision Process* is a tuple $\mathfrak{M} = (\mathcal{S},\, \mathit{Act},\, \mathbf{P},\, s_0,\, \mathit{rew})$ where $\mathcal{S}$ is a countable set of states, Act is a (finite) set of actions, $\mathbf{P}\colon \mathcal{S} \times \mathit{Act} \times \mathcal{S} \to [0,1]$ is the transition probability function such that for all states $s \in \mathcal{S}$ and actions $\alpha \in \mathit{Act}$,

$$\sum_{s' \in \mathcal{S}} \mathbf{P}(s, \alpha, s') \in \{0, 1\} \ ,$$

$s_0 \in \mathcal{S}$ is the initial state, and $\mathit{rew}\colon \mathcal{S} \to \mathbb{R}_{\geq 0}$ is a reward function. Instead of $\mathbf{P}(s, \alpha, s') = p$, we usually write $s \xrightarrow{\alpha} s' \vdash p$. An MDP $\mathfrak{M}$ is a *Markov chain* if no non–deterministic choice is possible, i.e. for each pair of states $s, s' \in \mathcal{S}$ there exists exactly one $\alpha \in \mathit{Act}$ with $\mathbf{P}(s, \alpha, s') \neq 0$.

A *scheduler* for $\mathfrak{M}$ is a mapping $\mathfrak{S}\colon \mathcal{S}^+ \to \mathit{Act}$, where $\mathcal{S}^+$ denotes the set of non–empty finite sequences of states. Intuitively, a scheduler resolves the non–determinism of an MDP by selecting an action for each possible sequence of states that has been visited so far. Hence, a scheduler $\mathfrak{S}$ induces a Markov chain which is denoted by $\mathfrak{M}_{\mathfrak{S}}$. In order to define the expected reward of an MDP, we first consider the reward collected along a path. Let $\mathrm{Paths}^{\mathfrak{M}_{\mathfrak{S}}}$ ($\mathrm{Paths}^{\mathfrak{M}}_{\mathit{fin}}$) denote the set of all (finite) paths π ($\hat{\pi}$) in $\mathfrak{M}_{\mathfrak{S}}$. Analogously, let $\mathrm{Paths}^{\mathfrak{M}_{\mathfrak{S}}}(s)$ and $\mathrm{Paths}^{\mathfrak{M}_{\mathfrak{S}}}_{\mathit{fin}}(s)$ denote the set of all infinite and finite paths in $\mathfrak{M}_{\mathfrak{S}}$ starting in state $s \in \mathcal{S}$, respectively. For a finite path $\hat{\pi} = s_0 \ldots s_n$, the *cumulative reward* of $\hat{\pi}$ is defined as

$$\mathit{rew}(\hat{\pi}) \ \triangleq\ \sum_{k=0}^{n-1} \mathit{rew}(s_k) \ .$$

For an infinite path π, the cumulative reward of reaching a non–empty set of target states $T \subseteq \mathcal{S}$, is defined as $\mathit{rew}(\pi, \Diamond T) \triangleq \mathit{rew}(\pi(0) \ldots \pi(n))$ if there exists an n such that $\pi(n) \in T$ and $\pi(i) \notin T$ for $0 \leq i < n$ and $\mathit{rew}(\pi, \Diamond T) \triangleq \infty$ otherwise. Moreover, we write $\mathit{\Pi}(s, T)$ to denote the set of all finite paths $\hat{\pi} \in \mathrm{Paths}^{\mathfrak{M}_{\mathfrak{S}}}_{\mathit{fin}}(s)$, $s \in \mathcal{S}$, with $\hat{\pi}(n) \in T$ for some $n \in \mathbb{N}$ and $\hat{\pi}(i) \notin T$ for $0 \leq i < n$. The probability of a finite path $\hat{\pi}$ is

$$\mathrm{Pr}^{\mathfrak{M}_{\mathfrak{S}}}\{\hat{\pi}\} \ \triangleq\ \prod_{k=0}^{|\hat{\pi}|-1} \mathbf{P}(s_k, \mathfrak{S}(s_1, \ldots, s_k), s_{k+1}) \ .$$

The *expected reward* that an MDP $\mathfrak{M}$ eventually reaches a non–empty set of states $T \subseteq \mathcal{S}$ from a state $s \in \mathcal{S}$ is defined as follows. If

$$\inf_{\mathfrak{S}} \mathrm{Pr}^{\mathfrak{M}_{\mathfrak{S}}}\{s \models \Diamond T\} \ =\ \inf_{\mathfrak{S}} \sum_{\hat{\pi} \in \mathit{\Pi}(s,T)} \mathrm{Pr}^{\mathfrak{M}_{\mathfrak{S}}}\{\hat{\pi}\} \ <\ 1$$

$$\frac{}{\langle \downarrow,\, \sigma\rangle \xrightarrow{\tau} \langle\, \mathit{sink}\,\rangle \vdash 1}\ [\text{terminated}] \qquad \frac{}{\langle\, \mathit{sink}\,\rangle \xrightarrow{\tau} \langle\, \mathit{sink}\,\rangle \vdash 1}\ [\text{sink}]$$

$$\frac{}{\langle \texttt{empty},\, \sigma\rangle \xrightarrow{\tau} \langle \downarrow,\, \sigma\rangle \vdash 1}\ [\text{empty}] \qquad \frac{}{\langle \texttt{skip},\, \sigma\rangle \xrightarrow{\tau} \langle \downarrow,\, \sigma\rangle \vdash 1}\ [\text{skip}]$$

$$\frac{}{\langle \texttt{halt},\, \sigma\rangle \xrightarrow{\tau} \langle\, \mathit{sink}\,\rangle \vdash 1}\ [\text{halt}] \qquad \frac{[\![\mu\colon v]\!](\sigma) = p > 0}{\langle x :\approx \mu,\, \sigma\rangle \xrightarrow{\tau} \langle \downarrow,\, \sigma\,[x/v]\rangle \vdash p}\ [\text{pr–assgn}]$$

$$\frac{\langle C_1,\, \sigma\rangle \xrightarrow{\alpha} \langle C_1',\, \sigma'\rangle \vdash p,\ \alpha \in \mathit{Act} \quad 0 < p \leq 1}{\langle C_1; C_2,\, \sigma\rangle \xrightarrow{\alpha} \langle C_1'; C_2,\, \sigma'\rangle \vdash p}\ [\text{seq}_1] \qquad \frac{}{\langle \downarrow; C_2,\, \sigma\rangle \xrightarrow{\tau} \langle C_2,\, \sigma\rangle \vdash 1}\ [\text{seq}_2]$$

$$\frac{}{\langle \{C_1\} \,\Box\, \{C_2\},\, \sigma\rangle \xrightarrow{L} \langle C_1,\, \sigma\rangle \vdash 1}\ [\Box\text{–}L] \qquad \frac{}{\langle \{C_1\} \,\Box\, \{C_2\},\, \sigma\rangle \xrightarrow{R} \langle C_2,\, \sigma\rangle \vdash 1}\ [\Box\text{–}R]$$

$$\frac{[\![\xi\colon \mathsf{true}]\!](\sigma) = p > 0}{\langle \texttt{if}\ (\xi)\ \{C_1\}\ \texttt{else}\ \{C_2\},\, \sigma\rangle \xrightarrow{\tau} \langle C_1,\, \sigma\rangle \vdash p}\ [\text{if–true}]$$

$$\frac{[\![\xi\colon \mathsf{false}]\!](\sigma) = p > 0}{\langle \texttt{if}\ (\xi)\ \{C_1\}\ \texttt{else}\ \{C_2\},\, \sigma\rangle \xrightarrow{\tau} \langle C_2,\, \sigma\rangle \vdash p}\ [\text{if–false}]$$

$$\frac{}{\langle \texttt{while}\ (\xi)\ \{C\},\, \sigma\rangle \xrightarrow{\tau} \langle \texttt{if}\ (\xi)\ \{C; \texttt{while}\ (\xi)\ \{C\}\}\ \texttt{else}\ \{\texttt{empty}\},\, \sigma\rangle \vdash 1}\ [\text{while}]$$

Fig. 2. Rules for the transition probability function of operational MDPs.

then $\mathsf{ExpRew}^{\mathfrak{M}}\,(s \models \Diamond T) \triangleq \infty$. Otherwise,

$$\mathsf{ExpRew}^{\mathfrak{M}}\,(s \models \Diamond T) \ \triangleq\ \sup_{\mathfrak{S}} \sum_{\hat{\pi} \in \Pi(s,T)} \Pr{}^{\mathfrak{M}_{\mathfrak{S}}}\{\hat{\pi}\} \cdot \mathit{rew}(\hat{\pi})\ .$$

We are now in a position to define an operational model for our probabilistic programming language. Let $\downarrow$ denote a special symbol indicating successful termination of a program.

