6

Expected Runtime Analysis by Program Verification

Benjamin Lucien Kaminski, Joost-Pieter Katoen and Christoph Matheja
RWTH Aachen University

Abstract: This chapter is concerned with analysing the expected runtime of probabilistic programs by exploiting program verification techniques. We introduce a weakest pre-conditioning framework à la Dijkstra that enables to determine the expected runtime in a compositional manner. Like weakest pre-conditions, it is a reasoning framework at the syntax level of programs. Applications of the weakest pre-conditioning framework include determining the expected runtime of randomised algorithms, as well as determining whether a program is positive almost-surely terminatiing, i.e., whether the expected number of computation steps until termination is finite for every possible input. For Bayesian networks, a restricted class of probabilistic programs, we show that the expected runtime analysis can be fully automated. In this way, the simulation time under rejection sampling can be determined. This is in particular useful for ill-conditioned inference queries.

6.1 Introduction

In 1976, Michael Rabin published his paper titled *Randomized Algorithms* in which he describes a method for solving the *closest-pair problem* in computational geometry (Rabin, 1976). This work is today considered *the* seminal paper on randomized algorithms (Smid, 2000). While a naïve deterministic brute-force approach takes quadratic time, Rabin's randomized algorithm solves the closest-pair problem in *expected linear time*.

One year later, in 1977, Robert Solovay and Volker Strassen presented a randomized primality test that decides in *polynomial time* whether a given number is either composite or probably prime, thus proving that primality testing is in the complexity class coRP (Solovay and Strassen, 1977). In 1992, Leonard Adleman and Ming-Deh Huang further reduced the complexity of primality testing

[a] From *Foundations of Probabilistic Programming*, edited by Gilles Barthe, Joost-Pieter Katoen and Alexandra Silva published 2020 by Cambridge University Press.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

to ZPP, thus proving that primality testing can be solved efficiently in expectation (Adleman and Huang, 1992). Turning an inefficient deterministic algorithm into a randomized algorithm that is – in expectation – more efficient (possibly at the cost of incorrect results, though with low probability) is a principal motivation of introducing randomization into the computation. Prime examples are Freivalds' matrix multiplication verification (Freivalds, 1979) or Hoare's variant of quicksort with random pivot selection (Hoare, 1962). Some problems even inherently require randomized solutions such as various self-stabilization algorithms in anonymous distributed systems.

Probabilistic programs are, however, not limited to randomized algorithms. In fact, they are a powerful modeling formalism for describing, amongst others, graphical models, such as Bayesian networks or Markov random fields (Koller and Friedman, 2009). This lead to the emergence of *probabilistic programming* (Gordon et al., 2014) as a new paradigm for probabilistic modeling. A key feature of probabilistic programming languages is that they decouple individual models from algorithms for their analysis, e.g. Bayesian inference techniques. For example, one of the first approaches to perform inference on probabilistic programs first compiles a program into a Bayesian network and then applies standard techniques for graphical models (Minka and Winn, 2017). Probabilistic programming also enables to resort to established program analysis techniques, such as slicing (Hur et al., 2014), to optimize probabilistic models. In this context, analyzing expected runtimes, i.e. the expected time required for sampling from a complex probability distribution described by a probabilistic program, to speed up inference algorithms is of paramount importance, too.

Reasoning about expected runtimes of probabilistic programs is surprisingly subtle and full of nuances as we will discuss in detail in this chapter. Thus, there is a desire for *a formal verification technique* suited for reasoning about expected runtimes. The main objective of this chapter is to provide a gentle introduction to one particular formal method for analyzing expected runtimes of probabilistic programs: The *expected runtime calculus*. This approach was originally developed in Kaminski et al. (2016) and was further studied in Olmedo et al. (2016); Batz et al. (2018); Kaminski et al. (2018b); Kaminski (2019). The calculus is a weakest-precondition style calculus à la Dijkstra (see Dijkstra, 1976) to derive runtime assertions. In a similar vein to Dijkstra's predicate transformers, our calculus uses *runtime transformers*. Its core is the expected runtime transformer ert:

$$\mathsf{ert}[c](t)(\sigma)$$

captures the expected runtime of program c when started in initial state σ. The t appearing above is the so-called *postruntime*: t is a function mapping program states to non-negative reals and captures the expected runtime of the computation

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

following program c. It is hence evaluated in the *final* states reached after termination of c on σ. In particular, this subsumes the plain expected runtime of program c on initial state σ if t is the constantly zero runtime. For most control structures, ert is defined in a straightforward compositional manner. The action of the transformer on loops is given using fixed-point techniques. To avoid the tedious reasoning about such fixed points and to enhance the calculus' usability, we provide *invariant-based proof rules* that establish bounds on the expected runtime of loops.

A notable feature of the ert-calculus is that it firmly builds upon standard techniques from formal semantics and program verification – in particular denotational semantics, fixed point theory, and invariants. It provides a useful abstraction from the semantical intricacies of probabilistic programs and the underlying probability theory. ert thus enables writing elegant and *compositional* proofs to bound expected runtimes (from above and below) *on source code level*. Furthermore, the reliance on standard techniques makes ert amenable to a large degree of automation. The ert-calculus yields *comprehensible* proofs for the expected runtime of complex randomized algorithms. For instance, it has been successfully applied to analyze the Coupon Collector's problem (Kaminski et al., 2016), a Sherwood binary search (Olmedo et al., 2016), and expected sampling times in Bayesian networks (Batz et al., 2018). The latter can be derived fully automatically.

Ngo et al. (2018) have developed an automatic approach for deriving polynomial runtime bounds, using our ert calculus as an underlying theoretical framework for proving soundness of their approach. The ert calculus has also been mechanized in the interactive theorem prover Isabelle/HOL by Hölzl (2016). In particular, Hölzl proved that our calculus is indeed sound and complete and that our proof rules for deriving runtime bounds are correct.

A second asset is that ert enables determining whether the expected runtime of a randomized algorithm (for all possible inputs) is finite or not. To the best of our knowledge, this is the first formal verification framework that can handle both almost-sure termination (does a program terminate with probability one?) and positive almost-sure termination (does a program terminate within finite expected time?). The universal positive almost-sure termination problem is complete for level Π^0_3 of the arithmetical hierarchy (Kaminski and Katoen, 2015) and hence strictly harder to decide than the universal halting problem for deterministic programs (which is Π^0_2-complete).

Organization of this chapter. We describe a simple probabilistic programming language in Section 6.2. In Section 6.3, we then present challenges and phenomena encountered when reasoning about expected runtimes. The expected runtime calculus and our proof rules for loops are presented in Section 6.4. An application of our calculus to the automated analysis of expected sampling times of Bayesian

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

networks is presented in Section 6.6. Finally, in Section 6.7, we conclude with a discussion of recent research directions.

6.2 Probabilistic Programs

In this chapter, we consider probabilistic programs written in a simple probabilistic extension of Dijkstra's Guarded Command Language (GCL) (Dijkstra, 1976). To that end, we extend GCL with *random assignments*. Let us briefly go over the statements in the resulting *probabilistic* Guarded Command Language (pGCL) by means of small examples. Furthermore, since we want to reason about (expected) runtimes of pGCL programs, we also discuss our underlying runtime model, i.e. the time consumed by each pGCL statement. Our pGCL programs adhere to the grammar

$$\begin{aligned} C \quad \longrightarrow \quad & \texttt{empty} \mid x :\approx \mu \mid C\,;\, C \\ & \mid \texttt{if}\,(\varphi)\ \{C\}\ \texttt{else}\ \{C\} \\ & \mid \{c_1\} \,\square\, \{c_2\} \\ & \mid \texttt{while}\,(\varphi)\ \{C\} \end{aligned}$$

where `empty` is a program that has no effect and consumes no time, x is a program variable, μ represents a (discrete) probability distribution, and φ a Boolean expression. We now go over the language constructs and the runtime model. For more details, in particular on an operational semantics capturing our runtime model, please refer toKaminski et al. (2016).

Random assignments. The key feature of pGCL is the ability to sample values from a (discrete) probability distribution, say μ, and assign the sampled value to a program variable, say x. The corresponding pGCL statement is a *random assignment* of the form

$$x :\approx \mu\,.$$

For example, the random assignment

$$\mathit{heads} :\approx {}^1\!/_3 \cdot \langle \mathsf{true} \rangle + {}^2\!/_3 \cdot \langle \mathsf{false} \rangle$$

simulates a biased coin flip: With probability $1/3$ the value true is assigned to variable *heads* and with the remaining probability, i.e. $2/3$, the value false is assigned, respectively. The right-hand side of a random assignment may be any (computable) discrete probability distribution over the set of possible values of a variable. In particular, we allow probability distributions to depend on the current program state, i.e. an evaluation of all program variables. For instance, the random assignment

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$y :\approx \texttt{uniform}(0, x)$ samples from a (discrete) uniform distribution over the range from 0 to the current value stored in variable x and assigns the thereby obtained value to variable y. Notice that deterministic assignments, such as $x :\approx x + 1$, are a special case in which a probability of one is assigned to a single value.

In our runtime model, every random assignment consumes *one* unit of time. Note that this is a design choice in order to keep the calculus we are going to present as clean and simple as possible. It is unproblematic to assume a more nuanced runtime model for random assignments. The same holds for the runtimes we associate with the other language constructs below.