Definition 2 (The operational MDP of a program). *Given program $C \in$ pProgs, initial program state $\sigma_0 \in \Sigma$, and continuation $f \in \mathbb{T}$, the* operational MDP *of C is given by $\mathfrak{M}^f_{\sigma_0}[\![C]\!] = (\mathcal{S},\, \mathit{Act},\, \mathbf{P},\, s_0,\, \mathit{rew})$, where*

- $\mathcal{S} \triangleq ((\mathsf{pProgs} \cup \{\downarrow\} \cup \{\downarrow; C \mid C \in \mathsf{pProgs}\}) \times \Sigma) \cup \{\langle\, \mathit{sink}\,\rangle\}$,
- $\mathit{Act} \triangleq \{L,\, \tau,\, R\}$,
- *the transition probability function $\mathbf{P}$ is given by the rules in Figure 2,*
- $s_0 \triangleq \langle C,\, \sigma_0\rangle$*, and*
- $\mathit{rew}\colon \mathcal{S} \to \mathbb{R}_{\geq 0}$ *is the reward function defined according to Table 2.*

Since the initial state of the MDP $\mathfrak{M}^f_{\sigma_0}[\![C]\!]$ of a program C with initial state σ_0 is uniquely given, instead of $\mathsf{ExpRew}^{\mathfrak{M}^f_{\sigma_0}[\![C]\!]}\,(\langle C,\, \sigma_0\rangle \models \Diamond T)$ we simply write

$$\mathsf{ExpRew}^{\mathfrak{M}^f_{\sigma}[\![C]\!]}\,(T)\ .$$

s	$rew(s)$
$\langle \downarrow, \sigma \rangle$	$f(\sigma)$
$\langle \texttt{skip}, \sigma \rangle$, $\langle x :\approx \mu, \sigma \rangle$, $\langle \texttt{if}\ (\xi)\ \{C_1\}\ \texttt{else}\ \{C_2\}, \sigma \rangle$	1
$\langle\, sink \,\rangle$, $\langle \texttt{empty}, \sigma \rangle$, $\langle \texttt{halt}, \sigma \rangle$, $\langle \downarrow ; C_2, \sigma \rangle$, $\langle \{C_1\} \Box \{C_2\}, \sigma \rangle$, $\langle \texttt{while}\ (\xi)\ \{C\}, \sigma \rangle$	0
$\langle C_1 ; C_2, \sigma \rangle$	$rew(\langle C_1, \sigma \rangle)$

Table 2. Definition of the reward function $rew \colon \mathcal{S} \to \mathbb{R}_{\geq 0}$ of operational MDPs.

The rules in Figure 2 defining the transition probability function of a program's MDP are self–explanatory. Since only guard evaluations, assignments and `skip` statements are assumed to consume time, i.e. have a positive reward, we assign a reward of 0 to all other program statements. Moreover, note that all states of the form $\langle \texttt{empty}, \sigma \rangle$, $\langle \downarrow, \sigma \rangle$ and $\langle\, sink \,\rangle$ are needed, because an operational MDP is defined with respect to a given continuation $f \in \mathbb{T}$. In case of $\langle \texttt{empty}, \sigma \rangle$, a reward of 0 is collected and after that the program successfully terminates, i.e. enters state $\langle \downarrow, \sigma \rangle$ where the continuation f is collected as reward. In contrast, since no state other than $\langle\, sink \,\rangle$ is reachable from the unique sink state $\langle\, sink \,\rangle$, the continuation f is not taken into account if $\langle\, sink \,\rangle$ is reached without reaching a state $\langle \downarrow, \sigma \rangle$ first. Hence the operational MDP directly enters $\langle\, sink \,\rangle$ from a state of the form $\langle \texttt{halt}, \sigma \rangle$.

Example 3 (MDP of C_{trunc}). Recall the probabilistic program C_{trunc} from Example 2. Figure 3 depicts the MDP $\mathfrak{M}^{f}_{\sigma}[\![C_{trunc}]\!]$ for an arbitrary fixed state $\sigma \in \Sigma$ and an arbitrary continuation $f \in \mathbb{T}$. Here labeled edges denote the value of the transition probability function for the respective states, while the reward of each state is provided in gray next to the state. To improve readability, edge labels are omitted if the probability of a transition is 1. Moreover, $\mathfrak{M}^{f}_{\sigma}[\![C_{trunc}]\!]$ is a Markov chain, because C_{trunc} contains no non-deterministic choice.

A brief inspection of Figure 3 reveals that $\mathfrak{M}^{f}_{\sigma}[\![C_{trunc}]\!]$ contains three finite paths $\hat{\pi}_{\mathsf{true}}$, $\hat{\pi}_{\mathsf{false\ true}}$, $\hat{\pi}_{\mathsf{false\ false}}$ that eventually reach state $\langle\, sink \,\rangle$ starting from the initial state $\langle C_{trunc}, \sigma \rangle$. These paths correspond to the results of the two probabilistic guards in C. Hence the expected reward of $\mathfrak{M}^{f}_{\sigma}[\![C]\!]$ to eventually reach $T = \{\langle\, sink \,\rangle\}$ is given by

$$
\begin{aligned}
&\mathsf{ExpRew}^{\mathfrak{M}^{f}_{\sigma}[\![C_{trunc}]\!]}(T) \\
&\quad = \sup\nolimits_{\mathfrak{S}} \textstyle\sum_{\hat{\pi} \in \Pi(s,T)} \Pr^{\mathfrak{M}_{\mathfrak{S}}}\{\hat{\pi}\} \cdot rew(\hat{\pi}) \\
&\quad = \textstyle\sum_{\hat{\pi} \in \Pi(s,T)} \Pr^{\mathfrak{M}}\{\hat{\pi}\} \cdot rew(\hat{\pi}) \quad (\mathfrak{M}^{f}_{\sigma}[\![C_{trunc}]\!] = \mathfrak{M} \text{ is a Markov chain}) \\
&\quad = \Pr^{\mathfrak{M}}\{\hat{\pi}_{\mathsf{true}}\} \cdot rew(\hat{\pi}_{\mathsf{true}}) \;+\; \Pr^{\mathfrak{M}}\{\hat{\pi}_{\mathsf{false\ true}}\} \cdot rew(\hat{\pi}_{\mathsf{false\ true}}) \\
&\quad\quad + \Pr^{\mathfrak{M}}\{\hat{\pi}_{\mathsf{false\ false}}\} \cdot rew(\hat{\pi}_{\mathsf{false\ false}}) \\
&\quad = \left(\tfrac{1}{2} \cdot 1 \cdot 1\right) \cdot (1 + 1 + f(\sigma\,[succ/\mathsf{true}]))
\end{aligned}
$$

$$\begin{tikzcd}
 & & & \overset{1}{\langle C,\, \sigma\rangle} \arrow[dl, "1/2"'] \arrow[dr, "1/2"] & & \\
1 & \langle succ \;\mathsf{:=}\; \mathsf{true},\, \sigma\rangle \arrow[d] & & & \langle C',\, \sigma\rangle \arrow[lll, "1/2"'] \arrow[d, "1/2"] & 1 \\
f(\sigma\,[succ/\mathsf{true}]) & \langle {\downarrow},\, \sigma\,[succ/\mathsf{true}]\rangle \arrow[d] & & & \langle succ \;\mathsf{:=}\; \mathsf{false},\, \sigma\rangle \arrow[d] & 1 \\
0 & \langle\, sink \,\rangle \arrow[loop below] & & & \langle {\downarrow},\, \sigma\,[succ/\mathsf{false}]\rangle \arrow[lll] & f(\sigma\,[succ/\mathsf{false}])
\end{tikzcd}$$

Fig. 3. The operational MDP $\mathfrak{M}_\sigma^f[\![C_{trunc}]\!]$ corresponding to the program in Example 3. C' denotes the subprogram $\texttt{if}\,(1/2\cdot\langle\mathsf{true}\rangle + 1/2\cdot\langle\mathsf{false}\rangle)\,\{succ\;\mathsf{:=}\;\mathsf{true}\}\,\texttt{else}\,\{succ\;\mathsf{:=}\;\mathsf{false}\}$.

$$\begin{aligned}
&+\;\left(\tfrac{1}{2}\cdot\tfrac{1}{2}\cdot 1\cdot 1\right)\cdot\left(1+1+1+f(\sigma\,[succ/\mathsf{true}])\right)\\
&+\;\left(\tfrac{1}{2}\cdot\tfrac{1}{2}\cdot 1\cdot 1\right)\cdot\left(1+1+1+f(\sigma\,[succ/\mathsf{false}])\right)\\
=&\;\;1+\tfrac{1}{2}\cdot f(\sigma\,[succ/\mathsf{true}])\;+\;\tfrac{1}{4}\cdot\left(6+f(\sigma\,[succ/\mathsf{true}])\right.\\
&+\;\left.f(\sigma\,[succ/\mathsf{false}])\right)\\
=&\;\;\tfrac{5}{2}\;+\;\tfrac{3}{4}\cdot f(\sigma\,[succ/\mathsf{true}])\;+\;\tfrac{1}{4}\cdot f(\sigma\,[succ/\mathsf{false}]).
\end{aligned}$$

Observe that for $f = \mathbf{0}$, the expected reward $\mathsf{ExpRew}^{\mathfrak{M}_\sigma^f[\![C_{trunc}]\!]}(T)$ and the expected run–time $\mathsf{ert}\,[C]\,(f)\,(\sigma)$ (cf. Example 2) coincide, both yielding $5/2$. △

The main result of this section is that ert precisely captures the expected reward of the MDPs associated to our probabilistic programs.

Theorem 2 (Soundness of the ert transformer). *Let* $\xi \in \mathsf{DExp}$*,* $C \in \mathsf{pProgs}$*, and* $f \in \mathbb{T}$*. Then, for each* $\sigma \in \Sigma$*, we have*

$$\mathsf{ExpRew}^{\mathfrak{M}_\sigma^f[\![C]\!]}(\langle\, sink \,\rangle) \;=\; \mathsf{ert}\,[C]\,(f)\,(\sigma)\;.$$

Proof. By induction on the program structure; see [17] for details. □

5 Expected Run–Time of Loops

Reasoning about the run–time of loop–free programs consists mostly of syntactic reasoning. The run–time of a loop, however, is given in terms of a least fixed point. It is thus obtained by fixed point iteration but need not be reached within a finite number of iterations. To overcome this problem we next study invariant–based proof rules for approximating the run–time of loops.

We present two families of proof rules which differ in the kind of the invariants they build on. In Section 5.1 we present a proof rule that rests on the presence of an invariant approximating the entire run–time of a loop in a global manner, while in Section 5.2 we present two proof rules that rely on a parametrized invariant that approximates the run–time of a loop in an incremental fashion. Finally in Section 5.3 we discuss how to improve the run–time bounds yielded by these proof rules.

5.1 Proof Rule Based on Global Invariants

The first proof rule that we study allows upper–bounding the expected run–time of loops and rests on the notion of *upper invariants.*

Definition 3 (Upper invariants). *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$ and $\xi \in \mathsf{DExp}$. We say that $I \in \mathbb{T}$ is an* upper invariant *of loop $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f iff*

$$\mathbf{1} + [\![\xi\colon \mathsf{false}]\!] \cdot f + [\![\xi\colon \mathsf{true}]\!] \cdot \mathsf{ert}\,[C]\,(I) \quad \preceq I$$

or, equivalently, iff $F_f^{\langle \xi, C\rangle}(I) \preceq I$, where $F_f^{\langle \xi, C\rangle}$ is the characteristic functional.

The presence of an upper invariant of a loop readily establishes an upper bound of the loop's run–time.

Theorem 3 (Upper bounds from upper invariants). *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$ and $\xi \in \mathsf{DExp}$. If $I \in \mathbb{T}$ is an upper invariant of $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f then*

$$\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) \quad \preceq \quad I~.$$

Proof. The crux of the proof is an application of Park's Theorem[3] [25] which, given that $F_f^{\langle \xi, C\rangle}$ is continuous (see Lemma 1), states that

$$F_f^{\langle \xi, C\rangle}(I) \preceq I \quad \Longrightarrow \quad \mathsf{lfp}\, F_f^{\langle \xi, C\rangle} \preceq I~.$$

The left–hand side of the implication stands for I being an upper invariant, while the right–hand side stands for $\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) \preceq I$. □

Notice that if the loop body C is itself loop–free, it is usually fairly easy to verify that some $I \in \mathbb{T}$ is an upper invariant, whereas *inferring* the invariant is—as in standard program verification—one of the most involved part of the verification effort.

[3] If $H\colon \mathcal{D} \to \mathcal{D}$ is a continuous function over an ω–cpo $(\mathcal{D}, \sqsubseteq)$ with bottom element, then $H(d) \sqsubseteq d$ implies $\mathsf{lfp}\, H \sqsubseteq d$ for every $d \in \mathcal{D}$.

Example 4 (Geometric distribution). Consider loop

$$C_{\mathsf{geo}}\texttt{:}\quad \texttt{while}\,(c = 1)\,\{c :\approx {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle\}\ .$$

From the calculations below we conclude that $I = \mathbf{1} + [\![c = 1]\!] \cdot \mathbf{4}$ is an upper invariant with respect to $\mathbf{0}$:

$$\begin{aligned}
&\mathbf{1} + [\![c \neq 1]\!] \cdot \mathbf{0} + [\![c = 1]\!] \cdot \mathsf{ert}\,[c :\approx {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle]\,(I) \\
&\quad = \mathbf{1} + [\![c = 1]\!] \cdot \big(\mathbf{1} + \tfrac{1}{2} \cdot I\,[c/0] + \tfrac{1}{2} \cdot I\,[c/1]\big) \\
&\quad = \mathbf{1} + [\![c = 1]\!] \cdot \big(\mathbf{1} + \tfrac{1}{2} \cdot (\underbrace{\mathbf{1} + [\![0 = 1]\!] \cdot \mathbf{4}}_{=\mathbf{1}}) + \tfrac{1}{2} \cdot (\underbrace{\mathbf{1} + [\![1 = 1]\!] \cdot \mathbf{4}}_{=\mathbf{5}})\big) \\
&\quad = \mathbf{1} + [\![c = 1]\!] \cdot \mathbf{4} \;=\; I \;\preceq\; I
\end{aligned}$$

Then applying Theorem 3 we obtain

$$\mathsf{ert}\,\big[C_{\mathsf{geo}}\big]\,(\mathbf{0}) \;\preceq\; \mathbf{1} + [\![c = 1]\!] \cdot \mathbf{4}\ .$$

In words, the expected run–time of C_{geo} is at most 5 from any initial state where $c = 1$ and at most 1 from the remaining states. △

The invariant–based technique to reason about the run–time of loops presented in Theorem 3 is complete in the sense that there always exists an upper invariant that establishes the exact run–time of the loop at hand.

Theorem 4. *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$, $\xi \in \mathsf{DExp}$. Then there exists an upper invariant I of* $\texttt{while}\,(\xi)\,\{C\}$ *with respect to f such that* $\mathsf{ert}\,[\texttt{while}\,(\xi)\,\{C\}]\,(f) = I$.

Proof. The result follows from showing that $\mathsf{ert}\,[\texttt{while}\,(\xi)\,\{C\}]\,(f)$ is itself an upper invariant. Since $\mathsf{ert}\,[\texttt{while}\,(\xi)\,\{C\}]\,(f) = \mathsf{lfp}\,F_f^{\langle \xi, C\rangle}$ this amounts to showing that

$$F_f^{\langle \xi, C\rangle}\big(\,\mathsf{lfp}\,F_f^{\langle \xi, C\rangle}\,\big) \;\preceq\; \mathsf{lfp}\,F_f^{\langle \xi, C\rangle}\ ,$$

which holds by definition of lfp. □

Intuitively, the proof of this theorem shows that $\mathsf{ert}\,[\texttt{while}\,(\xi)\,\{C\}]\,(f)$ itself is the tightest upper invariant that the loop admits.