Control flow. pGCL is equipped with standard control flow constructs for sequential composition, conditional choices, and loops:

- The *sequential composition* $c_1\,;\ c_2$ first executes program c_1 and then executes program c_2. The composition operation itself consumes *no* time.
- The *conditional choice* $\texttt{if}\,(\varphi)\ \{c_1\}\ \texttt{else}\ \{c_2\}$ executes c_1 if the (deterministic) guard φ evaluates to true and c_2 if φ evaluates to false, respectively. The guard evaluation consumes *one* unit of time.
- The *loop* $\texttt{while}\,(\varphi)\ \{c\}$ keeps executing the loop body c as long as the guard φ evaluates to true at the loop head. If the guard evaluates to false, the loop terminates. *Every* guard evaluation consumes *one* unit of time.

Let us consider the following probabilistic program inspired by Chakarov and Sankaranarayanan (2013):

$$
\begin{array}{l}
h :\approx 0;\ t :\approx 30;\\
\texttt{while}\,(h \leq t)\,\{\\
\qquad c :\approx {}^1\!/_2 \cdot \langle \mathsf{true} \rangle + {}^1\!/_2 \cdot \langle \mathsf{false} \rangle;\\
\qquad \texttt{if}\,(c = \mathsf{true})\,\{\\
\qquad\qquad h :\approx h + \mathbf{uniform}[0 \ldots 10]\\
\qquad \}\ \texttt{else}\ \{\texttt{empty}\};\\
\qquad t :\approx t + 1\\
\}\,.
\end{array}
$$

Here, `empty` is a pGCL statement representing the empty program, i.e. the statement has *no effect* and consumes *no time*. The example models a race between a tortoise and a hare; the variables t and h represent their respective positions. The tortoise starts with a lead of 30 and advances one step in each round, i.e. each loop iteration. The hare with probability $1/2$ advances a random number of steps between 0 and 10

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

(governed by a uniform distribution) and with the remaining probability remains still. The race ends when the hare passes the tortoise.

Regarding the runtime, the program requires two units of time for the initial assignments. In every loop iteration, the program consumes either four or five units of time: It always takes one unit of time to evaluate the loop guard, flip a coin, evaluate the conditional, and to update variable t, respectively. If the conditional is evaluated to true, an additional unit of time is consumed to update the value of variable h.

Nondeterminism. Apart from sampling from a known distribution, pGCL also supports true nondeterminism: The statement $\{c_1\} \,\square\, \{c_2\}$ represents a nondeterministic choice between programs c_1 and c_2, i.e. either c_1 or c_2 is executed, but there is no probability distribution underlying the choice between the two programs. Similarly to sequential composition, a nondeterministic choice itself consumes no additional time in our runtime model.

As an example, consider the program

$$
\begin{aligned}
&\{x :\approx 3\} \,\square\, \{x :\approx 5\}\,;\\
&\texttt{while}\,(x > 0)\,\{\\
&\qquad x :\approx x - 1\\
&\}\,.
\end{aligned}
$$

This program has two possible executions: One execution initially assigns 3 to x and has a runtime of 8 units of time. The other execution initially assigns 5 to x and has a runtime of 12 units of time. When reasoning about *the* expected runtime of a pGCL program, we resolve nondeterminism by a *demonic* scheduler (cf. McIver and Morgan, 2004; Dijkstra, 1976). That is, nondeterministic choices are resolved in a way that *maximizes* the runtime. In the above example, this means that the nondeterministic choice is resolved such that 5 is assigned to x. Strictly speaking, we thus reason about *worst-case* expected runtimes.

Remark on the runtime model. We stress that the runtime model presented in this section is one particular design choice for the sake of concreteness. It is straightforward to adapt our approach to alternative runtime models, where, for example, only loop iterations or assignments are considered relevant. The same holds for alternative resolutions of nondeterminism such as using an angelic scheduler. Furthermore, more fine-grained models that take, for instance, the size of expressions and distributions that appear in the random assignments into account can easily be incorporated.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

6.3 Semantic Intricacies

Reasoning about expected runtimes of probabilistic programs is full of nuances. Let us illustrate this by discussing a few phenomena. To this end, we consider a fundamental property of ordinary, i.e. non-probabilistic, programs: *An ordinary program terminates, meaning all of its runs are finite, if and only if it has a finite runtime.* What is a probabilistic analog of this property when considering *expected* runtimes?

Termination is too strong. A single diverging run of an ordinary program causes its runtime to be infinite. This is not the case for probabilistic programs. They may admit *arbitrarily long* and even *infinite* runs while still having a *finite* expected runtime. For example, the program

$$
\begin{aligned}
c_{geo}\colon\quad & b :\approx 1;\\
& \texttt{while}(b = 1)\{\\
& \qquad b :\approx {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle\\
& \}\,.
\end{aligned}
$$

keeps flipping a fair coin until observing the first heads (represented by 0). It admits arbitrarily long runs, since – for every natural number n – the probability of not seeing a heads in the first n trials is non-zero. It even admits a non-terminating run, namely the one in which the outcome of all coin flips is tails. The runtime of program c_{geo}, however, is geometrically distributed and therefore its expected runtime is finite, even constant: On average, it terminates after two loop iterations.

The classical notion of termination for ordinary programs – all program runs have to be finite – is thus too strong for probabilistic programs (with respect to the considered property). In the above example, we observe that the only infinite run of program c_{geo} has probability zero. A more sensible notion of termination might thus require that all infinite runs of a program have probability zero. Conversely: The probability of termination is one. This is referred to as *almost-sure termination* (Hart et al., 1983). Does almost-sure termination – instead of classical termination – capture finite expected runtimes?

Almost-sure termination is too weak. For ordinary programs, termination always implies finite runtime. For probabilistic programs this is not always true – even if we consider almost-surely terminating programs only. For example, consider the program

$$
\begin{aligned}
c_{rw}\colon\quad & x :\approx 10;\\
& \texttt{while}\,(x > 0)\{
\end{aligned}
$$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$$x :\approx 1/2 \cdot \langle x-1 \rangle + 1/2 \cdot \langle x+1 \rangle$$
$$\}\,,$$

which models a one-dimensional random walk of a particle: Starting from position 10, in each step the particle moves randomly one step to the left or one step to the right, until reaching position 0. The particle reaches position 0 with probability one. The program c_{rw} thus terminates almost-surely. However, doing so requires infinitely many steps on average (cf. Ibe, 2013, Chapter 3.7.3). The expected runtime of c_{rw} is thus infinite.

Since an almost-surely terminating program might run – in expectation – infinitely long, a better probabilistic analog of classical termination might be to require that a program's expected runtime is finite. This is referred to as *positive* almost-sure termination (Bournez and Garnier, 2005). In fact, having a finite expected runtime implies almost-sure termination (Olmedo et al., 2016, Theorem 5.3). However, there are subtle differences between classical termination and positive almost-sure termination. From a complexity-theoretic view, it is noteworthy that the decision problem "does a program terminate in finite expected time (on all inputs)?" is Π^0_3-complete in the arithmetical hierarchy, and thus strictly harder than the universal halting problem for ordinary programs (Kaminski and Katoen, 2015; Kaminski et al., 2018a).

Positive almost-sure termination is not compositional. Running two ordinary terminating programs in sequence yields again a terminating program. Termination is thus closed under sequential composition. This is not true for probabilistic programs when considering positive almost-sure termination. Consider the pair of programs

c_1: $x :\approx 1;\ b :\approx 1;$
`while`$(b = 1)\{$
$\quad b :\approx 1/2 \cdot \langle 0 \rangle + 1/2 \cdot \langle 1 \rangle;$
$\quad x :\approx 2x$
$\}$

c_2: `while`$(x > 0)\{$
$\quad x :\approx x - 1$
$\}$

Both programs terminate positive almost-surely: As the loop in C_1 terminates on average in two iterations, it has a finite expected runtime. Furthermore, program c_2 performs at most $\lceil x \rceil$ iterations for any initial value of x, i.e. its expected runtime is finite, too. However, the composed program $c_1\,;\,c_2$ has an *infinite* expected runtime – even though it almost-surely terminates. This is intuitively due to the fact that the expected value of x after termination of c_1 is infinite and c_2 needs x steps to terminate.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

6.4 The Expected Runtime Calculus

We now present a sound and complete calculus that enables rigorous reasoning about expected runtimes of probabilistic programs. Apart from programs, the two central objects used within our calculus are program states and runtimes. A *program state* σ is an evaluation of program variables, i.e. a mapping from variables – collected in the set Var – to possible values, such as integers or rationals, which are collected in the set Val. A *runtime* is a function mapping every program state σ to a non-negative real number or infinity. Formally, the sets of program states and runtimes are given by

$$\Sigma \;=\; \{\sigma \colon \mathsf{Var} \to \mathsf{Val}\} \quad \text{and} \quad \mathbb{T} \;=\; \{t \colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}\,.$$

Our goal is to associate with every program c a runtime t mapping every program state σ to the average or expected runtime of executing program c on initial state σ. To this end, we express the expected runtime of programs in a continuation-passing style by means of the runtime transformer

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

The number $\mathsf{ert}[c](t)(\sigma)$ is the expected runtime of executing program c on initial state σ assuming that t captures the runtime of the computation *following* c. The ert-transformer thus applies backward reasoning. The runtime t is usually referred to as the *continuation* (or postruntime) and we can think of it as being evaluated in the final states that are reached upon termination of c. Thus, the plain expected runtime of executing c on initial state σ is $\mathsf{ert}[c](0)(\sigma)$, where 0 is a shortcut for the constant runtime $\lambda\sigma.\, 0$.[1] In general, we write k to denote the constant runtime $\lambda\sigma.\, k$ for $k \in \mathbb{R}^{\infty}_{\geq 0}$.

The ert-transformer is defined by induction on the structure of pGCL programs and adheres to our simple runtime model described in Section 6.2. That is, $\mathsf{ert}[c](0)$ captures the expected number of assignments and guard evaluations. A summary of the ert definitions is found in Table 6.1. Let us briefly go over the definitions for each pGCL statement.