5.2 Proof Rules Based on Incremental Invariants

We now study a second family of proof rules which builds on the notion of ω–invariants to establish *both* upper and lower bounds for the run–time of loops.

Definition 4 (ω–invariants). *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$ and $\xi \in \mathsf{DExp}$. Moreover let $I_n \in \mathbb{T}$ be a run–time parametrized by $n \in \mathbb{N}$. We say that I_n is a* lower ω–invariant *of loop* $\texttt{while}\,(\xi)\,\{C\}$ *with respect to f iff*

$$F_f^{\langle \xi, C\rangle}(\mathbf{0}) \succeq I_0 \qquad \textit{and} \qquad F_f^{\langle \xi, C\rangle}(I_n) \succeq I_{n+1} \quad \textit{for all } n \geq 0\ .$$

Dually, we say that I_n is an upper ω–invariant *iff*

$$F_f^{\langle \xi, C\rangle}(\mathbf{0}) \preceq I_0 \qquad \textit{and} \qquad F_f^{\langle \xi, C\rangle}(I_n) \preceq I_{n+1} \quad \textit{for all } n \geq 0\ .$$

Intuitively, a lower (resp. upper) ω–invariant I_n represents a lower (resp. upper) bound for the expected run–time of those program runs that finish within $n+1$ iterations, weighted according to their probabilities. Therefore we can use the asymptotic behavior of I_n to approximate from below (resp. above) the expected run–time of the entire loop.

Theorem 5 (Bounds from ω–invariants). *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$, $\xi \in \mathsf{DExp}$.*

1. *If I_n is a lower ω–invariant of $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f and $\lim_{n\to\infty} I_n$ exists*[4]*, then*
$$\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) \succeq \lim_{n\to\infty} I_n\ .$$
2. *If I_n is an upper ω–invariant of $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f and $\lim_{n\to\infty} I_n$ exists, then*
$$\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) \preceq \lim_{n\to\infty} I_n\ .$$

Proof. We prove only the case of lower ω–invariants since the other case follows by a dual argument. Let F_f be the characteristic functional of the loop with respect to f. Let $F_f^0 = \mathbf{0}$ and $F_f^{n+1} = F_f(F_f^n)$. By the Kleene Fixed Point Theorem, $\mathsf{ert}\,[\mathtt{while}\,(\xi)\,\{C\}]\,(f) = \sup_n F_f^n$ and since $F_f^0 \preceq F_f^1 \preceq \ldots$ forms an ω–chain, by the Monotone Sequence Theorem[5], $\sup_n F_f^n = \lim_{n\to\infty} F_f^n$. Then the proof follows from showing that $F_f^{n+1} \succeq I_n$. We prove this by induction on n. The base case $F_f^1 \succeq I_0$ holds because I_n is a lower ω–invariant. For the inductive case we reason as follows:

$$F_f^{n+2} = F_f\big(F_f^{n+1}\big) \succeq F_f(I_n) \succeq I_{n+1}\ .$$

Here the first inequality follows by I.H. and the monotonicity of F_f (recall that $\mathsf{ert}[C]$ is monotonic by Theorem 1), while the second inequality holds because I_n is a lower ω–invariant. □

Example 5 (Lower bounds for C_{geo}). Reconsider loop C_{geo} from Example 4. Now we use Theorem 5.1 to show that $\mathbf{1} + [\![c=1]\!]\cdot\mathbf{4}$ is also a lower bound of its run–time. To this end we first show that $I_n = \mathbf{1} + [\![c=1]\!]\cdot(\mathbf{4} - \mathbf{3}/\mathbf{2}^n)$ is a lower ω–invariant of the loop with respect to $\mathbf{0}$:

$$\begin{aligned}
F_{\mathbf{0}}(\mathbf{0}) &= \mathbf{1} + [\![c\neq 1]\!]\cdot\mathbf{0} + [\![c=1]\!]\cdot\mathsf{ert}\,[c :\approx {}^1\!/_2\langle 0\rangle + {}^1\!/_2\langle 1\rangle]\,(\mathbf{0})\\
&= \mathbf{1} + [\![c=1]\!]\cdot\big(\mathbf{1} + \tfrac{1}{2}\cdot\mathbf{0}\,[c/0] + \tfrac{1}{2}\cdot\mathbf{0}\,[c/1]\big)\\
&= \mathbf{1} + [\![c=1]\!]\cdot\mathbf{1} \;=\; \mathbf{1} + [\![c=1]\!]\cdot(\mathbf{4} - \mathbf{3}/\mathbf{2}^0) \;=\; I_0 \;\succeq\; I_0
\end{aligned}$$

$$\begin{aligned}
F_{\mathbf{0}}(I_n) &= \mathbf{1} + [\![c\neq 1]\!]\cdot\mathbf{0} + [\![c=1]\!]\cdot\mathsf{ert}\,[c :\approx {}^1\!/_2\langle 0\rangle + {}^1\!/_2\langle 1\rangle]\,(I_n)\\
&= \mathbf{1} + [\![c=1]\!]\cdot\big(\mathbf{1} + \tfrac{1}{2}\cdot I_n\,[c/0] + \tfrac{1}{2}\cdot I_n\,[c/1]\big)
\end{aligned}$$

[4] Limit $\lim_{n\to\infty} I_n$ is to be understood pointwise, on $\mathbb{R}_{\geq 0}^{\infty}$, i.e. $\lim_{n\to\infty} I_n = \lambda\sigma.\lim_{n\to\infty} I_n(\sigma)$ and $\lim_{n\to\infty} I_n(\sigma) = \infty$ is considered a valid value.

[5] If $\langle a_n\rangle_{n\in\mathbb{N}}$ is an increasing sequence in $\mathbb{R}_{\geq 0}^{\infty}$, then $\lim_{n\to\infty} a_n$ coincides with supremum $\sup_n a_n$.

$$= \mathbf{1} + [\![c=1]\!] \cdot \Big(\mathbf{1} + \tfrac{1}{2}\cdot(\mathbf{1}+\mathbf{0}) + \tfrac{1}{2}\cdot\big(\mathbf{1} + \big(\mathbf{4} - \tfrac{\mathbf{3}}{\mathbf{2}^n}\big)\big)\Big)$$
$$= \mathbf{1} + [\![c=1]\!] \cdot \big(\mathbf{4} - \tfrac{\mathbf{3}}{\mathbf{2}^{n+1}}\big) \;=\; I_{n+1} \;\succeq\; I_{n+1}$$

Then from Theorem 5.1 we obtain

$$\mathsf{ert}\left[C_{\mathsf{geo}}\right](\mathbf{0}) \;\succeq\; \lim_{n\to\infty} \Big(\mathbf{1} + [\![c=1]\!]\cdot\big(\mathbf{4} - \tfrac{\mathbf{3}}{\mathbf{2}^n}\big)\Big) \;=\; \mathbf{1} + [\![c=1]\!]\cdot\mathbf{4}\;.$$

Combining this result with the upper bound $\mathsf{ert}\left[C_{\mathsf{geo}}\right](\mathbf{0}) \preceq \mathbf{1} + [\![c=1]\!]\cdot\mathbf{4}$ established in Example 4 we conclude that $\mathbf{1} + [\![c=1]\!]\cdot\mathbf{4}$ is the exact run–time of C_{geo}. Observe, however, that the above calculations show that I_n is both a lower and an upper ω–invariant (exact equalities $F_0(\mathbf{0}) = I_0$ and $F_0(I_n) = I_{n+1}$ hold). Then we can apply Theorem 5.1 and 5.2 simultaneously to derive the exact run–time without having to resort to the result from Example 4.

Invariant Synthesis. In order to synthesize invariant $I_n = \mathbf{1} + [\![c=1]\!]\cdot(\mathbf{4} - \mathbf{3}/\mathbf{2}^n)$, we proposed template $I_n = \mathbf{1} + [\![c=1]\!]\cdot a_n$ and observed that under this template the definition of lower ω–invariant reduces to $a_0 \leq 1$, $a_{n+1} \leq 2 + \frac{1}{2}a_n$, which is satisfied by $a_n = 4 - 3/2^n$. △

Now we apply Theorem 5.1 to a program with infinite expected run–time.