Empty program. Since the empty program `empty` has no effect on the program state and consumes no time, the expected runtime of `empty` with respect to continuation t corresponds to the identity, i.e.

$$\mathsf{ert}[\texttt{empty}](t) \;=\; t\,.$$

[1] We use λ-expressions to denote functions: Function $\lambda X.\, f$ applied to an argument α evaluates to f in which every free occurrence of X is replaced by α.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Table 6.1 *Rules for defining the expected runtime transformer* ert.

c	$\mathsf{ert}\,[c]\,(t)$
`empty`	t
$x :\approx \mu$	$1 \;+\; \lambda\sigma.\ \sum_{v \in \mathsf{Val}} \mathsf{Pr}_{[\![\mu]\!](\sigma)}(v) \cdot t[x/v](\sigma)$
$c_1 \,;\, c_2$	$\mathsf{ert}[c_1](\mathsf{ert}[c_2](t))$
`if` (φ) $\{c_1\}$ `else` $\{c_2\}$	$1 \;+\; [\varphi] \cdot \mathsf{ert}[c_1](t) \;+\; [\neg\varphi] \cdot \mathsf{ert}[c_2](t)$
$\{c_1\} \,\square\, \{c_2\}$	$\max\{\mathsf{ert}[c_1](t), \mathsf{ert}[c_2](t)\}$
`while` (φ) $\{c'\}$	$\mathsf{lfp}\, F_t$, where
	$F_t(X) \;=\; 1 \;+\; [\varphi] \cdot \mathsf{ert}[c'](X) \;+\; [\neg\varphi] \cdot t$

Random assignment. For the random assignment $x :\approx \mu$, one unit of time is consumed. Moreover, the remaining expected runtime represented by continuation t has to be considered after updating t to take the possible new values of variable x – weighted according to their probabilities – into account. Formally, we define

$$\mathsf{ert}[x :\approx \mu](t)(\sigma) \;=\; 1 + \sum_{v \in \mathsf{Val}} \mathsf{Pr}_{[\![\mu]\!](\sigma)}(v) \cdot t\big(\sigma[x/v]\big)\,,$$

where $\mathsf{Pr}_{[\![\mu]\!](\sigma)}(v)$ is the probability that – for state σ – distribution expression μ evaluates to value v. Moreover, $\sigma[x/v]$ is the program state σ in which the value assigned to variable x is updated to v. The above definition does, unfortunately, depend on the program state σ. If the distribution expression μ is agnostic of the program state, we can obtain a simpler definition. To this end, we define the "syntactic replacement" of a variable x in a runtime t by value v as $t[x/v] = \lambda\sigma.\, t(\sigma[x/v])$. Then, since the probability distribution μ is independent of σ, we can write the expected runtime of the random assignment without referring explicitly to a program state:

$$\mathsf{ert}[x :\approx \mu](t) \;=\; 1 + \sum_{v \in \mathsf{Val}} \mathsf{Pr}_{[\![\mu]\!]}(v) \cdot t[x/v]$$

For example, consider a biased coin flip $x :\approx {}^1\!/_3 \cdot \langle 0\rangle + {}^2\!/_3 \cdot \langle x+1\rangle$. Since the probability distribution does not depend on the program state, we have

$$\mathsf{ert}\big[x :\approx {}^1\!/_3 \cdot \langle 0\rangle + {}^2\!/_3 \cdot \langle x+1\rangle\big](t) \;=\; 1 \;+\; {}^1\!/_3 \cdot t[x/0] \;+\; {}^2\!/_3 \cdot t[x/x+1]\,.$$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

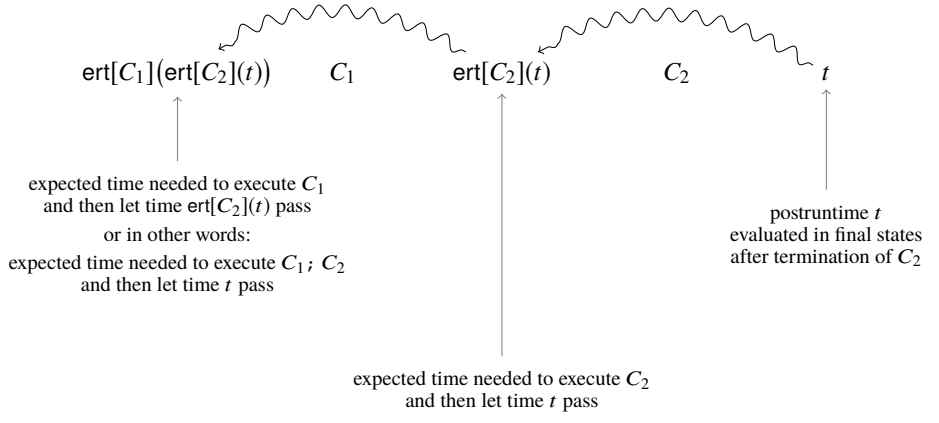

Figure 6.1 Continuation-passing style expected runtime transformer.

Sequential composition. For the composition $c_1\,;\,c_2$ of pGCL programs c_1 and c_2, it becomes evident that we reason backwards: We first determine the expected runtime of c_2 with respect to continuation t, i.e. $\mathsf{ert}[c_2](t)$. Afterwards, we determine the expected runtime of c_1 with respect to continuation $\mathsf{ert}[c_2](t)$. As a diagram, the intuition behind this is depicted in Figure 6.1. Formally, we define the ert of sequential composition as

$$\mathsf{ert}[c_1\,;\,c_2](t) \;=\; \mathsf{ert}[c_1](\mathsf{ert}[c_2](t))\,.$$

Conditional choice. For the conditional $\texttt{if}\,(\varphi)\,\{c_1\}\,\texttt{else}\,\{c_2\}$, one unit of time is consumed to account for the guard evaluation. Furthermore, the expected runtime of c_1 (with respect to continuation t) is added if guard φ evaluates to true. Otherwise, the expected runtime of c_2 is added. The truth value of a Boolean expression ξ is captured by the indicator function, also called *Iverson bracket*,

$$[\xi] : \Sigma \to \{0,1\}\,,$$

which, for a given state σ, evaluates to 1 if ξ evaluates to true in state σ. Otherwise, it evaluates to 0. Using Iverson brackets, the ert of the conditional choice statement is given by

$$\mathsf{ert}\big[\texttt{if}\,(\varphi)\,\{c_1\}\,\texttt{else}\,\{c_2\}\big](t) \;=\; 1 \;+\; [\varphi]\cdot\mathsf{ert}[c_1](t) \;+\; [\neg\varphi]\cdot\mathsf{ert}[c_2](t)\,.$$

Nondeterminism. For the *nondeterministic choice* $\{c_1\}\,\square\,\{c_2\}$, either c_1 or c_2 is executed. In particular, no probability distribution guiding which of the two programs is executed is known. Since the choice itself consumes no time, the

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

expected runtime of $\{c_1\} \,\square\, \{c_2\}$ is either the expected runtime of c_1 or the expected runtime of c_2. Following the "demonic nondeterminism" school of thought (cf. McIver and Morgan, 2004; Dijkstra, 1976), we assume that the program with the worst expected runtime will be executed, i.e. we take the maximum of both expected runtimes. Formally, we define

$$\mathsf{ert}\big[\, \{c_1\} \,\square\, \{c_2\} \,\big](t) \;=\; \max\big\{\mathsf{ert}[c_1](t), \mathsf{ert}[c_2](t)\big\} \;.$$

Loops. For the loop $\texttt{while}\,(\varphi)\,\{c\}$, we exploit the fact that the expected runtime of the loop coincides with the expected runtime of its unrolling

$$\texttt{if}\,(\varphi)\,\big\{c\texttt{;}\;\texttt{while}\,(\varphi)\,\{c\}\big\}\;\texttt{else}\;\{\texttt{empty}\}\;.$$

Applying ert to this program then yields (for a fixed continuation t):

$$\begin{aligned}
&\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t)\\
=\;&\mathsf{ert}[\texttt{if}\,(\varphi)\,\{c\texttt{;}\;\texttt{while}\,(\varphi)\,\{c\}\}\;\texttt{else}\;\{\texttt{empty}\}](t)\\
=\;&1 + [\varphi]\cdot\mathsf{ert}[c\texttt{;}\;\texttt{while}\,(\varphi)\,\{c\}](t) && \text{(Definition of ert)}\\
&\qquad + [\neg\varphi]\cdot\mathsf{ert}[\texttt{empty}](t)\\
=\;&1 + [\varphi]\cdot\mathsf{ert}[c]\big(\mathsf{ert}\big[\texttt{while}\,(\varphi)\,\{c\}\big](t)\big) + [\neg\varphi]\cdot t\;. && \text{(Definition of ert)}
\end{aligned}$$

Every solution of this equation is a fixed point of the transformer

$$F_t\colon \qquad \mathbb{T}\to\mathbb{T},\quad X \;\mapsto\; 1 \;+\; [\varphi]\cdot\mathsf{ert}[c](X) \;+\; [\neg\varphi]\cdot t\;,$$

in which we substituted $\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t)$ by X. In fact, as is standard in denotational semantics, we are interested in the *least* fixed point. The underlying intuition is that, for every natural number n,

$$F_t^n(0) \;=\; F_t(F_t(\ldots(0)\ldots))$$

precisely captures the expected runtime when allowing at most n loop iterations. Consequently, we approximate the expected runtime of the loop from below and take the first solution capturing the expected runtime for all loop iterations, i.e. the least fixed point of F_t. The least fixed point of the characteristic functional F_t is guaranteed to exist for every loop $\texttt{while}\,(\varphi)\,\{c\}$ and every continuation t, because $\mathbb{T}$ (with pointwise ordering of runtimes) is a complete lattice and F_t is continuous (and hence also monotonic). The Kleene Fixed Point Theorem (Kleene, 1952) then ensures by continuity that the least fixed point of F_t, which we denote by $\mathsf{lfp}\,F_t$, exists and coincides with the limit of $F_t^n(0)$ for $n\to\infty$. Hence, we define the expected runtime of loop $\texttt{while}\,(\varphi)\,\{c\}$ with respect to continuation t as

$$\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t) \;=\; \mathsf{lfp}\,F_t \;=\; \lim_{n\to\infty} F_t^n(0)\;.$$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Remark on soundness. Our expected runtime calculus is sound with respect to a simple operational semantics based on Markov Decision Processes, where each state corresponds to a program statement and each transition corresponds to one execution step. The time consumed by a statement is modeled using state rewards. More precisely, one can show that for every pGCL program c, every continuation t and every program state σ, the expected runtime $\mathsf{ert}[c](t)(\sigma)$ coincides with the maximal expected reward (assuming demonic nondeterminism) of the unique Markov Decision Process corresponding to c, t, and σ as defined by the operational semantics. We refer the interested reader to Kaminski et al. (2016) for further details.

A loop-free example. Consider the program $c_{\texttt{trunc}}$:

c_1: $\quad x :\approx 1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle;$
c_2: $\quad \texttt{if}(x = \mathsf{true})\{$
c_3: $\quad\quad x :\approx 1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle$
c_4: $\quad\quad \texttt{if}(x = \mathsf{true})\{$
c_5: $\quad\quad\quad x :\approx 1/2 \cdot \langle \mathsf{true} \rangle + 1/2 \cdot \langle \mathsf{false} \rangle$
$\quad\quad \}\,\texttt{else}\,\{\texttt{empty}\}$
$\quad \}\,\texttt{else}\,\{\texttt{empty}\}$

It simulates a geometric distribution that is truncated after the third coin flip. To determine its expected runtime, i.e. $\mathsf{ert}[c_{\texttt{trunc}}](0)$, it suffices to apply the rules of the ert-transformer (see Table 6.1). Throughout this chapter, we will use the notation

$$
\begin{array}{l}
/\!/\!/\ s' \\
/\!/\!/\ s \\
c \\
/\!/\!/\ t
\end{array}
$$

to express the fact that $s = \mathsf{ert}[c](t)$ and moreover that $s' = s$. It is thus more intuitive to read annotated programs from *bottom to top*, just like the ert-transformer moves from the back to the front. Using this notation, we can annotate the program $c_{\texttt{trunc}}$ simply by applying the ert rules from Table 6.1 as shown below in Figure 6.2. Hence, on average, running program $c_{\texttt{trunc}}$ takes $13/4$ units of time.

6.5 Proof Rules for Loops

Reasoning about the runtime of loop-free programs, such as the program $c_{\texttt{trunc}}$ in the above example, amounts mostly to syntactic reasoning. The runtime of a loop, however, is defined as the least fixed point of its characteristic functional F_t. It can

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$/\!/\!/\ \frac{13}{4}$
$/\!/\!/\ \ldots$
$/\!/\!/\ 1 + \frac{1}{2} \cdot \big(1 + [x = \text{true}] \cdot \frac{5}{2}\big)[x/\text{true}]$
$\qquad + \frac{1}{2} \cdot \big(1 + [x = \text{true}] \cdot \frac{5}{2}\big)[x/\text{false}]$
$x :\approx 1/2 \cdot \langle\text{true}\rangle + 1/2 \cdot \langle\text{false}\rangle;$
$/\!/\!/\ 1 + [x = \text{true}] \cdot \frac{5}{2}$
$/\!/\!/\ 1 + [x = \text{true}] \cdot \frac{5}{2} + [x = \text{false}] \cdot 0$
`if`$(x = \text{true})\{$
  $/\!/\!/\ \frac{5}{2}$
  $/\!/\!/\ 1 + \frac{1}{2} \cdot (1 + 1) + \frac{1}{2} \cdot (1 + 0)$
  $/\!/\!/\ 1 + \frac{1}{2} \cdot \big(1 + [\text{true} = \text{true}]\big) + \frac{1}{2} \cdot \big(1 + [\text{false} = \text{true}]\big)$
  $/\!/\!/\ 1 + \frac{1}{2} \cdot \big(1 + [x = \text{true}]\big)[x/\text{true}]$
  $\qquad + \frac{1}{2} \cdot \big(1 + [x = \text{true}]\big)[x/\text{false}]$
  $x :\approx 1/2 \cdot \langle\text{true}\rangle + 1/2 \cdot \langle\text{false}\rangle$
  $/\!/\!/\ 1 + [x = \text{true}]$
  $/\!/\!/\ 1 + [x = \text{true}] \cdot 1 + [x = \text{false}] \cdot 0$
  `if`$(x = \text{true})\{$
    $/\!/\!/\ 1$
    $/\!/\!/\ 1 + \frac{1}{2} \cdot 0[x/\text{true}] + \frac{1}{2} \cdot 0[x/\text{false}]$
    $x :\approx 1/2 \cdot \langle\text{true}\rangle + 1/2 \cdot \langle\text{false}\rangle$
    $/\!/\!/\ 0$
  `}else{`
    $/\!/\!/\ 0$
    `empty`
    $/\!/\!/\ 0$
  `}`
  $/\!/\!/\ 0$
`}else{`
  $/\!/\!/\ 0$
  `empty`
  $/\!/\!/\ 0$
`}`
$/\!/\!/\ 0$

Figure 6.2 Runtime annotations for the program c_{trunc}. It is more intuitive to read the annotations from bottom to top. The postruntime after executing the whole program is 0; the a priori expected runtime of executing the whole program is $13/4$.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

thus be obtained by fixed point iteration using Kleene's fixed point theorem (Kleene, 1952), i.e.

$$\mathsf{lfp}\, F_t \;=\; \sup_{n\in\mathbb{N}} F_t^n(0)\,,$$

where $F_t^n(X)$ denotes n-fold application of F_t to X. However, the fixed point is not necessarily reached within a finite number of iterations. We thus study various proof rules for approximating the expected runtime of loops.

6.5.1 Proof Rule for Upper Bounds

We first present a simple, yet powerful, proof rule for determining upper bounds on expected runtimes of loops. This rule is based on an alternative characterization of the least fixed point due to Tarski and Knaster (see Tarski *et al.*, 1955): Any X that satisfies $F(X) \leq X$ is called a *pre-fixed point* of the function F. The least fixed point of F can be characterized as its smallest pre-fixed point, i.e. the smallest X such that $F(X) \leq X$. Consequently, *every* pre-fixed point is greater than or equal to the least fixed point of F.

How does this lead us to a proof rule? Let F_t be the characteristic functional of some loop $\texttt{while}\,(\varphi)\,\{c\}$ with respect to a continuation $t \in \mathbb{T}$. In the context of runtimes, we refer to a pre-fixed point $I \in \mathbb{T}$ of F_t as an *upper invariant* of loop $\texttt{while}\,(\varphi)\,\{c\}$ and continuation t. By the above observation, we can then formulate our first proof rule as follows:

$$\underbrace{F_t(I) \;\leq\; I}_{I \text{ is an upper invariant}} \qquad \text{implies} \qquad \underbrace{\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t) \;=\; \mathsf{lfp}\, F_t \;\leq\; I}_{I \text{ is an upper bound of the expected runtime}}\,.$$

In particular, since the exact expected runtime of a loop itself is an upper invariant, completeness of the above proof rule is immediate.

Example: The geometric distribution. Let us consider an application of our proof rule. The loop c_{geo} below has a geometrically distributed runtime as it keeps flipping a fair coin until it hits tails ($c = 0$).

$$\begin{aligned} c_{\mathsf{geo}}\colon \quad & \texttt{while}\,(c = 1)\,\{ \\ & \quad c \;:\approx\; {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle \\ & \} \end{aligned}$$

How can we apply our proof rule to verify an upper bound on the expected runtime of c_{geo}? The corresponding characteristic functional with respect to postrun-

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

time $t = 0$ is:

$$\begin{aligned} F_0(X) &= 1 + [c \neq 1] \cdot 0 + [c = 1] \cdot \mathsf{ert}[c :\approx {}^1\!/_2 \cdot \langle 0 \rangle + {}^1\!/_2 \cdot \langle 1 \rangle](X) \\ &= 1 + [c = 1] \cdot \Big(1 + \tfrac{1}{2} \cdot \big(X[c/0] + X[c/1]\big)\Big) \,. \end{aligned}$$

By the calculations below we verify that $I = 1 + [c = 1] \cdot 4$ is an upper invariant of the loop (with respect to 0):

$$\begin{aligned} F_0(I) &= 1 + [c = 1] \cdot \Big(1 + \tfrac{1}{2} \cdot \big(I[c/0] + I[c/1]\big)\Big) \\ &= 1 + [c = 1] \cdot \Big(1 + \tfrac{1}{2} \cdot \big(\underbrace{1 + [0 = 1] \cdot 4}_{=1} + \underbrace{1 + [1 = 1] \cdot 4}_{=5}\big)\Big) \\ &= 1 + [c = 1] \cdot 4 \\ &= I \leq I \,. \end{aligned}$$

Hence, I is an upper invariant. By our proof rule, we then obtain

$$\mathsf{ert}\big[c_{\mathsf{geo}}\big](0) \leq 1 + [c = 1] \cdot 4 \,.$$

In words, the expected runtime of c_{geo} is at most $1 + 4 = 5$ from any initial state where $c = 1$ and at most $1 + 0 = 1$ from any other state. Notice that if the loop body is itself loop-free, as in the above example, verifying that some runtime I is an upper invariant is usually fairly easy. Inferring the invariant, in contrast, is one of the most involved part of the verification effort.