Example 6 (Almost–sure termination at infinite expected run–time). Recall the program from the introduction:

```
C:  1:  x := 1; b := 1;
    2:  while (b = 1) {b :≈ 1/2⟨0⟩ + 1/2⟨1⟩; x := 2x};
    3:  while (x > 0) {x := x − 1}
```

Let C_i denote the i-th line of C. We show that $\mathsf{ert}[C](\mathbf{0}) \succeq \infty$.[6] Since

$$\mathsf{ert}[C](\mathbf{0}) \;=\; \mathsf{ert}[C_1]\left(\mathsf{ert}[C_2]\left(\mathsf{ert}[C_3](\mathbf{0})\right)\right)$$

we start by showing that

$$\mathsf{ert}[C_3](\mathbf{0}) \;\succeq\; \mathbf{1} + [\![x>0]\!]\cdot 2x$$

using lower ω–invariant $J_n = \mathbf{1} + [\![n > x > 0]\!]\cdot 2x + [\![x \geq n]\!]\cdot(2n-1)$. We omit here the details of verifying that J_n is a lower ω–invariant. Next we show that

$$\begin{aligned}\mathsf{ert}[C_2]\left(\mathbf{1} + [\![x>0]\!]\cdot 2x\right) \;\succeq\; & \mathbf{1} + [\![b \neq 1]\!]\cdot\big(\mathbf{1} + [\![x>0]\!]\cdot 2x\big) \\ & + [\![b = 1]\!]\cdot\big(\mathbf{7} + [\![x>0]\!]\cdot\boldsymbol{\infty}\big)\end{aligned}$$

by means of the lower ω–invariant

$$I_n \;=\; \mathbf{1} + [\![b\neq 1]\!]\cdot\big(\mathbf{1} + [\![x>0]\!]\cdot 2x\big) + [\![b=1]\!]\cdot\Big(\mathbf{7} - \tfrac{\mathbf{5}}{\mathbf{2^n}} + n\cdot[\![x>0]\!]\cdot 2x\Big)\;.$$

[6] Note that while this program terminates with probability one, the expected run–time to achieve termination is infinite.

Let F be the characteristic functional of loop C_2 with respect to $\mathbf{1} + [\![x > 0]\!] \cdot 2x$. The calculations to establish that I_n is a lower ω–invariant now go as follows:

$$\begin{aligned} F(\mathbf{0}) = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) \\ & + [\![b = 1]\!] \cdot \Big(\mathbf{1} + \tfrac{1}{2} \cdot (\mathbf{1} + \mathbf{0}\,[x, b/2x, 0]) + \tfrac{1}{2} \cdot (\mathbf{1} + \mathbf{0}\,[x, b/2x, 1])\Big) \\ = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) + [\![b = 1]\!] \cdot \big(\mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1} + \tfrac{1}{2} \cdot \mathbf{1}\big) \\ = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) + [\![b = 1]\!] \cdot \mathbf{2} \; = \; I_0 \; \succeq \; I_0 \end{aligned}$$

$$\begin{aligned} F(I_n) = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) \\ & + [\![b = 1]\!] \cdot \Big(\mathbf{1} + \tfrac{1}{2} \cdot (\mathbf{1} + I_n\,[x, b/2x, 0]) + \tfrac{1}{2} \cdot (\mathbf{1} + I_n\,[x, b/2x, 1])\Big) \\ = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) \\ & + [\![b = 1]\!] \cdot \Big(\mathbf{1} + \tfrac{1}{2} \cdot (\mathbf{3} + [\![2x > 0]\!] \cdot 4x) + \tfrac{1}{2} \cdot \Big(\mathbf{9} - \tfrac{\mathbf{5}}{\mathbf{2^n}} + n \cdot [\![2x > 0]\!] \cdot 4x\Big)\Big) \\ = \; & \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) \\ & + [\![b = 1]\!] \cdot \Big(\mathbf{7} - \tfrac{\mathbf{5}}{\mathbf{2^{n+1}}} + (n{+}1) \cdot [\![x > 0]\!] \cdot 2x\Big) \\ = \; & I_{n+1} \; \succeq \; I_{n+1} \end{aligned}$$

Now we can complete the run–time analysis of program C:

$$\begin{aligned} & \mathsf{ert}\,[C]\,(\mathbf{0}) \\ & = \; \mathsf{ert}\,[C_1]\,(\mathsf{ert}\,[C_2]\,(\mathsf{ert}\,[C_3]\,(\mathbf{0}))) \\ & \succeq \; \mathsf{ert}\,[C_1]\,\big(\mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) + [\![b = 1]\!] \cdot \big(\mathbf{7} + [\![x > 0]\!] \cdot \infty\big)\big) \\ & = \; \mathsf{ert}[x \mathrel{:=} 1]\Big(\mathsf{ert}[b \mathrel{:=} 1]\Big(\mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) \\ & \qquad\qquad\qquad\qquad + [\![b = 1]\!] \cdot \big(\mathbf{7} + [\![x > 0]\!] \cdot \infty\big)\Big)\Big) \\ & = \; \mathsf{ert}\,[x \mathrel{:=} 1]\,(\mathbf{8} + [\![x > 0]\!] \cdot \infty) \; = \; \mathbf{8} + \infty \; = \; \infty \end{aligned}$$

Overall, we obtain that the expected run–time of the program C is infinite even though it terminates with probability one. Notice furthermore that sub–programs `while` $(b = 1)$ $\{b :\approx {}^1/_2\langle 0\rangle + {}^1/_2\langle 1\rangle\,;\; x \mathrel{:=} 2x\}$ and `while` $(x > 0)$ $\{x \mathrel{:=} x - 1\}$ have expected run–time $\mathbf{1} + [\![b]\!] \cdot \mathbf{4}$ and $\mathbf{1} + [\![x > 0]\!] \cdot 2x$, respectively, i.e. both have a finite expected run–time.

Invariant synthesis. In order to synthesize the ω–invariant I_n of loop C_2 we propose the template $I_n = \mathbf{1} + [\![b \neq 1]\!] \cdot \big(\mathbf{1} + [\![x > 0]\!] \cdot 2x\big) + [\![b = 1]\!] \cdot \big(a_n + b_n \cdot [\![x > 0]\!] \cdot 2x\big)$ and from the definition of lower ω–invariants we obtain $a_0 \leq 2$, $a_{n+1} \leq {}^7/_2 + {}^1/_2 \cdot a_n$ and $b_0 \leq 0$, $b_{n+1} \leq 1 + b_n$. These recurrences admit solutions $a_n = 7 - {}^5/_{2^n}$ and $b_n = n$. △

As the proof rule based on upper invariants, the proof rules based on ω-invariants are also complete: Given loop `while` (ξ) $\{C\}$ and run–time f, it is enough to consider the ω-invariant $I_n = F_f^{n+1}$, where F_f^n is defined as in the proof of

Theorem 5 to yield the exact run–time $\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f)$ from an application of Theorem 5. We formally capture this result by means of the following theorem:

Theorem 6. *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$ and $\xi \in \mathsf{DExp}$. Then there exists a (both lower and upper) ω–invariant I_n of $\mathtt{while}\,(\xi)\,\{C\}$ with respect to f such that $\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f) = \lim_{n\to\infty} I_n$.*

Theorem 6 together with Theorem 4 shows that the set of invariant–based proof rules presented in this section are complete. Next we study how to refine invariants to make the bounds that these proof rules yield more precise.

5.3 Refinement of Bounds

An important property of both upper and lower bounds of the run–time of loops is that they can be easily refined by repeated application of the characteristic functional.