6.5.2 Proof Rule for Lower Bounds

It is tempting to use the converse version of our proof rule based on upper invariants to reason about lower bounds on the expected runtime. That is, one would like to check $I \leq F_t(I)$ for some runtime $I \in \mathbb{T}$ in order to verify that $I \leq \mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](0)$. Such a rule is unfortunately *not sound* as is, but we can add further premises to make it sound.

Metering Functions

We would like first to point out that it is not self-evident that the simple lower bound rule suggested above must necessarily fail: For *non-probabilistic* programs, Frohn et al. (2016) have shown that this very lower bound rule is indeed sound. They call an I, such that $I \leq F_t(I)$, a *metering function*. Whenever I is a metering function, I is a lower bound on the runtime of a *non-probabilistic loop*, just as I being an upper invariant implies that I is an upper bound on the (expected) runtime of a loop – be it probabilistic or not.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

The intuition behind the soundness of the metering function rule is that for non-probabilistic programs there exists some $n \in \mathbb{N}$ such that

$$\big(\mathsf{lfp}\ F_t\big)(\sigma) \;=\; \big(F_t^n(0)\big)(\sigma)\,,$$

in case that the loop terminates on σ (this is not true if the loop diverges, but then any lower bound is a lower bound). Existence of n allows for proving soundness of the metering function rule by induction on n. This is *not true* for probabilistic programs: We may well have the situation that we need to take the limit for n, so that

$$\big(\mathsf{lfp}\ F_t\big)(\sigma) \;=\; \sup_{n\in\mathbb{N}}\big(F_t^n(0)\big)(\sigma)\,,$$

but for all $n \in \mathbb{N}$

$$\big(\mathsf{lfp}\ F_t\big)(\sigma) \;>\; \big(F_t^n(0)\big)(\sigma)\,,$$

even for a fixed initial state σ. Indeed, for probabilistic programs, the metering function approach is unfortunately unsound, as the following counterexample shows: Consider the loop c, given by

$$\begin{aligned}&\texttt{while}\,(y = 1)\,\{\\ &\qquad y :\approx {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle;\\ &\qquad x :\approx x + 1\\ &\}\,,\end{aligned}$$

where we assume that x ranges over $\mathbb{N}$ for simplicity. Suppose we want to reason about a lower bound on the expected runtime of c by a metering function. The characteristic functional of the while loop with respect to postruntime 0 is given by

$$F_0(X) \;=\; 1 \;+\; [y = 1] \cdot \Big(2 + \tfrac{1}{2}\big(X[x/x+1][y/0] + X[x/x+1][y/1]\big)\Big)\,.$$

We now propose *two* fixed points of F_0, namely

$$I_1 \;=\; 1 + [y = 1]\,6 \qquad \text{and} \qquad I_2 \;=\; 1 + [y = 1]\,\big(6 + 2^{x+a}\big)\,,$$

for any constant $a \geq 0$, are both a fixed point of F_0 and hence also a metering function. I_1 is in fact the least fixed point of F_0 and we clearly have $I_1 \lneq I_2$. Thus, if we prove $I_2 \leq F_0(I_2)$, we cannot possibly have proven that I_2 is a lower bound on the least fixed point of F_0, since I_1 is a fixed point strictly smaller than I_2. The intuitive reason is that the expected runtime of c is completely independent of x but x has an influence on the value that I_2 assumes. For more details on the connections between the ert calculus and metering functions, we refer the interested reader Kaminski (2019, Sections 7.6.3 and 7.6.4).

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Table 6.2 *Rules for obtaining upper or lower bounds on the (expected) runtime of a loop* `while` $(\varphi)\ \{c\}$ *with respect to postruntime* t.

	$\mathsf{ert}[\texttt{while}\ (\varphi)\ \{c\}](t)\ \leq\ I$	$I\ \leq\ \mathsf{ert}[\texttt{while}\ (\varphi)\ \{c\}](t)$
c **non-probabilistic**	$F_t(I)\ \leq\ I$	$I\ \leq\ F_t(I)$
c **probabilistic**	$F_t(I)\ \leq\ I$	$I\ \leq\ F_t(I)$ and $\lambda\sigma.\ \mathsf{ert}[c]\big(\lvert I(\sigma) - I\rvert\big)(\sigma) \leq c$

Probabilistic Metering Functions

We call a runtime I that satisfies $I \leq F_t(I)$ a *lower invariant*. We have learned above that I being a lower invariant is not enough of a premise to ensure that I is a lower bound on $\mathsf{ert}[\texttt{while}\ (\varphi)\ \{c\}](t)$. The additional premise that we have to add is that the *expected change of* I after performing one iteration of the loop body is *bounded by a constant*, i.e. for every initial state σ_{init}, we have

$$\mathsf{Exp}\Big(\big|I(\sigma_{\text{init}}) - I(\sigma_{\text{final}})\big|\Big)\ \leq\ c\,,$$

for some positive constant c (Hark et al., 2020). This property is called *conditional difference boundedness*. It is easy to show that the above expected value is upper bounded by $\lambda\sigma.\ \mathsf{ert}[c]\big(\lvert I(\sigma) - I\rvert\big)(\sigma)$ and hence we can add

$$\lambda\sigma.\ \mathsf{ert}[c]\big(\lvert I(\sigma) - I\rvert\big)(\sigma)\ \leq\ c\,,\quad \text{for some constant } c \geq 0$$

as a premise (additionally to $I \leq F_t(I)$), to ensure that I is a lower bound on $\mathsf{ert}[\texttt{while}\ (\varphi)\ \{c\}](t)$.[2] The overall situation is summarized in Table 6.2.

Example: The geometric distribution revisited. Let us consider an application of the lower bound proof rule by revisiting the loop c_{geo}:

$$\begin{aligned} c_{\mathsf{geo}}\colon\quad & \texttt{while}\ (c = 1)\,\{ \\ & \qquad c\ :\approx\ {}^1\!/_2 \cdot \langle 0\rangle + {}^1\!/_2 \cdot \langle 1\rangle \\ & \} \end{aligned}$$

We have already shown that $I = 1 + [c = 1] \cdot 4$ is not only a prefixed point of the corresponding characteristic function

$$F_0(X)\ =\ 1\ +\ [c = 1] \cdot \Big(1 + \tfrac{1}{2} \cdot \big(X[c/0] + X[c/1]\big)\Big)\,,$$

[2] Technically, we have to check a few finiteness conditions too, but those are easy to check.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

and thus an upper bound on the expected runtime of the loop, but indeed a true fixed point. Thus, if we show that there exists a constant c, such that

$$\lambda\sigma.\ \mathsf{ert}[\mathit{body}]\big(|I(\sigma) - I|\big)(\sigma) \ \leq\ c\,,$$

then I is also a lower bound and hence the *exact* expected runtime. We can check the above condition as follows:

$$\begin{aligned}
&\lambda\sigma.\ \mathsf{ert}[c \ :\approx\ ½ \cdot \langle 0\rangle + ½ \cdot \langle 1\rangle]\big(|I(\sigma) - I|\big)(\sigma)\\
&=\ \lambda\sigma.\ \left(1 + \tfrac{1}{2} \cdot \Big(\big|I(\sigma) - I[c/0]\big| + \big|I(\sigma) - I[c/1]\big|\Big)\right)(\sigma)\\
&=\ 1 + \tfrac{1}{2} \cdot \Big(\big|I - I[c/0]\big| + \big|I - I[c/1]\big|\Big)\\
&=\ 1 + \tfrac{1}{2} \cdot \Big(\big|1 + [c = 1] \cdot 4 - 1 - [0 = 1] \cdot 4\big|\\
&\qquad\qquad + \ \big|1 + [c = 1] \cdot 4 - 1 - [1 = 1] \cdot 4\big|\Big)\\
&=\ 1 + \tfrac{1}{2} \cdot \Big(\big|[c = 1] \cdot 4\big| \ +\ \big|[c = 1] \cdot 4 - 4\big|\Big)\\
&\leq\ 1 + \tfrac{1}{2} \cdot (4 + 4)\\
&=\ 5
\end{aligned}$$

Thus we have established that

$$\mathsf{ert}\big[c_{\mathsf{geo}}\big](0) \ =\ I \ =\ 1 + [c = 1] \cdot 4\,.$$

6.5.3 Another Proof Rule for Lower Bounds

One can show that the lower bound rule we presented in the previous section is incomplete in the sense that there exist lower bounds which are not conditionally difference bounded. Furthermore, that proof rule is incapable of verifying *infinite* expected runtimes. We present in this section a third proof rule, which is more difficult to apply but in turn *complete* for verifying lower bounds on expected runtimes, in particular infinite expected runtimes.

Recall that, by Kleene's fixed point theorem (Kleene, 1952), the least fixed point characterizing the expected runtime of a loop with characteristic functional F_t is given by

$$\mathsf{lfp}\, F_t \ =\ \sup_{n \in \mathbb{N}} F_t^n(0)\,.$$

It can thus be obtained by fixed point iteration: Starting at 0, we iteratively apply the characteristic functional and take the limit of this iteration process. We now under-approximate each step of this fixed point iteration. To this end, we use a

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$$
\begin{array}{ccccccccccc}
0 & \leq & F_t(0) & \leq & F_t^2(0) & \leq & F_t^3(0) & \leq & \ldots & \leq & \sup\limits_{n\in\mathbb{N}} F_t^n(0) = \mathsf{lfp}\, F_t \\
 & & \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} & & & & \rotatebox{90}{$\leq$} \\
 & & I_1 & & F_t(I_1) & & F_t(I_2) & & \ldots & & \lim\limits_{n\to\infty} F_t(I_n) \\
 & & & & \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} & & & & \rotatebox{90}{$\leq$} \\
 & & & & I_2 & & I_3 & & \ldots & & \lim\limits_{n\to\infty} I_n
\end{array}
$$

Figure 6.3 Illustration of approximating each step of a fixed point iteration of F_t on initial value 0 with a lower ω-invariant I_n. The chain $0 \leq F_t(0) \leq F_t^2(0) \leq \ldots$ on top is a consequence of the monotonicity of F_t which in turn is a consequence of its continuity.

runtime $I_n \in \mathbb{T}$ that is parameterized in a natural number n. Then, I_n is called a *lower ω-invariant* of $\texttt{while}\,(\varphi)\,\{c\}$ with respect to runtime t if and only if

$$I_1 \;\leq\; F_t(0) \qquad \text{and} \qquad \forall\, n \geq 1\colon \quad I_{n+1} \;\leq\; F_t(I_n)\,.$$

Intuitively, a lower ω-invariant I_n under-approximates the n-th step of a fixed point iteration that determines the exact expected runtime of a loop. It thus represents a lower bound on the expected runtime of those executions that finish within n loop iterations, weighted according to their probabilities. Intuitively, the limit of I_n consequently represents a lower bound on the expected runtime for any number of loop iterations. More formally, we can show by induction that

$$I_{n+1} \;\leq\; F_t(I_n) \;\leq\; F_t^{n+1}(0)\,.$$

An illustration of a fixed point iteration approximated by I_n is given in Figure 6.3. Formally, we obtain the following proof rule based on ω-invariants: If I_n is a lower ω-invariant of loop $\texttt{while}\,(\varphi)\,\{c\}$ with respect to runtime t and the limit of I_n exists then

$$\lim_{n\to\infty} I_n \;\leq\; \mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t)\,.$$

This rule is obviously complete as we can always choose the exact fixed point iteration sequence $I_n = F_t^n(0)$ as lower ω-invariant.

It is worthwhile to note that for upper bounds there is no need for ω-invariants, even though a dual rule exists: I_n is called an *upper ω-invariant* of $\texttt{while}\,(\varphi)\,\{c\}$ with respect to runtime t if and only if

$$F_t(0) \;\leq\; I_1 \qquad \text{and} \qquad \forall\, n \geq 1\colon \quad F_t(I_n) \;\leq\; I_{n+1}\,.$$

If I_n is an upper ω-invariant of loop $\texttt{while}\,(\varphi)\,\{c\}$ with respect to runtime t and

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

the limit of I_n exists, then indeed

$$\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](t) \;\leq\; \lim_{n\to\infty} I_n\ .$$

However, one can show that in this case $\lim_{n\to\infty} I_n$ itself is already an upper (global) invariant, i.e.

$$F_t\left(\lim_{n\to\infty} I_n\right) \;\leq\; \lim_{n\to\infty} I_n$$

and thus, there is no need to guess such a sequence and then find the limit. Instead, we can just immediately guess a limit and check whether this limit is an upper invariant, without having to perform an additional induction on n.

Example: Disproving positive almost-sure termination. The expected runtime transformer ert enables proving positive almost-sure termination by verifying a finite upper bound on the expected runtime of a program. With the help of ω-invariants, it can also be employed to prove that infinity is a *lower bound* on the expected runtime. In other words, we can verify that a given program does *not* terminate positively almost-surely. As an example, let us verify that the concatenation of two positively almost-surely terminating programs may itself not terminate positively almost-surely. We already presented a counterexample, but without proof, namely the program

$$\begin{aligned} c_{\mathsf{cex}}\colon\quad & 1\colon\ x := 1;\ b := 1;\\ & 2\colon\ \texttt{while}\,(b = 1)\,\{b :\approx {}^1\!/_2\langle 0\rangle + {}^1\!/_2\langle 1\rangle;\ x := 2x\};\\ & 3\colon\ \texttt{while}\,(x > 0)\,\{x := x - 1\}\ . \end{aligned}$$

Our goal is to formally prove that c_{cex} has an infinite expected runtime, i.e. $\infty \leq \mathsf{ert}[c_{\mathsf{cex}}](0)$. To this end, let us denote the program in the i-th line of c_{cex} by c_i. By the rule for sequential composition, we then obtain

$$\mathsf{ert}[c_{\mathsf{cex}}](0) \;=\; \mathsf{ert}[c_1](\mathsf{ert}[c_2](\mathsf{ert}[c_3](0)))\ .$$

Let us thus start by analyzing the second loop, i.e. program c_3. Its characteristic functional with respect to continuation 0 is given by

$$F_0(X) \;=\; 1 + [x > 0]\cdot\big(1 + X[x/x-1]\big)\ .$$

Since the variable x is decremented in each loop iteration and every iteration consumes two units of time (one for the guard evaluation and one for the assignment), we use the lower ω-invariant

$$I_n \;=\; 1 + [0 < x < n-1]\cdot 2x + [x \geq n-1]\cdot(2n-3)$$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

of the loop c_3 to conclude that

$$\mathsf{ert}[c_3](0) \;\geq\; \lim_{n\to\infty} I_n \;=\; 1 + [x > 0] \cdot 2x \;.$$

We now have an under-approximation of the expected runtime of c_3. We can continue reasoning about c_2 using this under-approximation because the ert-transformer satisfies a fundamental property: it is *monotonic*. Hence,

$$\begin{aligned} &\mathsf{ert}[c_3](0) \;\geq\; 1 + [x > 0] \cdot 2x \qquad \text{implies} \\ &\qquad \mathsf{ert}[c_2 \,;\, c_3](0) \;=\; \mathsf{ert}[c_2]\big(\mathsf{ert}[c_3](0)\big) \;\geq\; \mathsf{ert}[c_2](1 + [x > 0] \cdot 2x) \;. \end{aligned}$$

Our next step is therefore to analyze the expected runtime of the first loop, i.e. c_2, with respect to continuation $t = 1 + [x > 0] \cdot 2x$. The corresponding characteristic functional is given by

$$\begin{aligned} G_t(X) \;=\; & 1 + [b \neq 1] \cdot (1 + [x > 0] \cdot 2x) \\ & + [b = 1] \cdot \big(2 + \tfrac{1}{2} \cdot X[x/2x][b/0] + \tfrac{1}{2} \cdot X[x/2x][b/1]\big) \;. \end{aligned}$$

As a lower ω-invariant of c_2, we propose

$$\begin{aligned} J_n \;=\; & 1 + [b \neq 1] \cdot (1 + [x > 0] \cdot 2x) \\ & + [b = 1] \cdot (7 - \tfrac{5}{2^{n-1}} + (n-1) \cdot [x > 0] \cdot 2x) \;. \end{aligned}$$

The calculations establishing that J_n is a lower ω-invariant go as follows:

$$\begin{aligned} G_t(0) \;=\; & 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ & + [b = 1] \cdot \Big(2 + \tfrac{1}{2} \cdot 0[x/2x][b/0] + \tfrac{1}{2} \cdot 0[x/2x][b/1]\Big) \\ =\; & 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) + [b = 1] \cdot 2 \\ =\; & I_1 \;\geq\; I_1 \;\checkmark \end{aligned}$$

$$\begin{aligned} G_t(J_n) \;=\; & 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ & + [b = 1] \cdot \Big(2 + \tfrac{1}{2} \cdot J_n[x/2x][b/0] + \tfrac{1}{2} \cdot J_n[x/2x][b/1]\Big) \\ =\; & 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ & + [b = 1] \cdot \Big(2 + \tfrac{1}{2} \cdot (2 + [2x > 0] \cdot 4x) \\ & \qquad + \tfrac{1}{2} \cdot \Big(8 - \tfrac{5}{2^{n-1}} + (n-1) \cdot \underbrace{[2x > 0]}_{=\,[x>0]} \cdot 4x\Big)\Big) \\ =\; & 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ & + [b = 1] \cdot \Big(7 - \tfrac{5}{2^{n+1-1}} + (n+1-1) \cdot [x > 0] \cdot 2x\Big) \end{aligned}$$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$$= I_{n+1} \geq I_{n+1} \quad \checkmark$$

Hence, our proof rule for lower ω-invariants yields

$$\begin{aligned} \mathsf{ert}[c_2](1 + [x > 0] \cdot 2x) &\geq \lim_{n\to\infty} J_n \\ &= 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ &\quad + [b = 1] \cdot (7 + [x > 0] \cdot \infty) \,. \end{aligned}$$

Again appealing to monotonicity of the ert-transformer, we can complete the runtime analysis of program $c_{\mathtt{cex}}$:

$$\begin{aligned} \mathsf{ert}[c_{\mathtt{cex}}](0) &= \mathsf{ert}[c_1](\mathsf{ert}[c_2](\mathsf{ert}[c_3](0))) \\ &\geq \mathsf{ert}[c_1](\mathsf{ert}[c_2](1 + [x > 0] \cdot 2x)) \\ &\geq \mathsf{ert}[c_1]\Big(1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ &\qquad + [b = 1] \cdot (7 + [x > 0] \cdot \infty)\Big) \\ &= \mathsf{ert}[x := 1]\Big(\mathsf{ert}[b := 1]\Big(\\ &\qquad 1 + [b \neq 1] \cdot \big(1 + [x > 0] \cdot 2x\big) \\ &\qquad + [b = 1] \cdot (7 + [x > 0] \cdot \infty)\Big)\Big) \\ &= \mathsf{ert}[x := 1](1 + 8 + [x > 0] \cdot \infty) \\ &= 1 + 1 + 8 + [1 > 0] \cdot \infty \\ &= \infty \end{aligned}$$

Overall, we obtain that the expected runtime of program $c_{\mathtt{cex}}$ is infinite even though it terminates with probability one. In other words, $c_{\mathtt{cex}}$ terminates almost-surely, but not positively almost-surely. Furthermore, notice that both sub-programs c_2 and c_3 for themselves have finite expected runtimes, since

$$\mathsf{ert}[c_2](0) = 1 + [b = 1] \cdot 4 \qquad \text{and} \qquad \mathsf{ert}[c_3](0) = 1 + [x > 0] \cdot 2x \,.$$

We emphasize that the ert-calculus allows for reasoning about positive almost-sure termination (PAST) although PAST itself is not compositional.