Theorem 7 (Refinement of bounds). *Let $f \in \mathbb{T}$, $C \in \mathsf{pProgs}$ and $\xi \in \mathsf{DExp}$. If I is an upper (resp. lower) bound of $\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f)$ and $F_f^{\langle \xi, C\rangle}(I) \preceq I$ (resp. $F_f^{\langle \xi, C\rangle}(I) \succeq I$), then $F_f^{\langle \xi, C\rangle}(I)$ is also an upper (resp. lower) bound, at least as precise as I.*

Proof. If I is an upper bound of $\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f)$ we have $\mathsf{lfp}\, F_f^{\langle \xi, C\rangle} \preceq I$. Then from the monotonicity of $F_f^{\langle \xi, C\rangle}$ (recall that ert is monotonic by Theorem 1) and from $F_f^{\langle \xi, C\rangle}(I) \preceq I$ we obtain

$$\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f) \;=\; \mathsf{lfp}\, F_f^{\langle \xi, C\rangle} \;=\; F_f^{\langle \xi, C\rangle}(\mathsf{lfp}\, F_f^{\langle \xi, C\rangle}) \;\preceq\; F_f^{\langle \xi, C\rangle}(I) \;\preceq\; I\;,$$

which means that $F_f^{\langle \xi, C\rangle}(I)$ is also an upper bound, possibly tighter than I. The case for lower bounds is completely analogous. □

Notice that if I is an upper invariant of $\mathtt{while}\,(\xi)\,\{C\}$ then I fulfills all necessary conditions of Theorem 7. In practice, Theorem 7 provides a means of iteratively improving the precision of bounds yielded by Theorems 3 and 5, as for instance for upper bounds we have

$$\mathsf{ert}\left[\mathtt{while}\,(\xi)\,\{C\}\right](f) \;\preceq\; \cdots \;\preceq\; F_f^{\langle \xi, C\rangle}\left(F_f^{\langle \xi, C\rangle}(I)\right) \;\preceq\; F_f^{\langle \xi, C\rangle}(I) \;\preceq I\;.$$

If I_n is an upper (resp. lower) ω-invariant, applying Theorem 7 requires checking that $F_f^{\langle \xi, C\rangle}(L) \preceq L$ (resp. $F_f^{\langle \xi, C\rangle}(L) \succeq L$), where $L = \lim_{n\to\infty} I_n$. This proof obligation can be discharged by showing that I_n forms an ω-chain, i.e. that $I_n \preceq I_{n+1}$ for all $n \in \mathbb{N}$.

6 Run–Time of Deterministic Programs

The notion of expected run–times as defined by ert is clearly applicable to deterministic programs, i.e. programs containing neither probabilistic guards nor probabilistic assignments nor non–deterministic choice operators. We show that the ert of deterministic programs coincides with the tightest upper bound on the run–time that can be derived in an extension of Hoare logic [14] due to Nielson [23,24].

In order to compare our notion of ert to the aforementioned calculus we restrict our programming language to the language dProgs of deterministic programs considered in [24] which is given by the following grammar:

$$C \quad ::= \quad \texttt{skip} \;\mid\; x \texttt{ := } E \;\mid\; C\,;\,C \;\mid\; \texttt{if}\,(\xi)\,\{C\}\,\texttt{else}\,\{C\} \;\mid\; \texttt{while}\,(\xi)\,\{C\}\ ,$$

where E is a *deterministic* expression and ξ is a *deterministic* guard, i.e. $[\![E]\!](\sigma)$ and $[\![\xi]\!](\sigma)$ are Dirac distributions for each $\sigma \in \Sigma$. For simplicity, we slightly abuse notation and write $[\![E]\!](\sigma)$ to denote the unique value $v \in \mathsf{Val}$ such that $[\![E \colon v]\!](\sigma) = 1$.

For deterministic programs, the MDP $\mathfrak{M}^{\mathbf{0}}_{\sigma}[\![C]\!]$ of a program $C \in \mathsf{dProgs}$ and a program state $\sigma \in \Sigma$ is a labeled transition system. In particular, if a terminal state of the form $\langle\downarrow, \sigma'\rangle$ is reachable from the initial state of $\mathfrak{M}^{\mathbf{0}}_{\sigma}[\![C]\!]$, it is unique. Hence we may capture the effect of a deterministic program by a partial function $\mathbb{C}[\![\,\cdot\,]\!](\,\cdot\,) \colon \mathsf{dProgs} \times \Sigma \rightharpoonup \Sigma$ mapping each $C \in \mathsf{dProgs}$ and $\sigma \in \Sigma$ to a program state $\sigma' \in \Sigma$ if and only if there exists a state $\langle\downarrow, \sigma'\rangle$ that is reachable in the MDP $\mathfrak{M}^{\mathbf{0}}_{\sigma}[\![C]\!]$ from the initial state $\langle C, \sigma\rangle$. Otherwise, $\mathbb{C}[\![C]\!](\sigma)$ is undefined.

Nielson [23,24] developed an extension of the classical Hoare calculus for total correctness of programs in order to establish additionally upper bounds on the run–time of programs. Formally, a *correctness property* is of the form

$$\{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}\ ,$$

where $C \in \mathsf{dProgs}$, E is a deterministic expression over the program variables, and P, Q are (first–order) assertions. Intuitively, $\{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}$ is valid, written $\models_E \{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}$, if and only if there exists a natural number k such that for each state σ satisfying the precondition P, the program C terminates after at most $k \cdot [\![E]\!](\sigma)$ steps in a state satisfying postcondition Q. In particular, it should be noted that E is evaluated in the *initial* state σ.

Figure 4 is taken verbatim from [24] except for minor changes to match our notation. Most of the inference rules are self–explanatory extensions of the standard Hoare calculus for total correctness of deterministic programs [14] which is obtained by omitting the gray parts.

The run–time of $\texttt{skip}$ and $x \texttt{ := } E$ is one time unit. Since guard evaluations are assumed to consume no time in this calculus, any upper bound on the run–time of both branches of a conditional is also an upper bound on the run–time of the conditional itself (cf. rule [if]). The rule of consequence allows to increase an already proven upper bound on the run–time by an arbitrary constant factor. Furthermore, the run–time of two sequentially composed programs C_1 and C_2

$$\dfrac{}{\{\ P\ \}\ \texttt{skip}\ \{\ 1\ \Downarrow\ P\ \}}\ [\text{skip}] \qquad \dfrac{}{\{\ Q\,[x/[\![E]\!]]\ \}\ x\ \texttt{:=}\ E\ \{\ 1\ \Downarrow\ Q\ \}}\ [\text{assgn}]$$

$$\dfrac{\{\ P \wedge E_2' = u\ \}\ C_1\ \{\ E_1\ \Downarrow\ Q \wedge E_2 \leq u\ \} \quad \{\ Q\ \}\ C_2\ \{\ E_2\ \Downarrow\ R\ \}}{\{\ P\ \}\ C_1; C_2\ \{\ E_1 + E_2'\ \Downarrow\ R\ \}}\ [\text{seq}]$$

where u is a fresh logical variable

$$\dfrac{\{\ P \wedge \xi\ \}\ C_1\ \{\ E\ \Downarrow\ Q\ \} \quad \{\ P \wedge \neg\xi\ \}\ C_2\ \{\ E\ \Downarrow\ Q\ \}}{\{\ P\ \}\ \texttt{if}\ (\xi)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{\ E\ \Downarrow\ Q\ \}}\ [\text{if}]$$

$$\dfrac{\{\ P(z+1) \wedge E' = u\ \}\ C\ \{\ E_1\ \Downarrow\ P(z)\ \wedge\ E \leq u\ \}}{\{\ \exists z_\bullet\ P(z)\ \}\ \texttt{while}\ (\xi)\ \{C\}\ \{\ E\ \Downarrow\ P(0)\ \}}\ [\text{while}]$$

where $z \in \mathbb{N},\ P(z+1) \Rightarrow \xi\ \wedge\ E \geq E_1 + E',\ P(0) \Rightarrow \neg\xi\ \wedge\ E \geq 1$ and u is a fresh logical variable

$$\dfrac{\{\ P'\ \}\ C\ \{\ E'\ \Downarrow\ Q'\ \}}{\{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}}\ [\text{cons}]$$

where $P \Rightarrow P'\ \wedge\ E' \leq k \cdot E$ for some $k \in \mathbb{N}$ and $Q' \Rightarrow Q$

Fig. 4. Inference system for order of magnitude of run–time of deterministic programs according to Nielson [23].

is, intuitively, the sum of their run–times E_1 and E_2. However, run–times are expressions which are evaluated in the initial state. Thus, the run–time of C_2 has to be expressed in the initial state of $C_1; C_2$. Technically, this is achieved by adding a fresh (and hence universally quantified) variable u that is an upper bound on E_2 and at the same time is equal to a new expression E_2' in the precondition of $C_1; C_2$. Then, the run–time of $C_1; C_2$ is given by the sum $E_1 + E_2'$.

The same principle is applied to each loop iteration. Here, the run–time of the loop body is given by E_1 and the run–time of the remaining z loop iterations, E', is expressed in the initial state by adding a fresh variable u. Then, any upper bound of $E \geq E_1 + E'$ is an upper bound on the run–time of z loop iterations.

We denote provability of a correctness property $\{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}$ and a total correctness property $\{\ P\ \}\ C\ \{\ \Downarrow\ Q\ \}$ in the standard Hoare calculus by $\vdash_E \{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}$ and $\vdash \{\ P\ \}\ C\ \{\ \Downarrow\ Q\ \}$, respectively.

Theorem 8 (Soundness of ert for deterministic programs). *For all $C \in$ dProgs and assertions P, Q, we have*

$$\vdash \{\ P\ \}\ C\ \{\ \Downarrow\ Q\ \}\ \textit{implies}\ \vdash_E \{\ P\ \}\ C\ \{\ \mathsf{ert}\,[C]\,(\mathbf{0})\ \Downarrow\ Q\ \}.$$

Proof. By induction on the program structure; see [17] for details.

Intuitively, this theorem means that for every terminating deterministic program, the ert is an upper bound on the run–time, i.e. ert is sound with respect to the inference system shown in Figure 4. The next theorem states that no tighter bound can be derived in this calculus. We cannot get a more precise relationship, since we assume guard evaluations to consume time.

Theorem 9 (Completeness of ert w.r.t. Nielson). *For all $C \in \mathsf{dProgs}$, assertions P, Q and deterministic expressions E, $\vdash_E \{\ P\ \}\ C\ \{\ E\ \Downarrow\ Q\ \}$ implies that there exists a natural number k such that for all $\sigma \in \Sigma$ satisfying P, we have*

$$\mathsf{ert}\,[C]\,(\mathbf{0})\,(\sigma) \ \leq\ k \cdot (\llbracket E \rrbracket(\sigma)) \ .$$

Proof. By induction on the program structure; see [17] for details. □

Theorem 8 together with Theorem 9 shows that our notion of ert is a conservative extension of Nielson's approach for reasoning about the run–time of deterministic programs. In particular, given a correctness proof of a deterministic program C in Hoare logic, it suffices to compute $\mathsf{ert}\,[C]\,(\mathbf{0})$ in order to obtain a corresponding proof in Nielson's proof system.

7 Case Studies

In this section we use our ert–calculus to analyze the run–time of two well–known randomized algorithms: the *One–Dimensional (Symmetric) Random Walk* and the *Coupon Collector Problem.*

7.1 One–Dimensional Random Walk

Consider program

$$P_{rw}\texttt{:}\quad x \texttt{ := } 10\texttt{;}\ \texttt{while}\,(x > 0)\,\{x \texttt{ :}\approx 1/2 \cdot \langle x{-}1 \rangle + 1/2 \cdot \langle x{+}1 \rangle\}\ ,$$

which models a one–dimensional walk of a particle which starts at position $x = 10$ and moves with equal probability to the left or to the right in each turn. The random walk stops if the particle reaches position $x = 0$. It can be shown that the program terminates with probability one [15] but requires, on average, an infinite time to do so. We now apply our ert–calculus to formally derive this run–time assertion.

The expected run–time of P_{rw} is given by

$$\mathsf{ert}\,[P_{rw}]\,(\mathbf{0}) \ = \ \mathsf{ert}\,[x \texttt{ := } 10]\,(\mathsf{ert}\,[\texttt{while}\ (x > 0)\ \{C\}]\,(\mathbf{0}))\ ,$$

where C stands for the probabilistic assignment in the loop body. Thus, we need to first determine run–time $\mathsf{ert}\,[\texttt{while}\ (x > 0)\ \{C\}]\,(\mathbf{0})$. To do so we propose

$$I_n \ = \ \mathbf{1} + \llbracket 0 < x \leq n \rrbracket \cdot \boldsymbol{\infty}$$

as a lower ω–invariant of loop $\texttt{while}\ (x > 0)\ \{C\}$ with respect to $\mathbf{0}$; detailed calculations for verifying that I_n is indeed a lower ω–invariant can be found in the extended version of the paper [17]. Theorem 5 then states that

$$\mathsf{ert}\,[\texttt{while}\ (x > 0)\ \{C\}]\,(\mathbf{0}) \succeq \lim_{n \to \infty} \mathbf{1} + \llbracket 0 < x \leq n \rrbracket \cdot \boldsymbol{\infty} = \mathbf{1} + \llbracket 0 < x \rrbracket \cdot \boldsymbol{\infty}\ .$$

Altogether we have

$$\begin{aligned}
\mathsf{ert}\,[P_{rw}]\,(\mathbf{0}) &= \mathsf{ert}\,[x \mathrel{:=} 10]\,(\mathsf{ert}\,[\mathtt{while}\,(x > 0)\,\{C\}]\,(\mathbf{0})) \\
&\succeq \mathsf{ert}\,[x \mathrel{:=} 10]\,(\mathbf{1} + [\![0 < x]\!] \cdot \boldsymbol{\infty}) \\
&= \mathbf{1} + (\mathbf{1} + [\![0 < x]\!] \cdot \boldsymbol{\infty})\,[x/10] \\
&= \mathbf{1} + (\mathbf{1} + 1 \cdot \boldsymbol{\infty}) \;=\; \boldsymbol{\infty}\;,
\end{aligned}$$

which says that $\mathsf{ert}\,[P_{rw}]\,(\mathbf{0}) \succeq \boldsymbol{\infty}$. Since the reverse inequality holds trivially, we conclude that $\mathsf{ert}\,[P_{rw}]\,(\mathbf{0}) = \boldsymbol{\infty}$.

7.2 The Coupon Collector Problem

Now we apply our ert–calculus to solve the Coupon Collector Problem. This problem arises from the following scenario[7]: Suppose each box of cereal contains one of N different coupons and once a consumer has collected a coupon of each type, he can trade them for a prize. The aim of the problem is determining the average number of cereal boxes the consumer should buy to collect all coupon types, assuming that each coupon type occurs with the same probability in the cereal boxes.

The problem can be modeled by program C_{cp} below:

```
$cp \mathrel{:=} [0,\,\ldots,\,0]$; $i \mathrel{:=} 1$; $x \mathrel{:=} N$
while $(x > 0)$ {
    while $(cp[i] \neq 0)$ {
        $i \mathrel{:\approx} \mathtt{Unif}[1 \ldots N]$
    };
    $cp[i] \mathrel{:=} 1$; $x \mathrel{:=} x - 1$
}
```

Array cp is initialized to 0 and whenever we obtain the first coupon of type i, we set $cp[i]$ to 1. The outer loop is iterated N times and in each iteration we collect a new—unseen—coupon type. The collection of the new coupon type is performed by the inner loop.

We start the run–time analysis of C_{cp} introducing some notation. Let C_{in} and C_{out}, respectively, denote the inner and the outer loop of C_{cp}. Furthermore, let $\#col \triangleq \sum_{i=1}^{N}[cp[i] \neq 0]$ denote the number of coupons that have already been collected.