6.5.4 Independent and Identically Distributed Loops

So far, we have studied two classes of complete proof rules. However, both rules rely on finding invariants, which is usually the hardest part of the verification process. We now consider a restricted class of loops whose *exact* expected runtime can be analyzed without invariants or other user-supplied artifacts. That class thus offers

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

$$\texttt{while}\Big((x-5)^2+(y-5)^2 \leq 25\Big)\ \{$$
$$\quad x :\approx \texttt{uniform}[0\ldots 10];$$
$$\quad y :\approx \texttt{uniform}[0\ldots 10]$$
$$\}$$

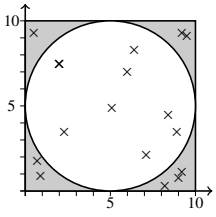

Figure 6.4 A probabilistic program sampling a point within a circle uniformly at random using rejection sampling. The picture on the right-hand side visualizes the procedure: In each iteration a point (×) is sampled. If we obtain a point within the white area inside the square, we terminate. Otherwise, we resample.

a high degree of automation. Intuitively, the key concept underlying our next proof rule are loops without data flow across different loop iterations. For deterministic programs, such loops are not very interesting: They either terminate after exactly one iteration or never. This is not the case for probabilistic programs. Consider, for example, the probabilistic program depicted in Figure 6.4. It samples a point within a circle with center $(5,5)$ and radius 5 uniformly at random by means of rejection sampling: In each loop iteration, we sample a point $(x,y) \in [0,10]^2$ with some fixed precision. If the sampled point lies within the circle, we terminate; otherwise, we resample. Although our program admits arbitrarily long runs, the program terminates within finite expected time. Moreover, there is no data flow across loop iterations. This is an example of an *independent and identically distributed (i.i.d.) loop*. In the remainder of this section we develop a proof rule to determine exact expected runtimes of i.i.d. loops.

Towards a rigorous proof rule, let us first formally characterize i.i.d. loops. Let $\mathsf{Var}(f)$ denote the set of all variables that "occur in" runtime $f \in \mathbb{T}$, i.e.

$$x \in \mathsf{Var}(f) \qquad \text{iff} \qquad \exists\,\sigma\ \exists\, v, v' \colon \quad f\Big(\sigma[x/v]\Big) \neq f\Big(\sigma[x/v']\Big)\,.$$

Furthermore, let the set $\mathsf{Mod}(c)$ be the collection of all variables that occur on the left-hand side of an assignment in a program c. We then call a runtime f *unaffected* by program c, denoted

$$f \not\Mod c$$

if and only if $\mathsf{Var}(f) \cap \mathsf{Mod}(c) = \emptyset$. Moreover, let us denote by

$$\wp c(f)$$

the expected value of f after executing program c. This value can be computed similarly to expected runtimes with our ert-calculus by assuming that the time consumed by each statement is zero (this corresponds to the weakest preexpectation calculus

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

of McIver and Morgan, 2004). With these notions at hand, a loop $\texttt{while}\,(\varphi)\,\{c\}$ is *f-i.i.d.* for some postruntime $f \in \mathbb{T}$ if and only if

(i) the probability of loop guard φ being true after one execution of loop body c is unaffected by c, i.e.

$$\wp c\big([\varphi]\big) \not\parallel c\,,$$

(ii) the probability of violating the loop guard and continuing with runtime f after one execution of loop body c is unaffected by c, i.e.

$$\wp c\big([\neg\varphi]\cdot f\big) \not\parallel c\,, \qquad \text{and}$$

(iii) every loop iteration runs in the same expected time, i.e.

$$\mathsf{ert}[c](0) \not\parallel c\,.$$

For example, the program in Figure 6.4 that samples a point in a circle is f-i.i.d. for every postruntime $f \in \mathbb{T}$. How does the fact that a loop is f-i.i.d. help us when analyzing the expected runtime of a program? Intuitively, since each loop iteration has the same expected runtime, we can characterize the expected runtime of the whole loop as the expected runtime of a single loop iteration divided by the probability of termination, i.e. 1 minus the probability of satisfying the loop guard after execution the loop body. Formally, we additionally have to take the time consumed by guard evaluations and the possibility that the loop guard is never satisfied into account. This leads us to the following proof rule (Batz et al., 2018):

Proof rule for f-i.i.d. loops. Let $\texttt{while}\,(\varphi)\,\{c\}$ be an f-i.i.d. loop such that the loop body c terminates almost-surely, i.e. $\wp c(1) = 1$. Then, the expected runtime of $\texttt{while}\,(\varphi)\,\{c\}$ with respect to postruntime $f \in \mathbb{T}$ is

$$\mathsf{ert}[\texttt{while}\,(\varphi)\,\{c\}](f) \;=\; 1 \;+\; [\varphi]\cdot\frac{1+\mathsf{ert}[c]([\neg\varphi]\cdot f)}{1-\mathtt{Pr}[c]([\varphi])} \;+\; [\neg\varphi]\cdot f\,,$$

where we define ${}^{0}\!/_{0} = 0$ and ${}^{a}\!/_{0} = \infty$ for $a \neq 0$.

6.6 Application: Bayesian networks

The notion of f-i.i.d. loops prevents data flow across loop iterations. While this might seem like a severe restriction, it naturally applies to certain classes of probabilistic models. In particular, Bayesian networks (Koller and Friedman, 2009) are probabilistic graphical models representing joint probability distributions of sets of random variables with conditional dependencies. Graphical models are popular as they allow to succinctly represent complex distributions in a human-readable way. For example, Bayesian networks have applications in machine learning (Heckerman,

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

2008), speech recognition (Zweig and Russell, 1998), sports betting (Constantinou et al., 2012), gene regulatory networks (Friedman et al., 2000), medicine (Jiang and Cooper, 2010), and finance (Neapolitan and Jiang, 2010).

The central problem for Bayesian networks is *probabilistic inference*, i.e. determining the probability of an event given observed evidence. This problem is often approached using sampling-based techniques, such as rejection sampling: Repeatedly sample from the joint distribution of the network until the obtained sample complies with all observed evidence. However, a major problem with rejection sampling is that for poorly conditioned data, many samples have to be rejected to obtain a single compliant sample. In fact, Gordon et al. (2014) point out that "the main challenge in this setting [i.e. sampling based approaches] is that many samples that are generated during execution are ultimately rejected for not satisfying the observations." If too many samples are rejected, the expected sampling time grows so large that sampling becomes infeasible. The expected sampling time of a Bayesian network is therefore a key figure for deciding whether sampling based inference is the method of choice. In other words, we are concerned with the question:

"Given a Bayesian network with observed evidence, how long does it take to obtain a single sample that satisfies the observations?"

In this section, we present how this question can be addressed fully automatically: We translate a Bayesian network into an equivalent pGCL program such that the expected runtime of the resulting program corresponds to the expected sampling time of the network. The expected runtime is then determined using the ert-calculus and our proof rule for f-i.i.d. loops.

6.6.1 From Bayesian Networks to Probabilistic Programs

Let us briefly introduce Bayesian networks by means of an example. Further details – including formal definitions – are found in Batz et al. (2018); Koller and Friedman (2009). A Bayesian network is a directed acyclic graph in which every node is a random variable and every edge between two nodes represents a probabilistic dependency between these nodes. As a running example, consider the network depicted in Figure 6.5 (inspired by Koller and Friedman, 2009) that models the mood of students after taking an exam. The network contains four random variables. They represent the difficulty of the exam (D), the level of preparation of a student (P), the achieved grade (G), and the resulting mood (M). For simplicity, let us suppose that each random variable assumes either value 0 or 1. The underlying dependencies express that the mood of a student depends on the achieved grade which, in turn, depends on the difficulty of the exam and the amount of preparation before taking it. Every node is accompanied by a conditional probability table that

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

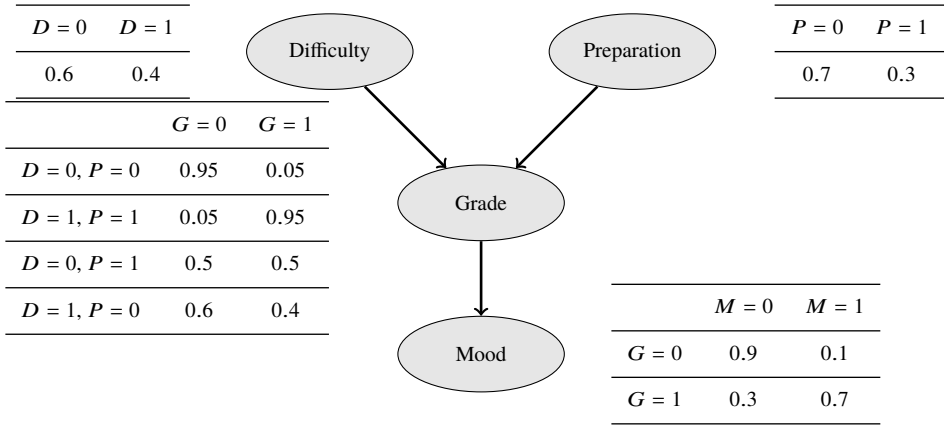

Figure 6.5 A Bayesian network

provides the probabilities of a node given the values of all the nodes it depends upon. We can then use the Bayesian network to answer queries such as "What is the probability that a student is well-prepared for an exam ($P = 1$), but ends up with a bad mood ($M = 0$)?"