Analysis of the inner loop. For analyzing the run–time of the outer loop we need to refer to the run–time of its body, with respect to an arbitrary continuation $g \in \mathbb{T}$. Therefore, we first analyze the run–time of the inner loop C_{in}. We propose the following lower and upper ω–invariant for the inner loop C_{in}:

[7] The problem formulation presented here is taken from [20].

$$\begin{aligned} J^g_{\mathrm{n}} \;=\; & \mathbf{1} \;+\; [cp[i] = 0] \cdot g \\ & +\; [cp[i] \neq 0] \cdot \sum_{k=0}^{n} \left(\frac{\#col}{N} \right)^k \left(\mathbf{2} + \frac{1}{N} \sum_{j=1}^{N} [\![cp[j] = 0]\!] \cdot g[i/j] \right) . \end{aligned}$$

Moreover, we write J^g for the same invariant where n is replaced by ∞. A detailed verification that J^g_{n} is indeed a lower and upper ω–invariant is provided in the extended version of the paper [17]. Theorem 5 now yields

$$J^g = \lim_{n \to \infty} J^g_{\mathrm{n}} \;\preceq\; \mathsf{ert}\left[C_{\mathrm{in}}\right](g) \;\preceq\; \lim_{n \to \infty} J^g_{\mathrm{n}} = J^g. \tag{$\star$}$$

Since the run–time of a deterministic assignment $x \;\texttt{:=}\; E$ is

$$\mathsf{ert}\left[x \;\texttt{:=}\; E\right](f) \;=\; \mathbf{1} + f\left[x/E\right] \;, \tag{$\maltese$}$$

the expected run–time of the body of the outer loop reduces to

$$\begin{aligned} & \mathsf{ert}\left[C_{\mathrm{in}}\,;\; cp[i] \;\texttt{:=}\; 1\,;\; x \;\texttt{:=}\; x-1\right](g) && (\dagger) \\ =\; & \mathbf{2} + \mathsf{ert}\left[C_{\mathrm{in}}\right](g[x/x-1,\; cp[i]/1]) && (\text{by } \maltese) \\ =\; & \mathbf{2} + J^{g[x/x-1,\; cp[i]/1]} && (\text{by } \star) \\ =\; & \mathbf{2} + J^g[x/x-1,\; cp[i]/1] \;. \end{aligned}$$

Analysis of the outer loop. Since program C_{cp} terminates right after the execution of the outer loop C_{out}, we analyze the run–time of the outer loop C_{out} with respect to continuation $\mathbf{0}$, i.e. $\mathsf{ert}\left[C_{out}\right](\mathbf{0})$. To this end we propose

$$\begin{aligned} I_n \;=\; & \mathbf{1} + \sum_{\ell=0}^{n} [x > \ell] \cdot \left(\mathbf{3} + [n \neq 0] + 2 \cdot \sum_{k=0}^{\infty} \left(\frac{\#col + \ell}{N} \right)^k \right) \\ & - 2 \cdot [cp[i] = 0] \cdot [x > 0] \cdot \sum_{k=0}^{\infty} \left(\frac{\#col}{N} \right)^k \end{aligned}$$

as both an upper and lower ω–invariant of C_{out} with respect to $\mathbf{0}$. A detailed verification that I_n is an ω-invariant is found in the extended version of the paper [17]. Now Theorem 5 yields

$$I \;=\; \lim_{n \to \infty} I_n \;\preceq\; \mathsf{ert}\left[C_{\mathrm{out}}\right](\mathbf{0}) \;\preceq\; \lim_{n \to \infty} I_n = I \;, \tag{$\ddagger$}$$

where I denotes the same invariant as I_n with n replaced by ∞.

Analysis of the overall program. To obtain the overall expected run–time of program C_{cp} we have to account for the initialization instructions before the outer loop. The calculations go as follows:

$$
\begin{aligned}
&\mathsf{ert}\,[C_{cp}]\,(\mathbf{0})\\
&\quad= \mathsf{ert}\,[cp \mathrel{:=} [0,\ldots,0]\,;\, i \mathrel{:=} 1\,;\, x \mathrel{:=} N\,;\, C_{\mathrm{out}}]\,(\mathbf{0})\\
&\quad= \mathbf{3} + \mathsf{ert}\,[C_{\mathrm{out}}]\,(\mathbf{0})\,[x/N, i/1, cp[1]/0, \ldots, cp[N]/0] && (\text{by } \maltese)\\
&\quad= \mathbf{3} + I[x/N, i/1, cp[1]/0, \ldots, cp[N]/0] && (\text{by } \ddagger)\\
&\quad= \mathbf{4} + [N > 0] \cdot \Big(4N + 2\, \textstyle\sum_{\ell=1}^{N-1} \Big(\sum_{k=0}^{\infty} \big(\frac{\ell}{N}\big)^k\Big)\Big)\\
&\quad= \mathbf{4} + [N > 0] \cdot \Big(4N + 2\, \textstyle\sum_{\ell=1}^{N-1} \frac{N}{\ell}\Big) && \begin{pmatrix}\text{geom. series and}\\ \text{sum reordering}\end{pmatrix}\\
&\quad= \mathbf{4} + [N > 0] \cdot 2N \cdot (\mathbf{2} + \mathcal{H}_{N-1})\ ,
\end{aligned}
$$

where $\mathcal{H}_{N-1} \triangleq 0 + 1/1 + 1/2 + 1/3 + \cdots + 1/N{-}1$ denotes the $(N{-}1)$-th harmonic number. Since the harmonic numbers approach asymptotically to the natural logarithm, we conclude that the coupon collector algorithm C_{cp} runs in expected time $\Theta(N \cdot \log(N))$.

8 Conclusion

We have studied a wp–style calculus for reasoning about the expected run–time and positive almost–sure termination of probabilistic programs. Our main contribution consists of several sound and complete proof rules for obtaining upper as well as lower bounds on the expected run–time of loops. We applied these rules to analyze the expected run–time of a variety of example programs including the well-known coupon collector problem. While finding invariants is, in general, a challenging task, we were able to guess correct invariants by considering a few loop unrollings most of the time. Hence, we believe that our proof rules are natural and widely applicable.

Moreover, we proved that our approach is a conservative extension of Nielson's approach for reasoning about the run–time of deterministic programs and that our calculus is sound with respect to a simple operational model.

Acknowledgement. We thank Gilles Barthe for bringing to our attention the coupon collector problem as a particularly intricate case study for formal verification of expected run–times and Thomas Noll for bringing to our attention Nielson's Hoare logic.

References

1. Arthan, R., Martin, U., Mathiesen, E.A., Oliva, P.: A general framework for sound and complete Floyd-Hoare logics. ACM Trans. Comput. Log. 11(1) (2009)
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
3. Berghammer, R., Müller-Olm, M.: Formal development and verification of approximation algorithms using auxiliary variables. In: Logic Based Program Synthesis and Transformation (LOPSTR). LNCS, vol. 3018, pp. 59–74. Springer (2004)

4. Brázdil, T., Kiefer, S., Kucera, A., Vareková, I.H.: Runtime analysis of probabilistic programs with unbounded recursion. J. Comput. Syst. Sci. 81(1), 288–310 (2015)
5. Celiku, O., McIver, A.: Compositional specification and analysis of cost-based properties in probabilistic programs. In: Formal Methods (FM). LNCS, vol. 3582, pp. 107–122. Springer (2005)
6. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Computer Aided Verification (CAV). LNCS, vol. 8044, pp. 511–526. Springer Berlin Heidelberg (2013)
7. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
8. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: Principles of Programming Languages (POPL). pp. 489–501. ACM (2015)
9. Frandsen, G.S.: Randomised algorithms (1998), Lecture Notes, University of Aarhus, Denmark
10. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering (FOSE). pp. 167–181. ACM (2014)
11. Hehner, E.C.R.: Formalization of time and space. Formal Aspects of Computing 10(3), 290–306 (1998)
12. Hehner, E.C.R.: A probability perspective. Formal Aspects of Computing 23(4), 391–419 (2011)
13. Hickey, T., Cohen, J.: Automating program analysis. J. ACM 35(1), 185–220 (1988)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
15. Hurd, J.: A formal approach to probabilistic termination. In: Theorem Proving in Higher Order Logics (TPHOL), LNCS, vol. 2410, pp. 230–245. Springer Berlin Heidelberg (2002)
16. Kaminski, B.L., Katoen, J.: On the hardness of almost-sure termination. In: Mathematical Foundations of Computer Science (MFCS), Part I. LNCS, vol. 9234, pp. 307–318. Springer (2015)
17. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. ArXiv e-prints (2016), `http://arxiv.org/abs/1601.01001`
18. Kozen, D.: Semantics of Probabilistic Programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
19. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer (2004)
20. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press (2005)
21. Monniaux, D.: An abstract analysis of the probabilistic termination of programs. In: Symposium on Static Analysis (SAS). Lecture Notes in Computer Science, vol. 2126, pp. 111–126. Springer (2001)
22. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
23. Nielson, H.R.: A Hoare-like proof system for analysing the computation time of programs. Sci. Comput. Program. 9(2), 107–136 (1987)
24. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science, Springer (2007)
25. Wechler, W.: Universal Algebra for Computer Scientists, EATCS Monographs on Theoretical Computer Science, vol. 25. Springer (1992)
26. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press (1993)