It can be shown that every Bayesian network with observed evidence can be translated into an equivalent pGCL program, i.e. a program describing the same conditional probability distribution (Batz et al., 2018). For instance, Figure 6.6 shows the probabilistic program corresponding to the Bayesian network in Figure 6.5 and observation $P = 1$. Here, the statement `repeat` $\{\ c\ \}$ `until` (φ) is a shortcut for c; `while` $(\varphi)\ \{c\}$. Essentially, every node in a network corresponds to a variable. It is then straightforward to encode the (discrete) conditional probability table of every node using conditional branching and random assignments.

To deal with observations, one could syntactically enrich the programming language to allow for `observe`-statements. This approach is taken in, for example,Katoen et al. (2015); Olmedo et al. (2018); Bichsel et al. (2018). However, taking that approach would not give us an insight on how long it would take to obtain an execution trace that complies with the observations. Instead, we wrap around the original program a global loop that implements rejection sampling. That is, the whole program is re-run until all observations are satisfied. Since no variable inside the loop body of such a program is accessed before it is set by a probabilistic assignment, there is no data flow across loop iterations. In other words, all loops that model Bayesian networks are f-i.i.d. for every postruntime $f \in \mathbb{T}$.

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

```
1  repeat {                                  10      } else {
2      x_D :≈ 0.6 · ⟨0⟩ + 0.4 · ⟨1⟩;         11          x_G :≈ 0.6 · ⟨0⟩ + 0.4 · ⟨1⟩
3      x_P :≈ 0.7 · ⟨0⟩ + 0.3 · ⟨1⟩          12      };
4      if (x_D = 0 ∧ x_P = 0) {              13      if (x_G = 0) {
5          x_G :≈ 0.95 · ⟨0⟩ + 0.05 · ⟨1⟩    14          x_M :≈ 0.9 · ⟨0⟩ + 0.1 · ⟨1⟩
6      } else if (x_D = 1 ∧ x_P = 1) {       15      } else {
7          x_G :≈ 0.05 · ⟨0⟩ + 0.95 · ⟨1⟩    16          x_M :≈ 0.3 · ⟨0⟩ + 0.7 · ⟨1⟩
8      } else if (x_D = 0 ∧ x_P = 1) {       17      }
9          x_G :≈ 0.5 · ⟨0⟩ + 0.5 · ⟨1⟩      18  } until (x_P = 1)
```

Figure 6.6 The probabilistic program obtained from the network in Figure 6.5.

Consequently, the expected runtime of programs obtained from Bayesian networks can be analyzed fully automatically by applying the rules of the ert-calculus.

6.6.2 Implementation

We implemented a prototype to analyze expected sampling times of Bayesian networks (cf. Batz et al., 2018). More concretely, our tool takes as input a Bayesian network together with observations in the popular Bayesian Network Interchange Format.[3] The network is first translated into a probabilistic program. The expected runtime of the resulting program is then determined fully automatically by applying our ert-calculus together with our proof rule for f-i.i.d. loops.

The size of the resulting program is linear in the total number of rows of all conditional probability tables in the network. The program size is thus *not* the bottleneck of our analysis. As we are dealing with an NP-hard problem (Cooper, 1990; Dagum and Luby, 1993), it is not surprising that our algorithm has a worst-case exponential time complexity. However, also the space complexity of our algorithm is exponential in the worst case: As an expectation is propagated backwards through an `if`-clause of the program, the size of the expectation is potentially multiplied. This is also the reason that our analysis runs out of memory on some benchmarks.

We evaluated our implementation on the *largest* Bayesian networks in the Bayesian Network Repository (Scutari, 2017) that consists – to a large extent – of real-world Bayesian networks including expert systems for, e.g., electromyography (`munin`) (Andreassen et al., 1989), hematopathology diagnosis (`hepar2`) (Onisko et al., 1998), weather forecasting (`hailfinder`) (Abramson et al., 1996), and printer troubleshooting in the Windows 95 operating system (`win95pts`) (Ramanna et al., 2013, Section 5.6.2). All experiments were performed on an HP BL685C G7.

[3] http://www.cs.cmu.edu/ fgcozman/Research/InterchangeFormat/

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Although up to 48 cores with 2.0GHz were available, only one core was used apart from Java's garbage collection. The Java virtual machine was limited to 8GB of RAM.

Our experimental results are shown in Table 6.3. The number of nodes of the considered networks ranges from 56 to 1041. For each Bayesian network, we computed the expected sampling time (EST) for different collections of observed nodes (#obs). EST = ∞ means that our implementation automatically infers that the expected sampling time is in fact infinite, which is the case if and only if the probability of complying with all observed evidence is precisely 0. #obs = 0 means that no evidence is observed; the repeat-until loop would thus be executed exactly once. For #obs > 0, observations were picked at random. Note that the time required by our prototype varies depending on both the number of observed nodes and the actual observations. Thus, there are cases in which we run out of memory although the total number of observations is small. Furthermore, Table 6.3 provides the *average Markov Blanket size*, i.e. the average number of parents, children and children's parents of nodes in the Bayesian network (Pearl, 1985), as an indicator measuring how independent nodes in the network are.

In order to obtain an understanding of what the EST corresponds to in actual execution times on a real machine, we also performed simulations for the `win95pts` network. More precisely, we generated Java programs from this network analogously to the translation from Bayesian networks into pGCL programs. This allowed us to approximate that our Java setup can execute $9.714 \cdot 10^6$ steps (in terms of EST) per second. For the `win95pts` with 17 observations, an EST of $1.11 \cdot 10^{15}$ then corresponds to an expected time of approximately 3.6 *years* in order to obtain a *single* valid sample. We were additionally able to find a case with 13 observed nodes where our tool discovered within 0.32 seconds an EST that corresponds to approximately 4.3 *million years*. In contrast, exact inference using variable elimination was almost instantaneous. This demonstrates that knowing expected sampling times upfront can indeed be beneficial when selecting an inference method.

6.7 Conclusion and Future Directions

We presented a weakest-precondition-style calculus for reasoning about the expected runtime of probabilistic programs. Our calculus demonstrates how standard techniques from program verification – in particular denotational semantics and invariants – enable elegant proofs of (non)positive-almost sure termination. Furthermore, both lower and upper bounds on the expected runtime can be determined. We studied the restricted class of independently and identically distributed loops, which enable a fully automated runtime analysis. In particular, Bayesian networks

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Table 6.3 *Experimental results. Time is in seconds. MO denotes out of memory.* #obs *refers to the number of observed nodes.*

Network	**#obs**	**Time**	**EST**	**#obs**	**Time**	**EST**	**#obs**	**Time**	**EST**
earthquake	*#nodes: 5, #edges: 4, avg. Markov Blanket: 2.00*								
	0	0.09	$8.000 \cdot 10^{0}$	2	0.23	$9.276 \cdot 10^{1}$	4	0.24	$1.857 \cdot 10^{2}$
cancer	*#nodes: 5, #edges: 4, avg. Markov Blanket: 2.00*								
	0	0.09	$8.000 \cdot 10^{0}$	2	0.22	$1.839 \cdot 10^{1}$	5	0.20	$5.639 \cdot 10^{2}$
survey	*#nodes: 6, #edges: 6, avg. Markov Blanket: 2.67*								
	0	0.09	$1.000 \cdot 10^{1}$	2	0.21	$2.846 \cdot 10^{2}$	5	0.22	$9.113 \cdot 10^{3}$
asia	*#nodes: 8, #edges: 8, avg. Markov Blanket: 2.50*								
	0	0.26	$1.400 \cdot 10^{1}$	2	0.25	$3.368 \cdot 10^{2}$	6	0.25	$8.419 \cdot 10^{4}$
sachs	*#nodes: 11, #edges: 17, avg. Markov Blanket: 3.09*								
	0	0.13	$2.000 \cdot 10^{1}$	3	0.24	$7.428 \cdot 10^{2}$	6	2.72	$5.533 \cdot 10^{7}$
insurance	*#nodes: 27, #edges: 52, avg. Markov Blanket: 5.19*								
	0	0.17	$5.200 \cdot 10^{1}$	3	0.31	$5.096 \cdot 10^{3}$	5	0.91	$1.373 \cdot 10^{5}$
alarm	*#nodes: 37, #edges: 46, avg. Markov Blanket: 3.51*								
	0	0.14	$6.200 \cdot 10^{1}$	2	MO	—	6	40.47	$3.799 \cdot 10^{5}$
barley	*#nodes: 48, #edges: 84, avg. Markov Blanket: 5.25*								
	0	0.46	$8.600 \cdot 10^{1}$	2	0.53	$5.246 \cdot 10^{4}$	5	MO	—
hailfinder	*#nodes: 56, #edges: 66, avg. Markov Blanket: 3.54*								
	0	0.23	$9.500 \cdot 10^{1}$	5	0.63	$5.016 \cdot 10^{5}$	9	0.46	$9.048 \cdot 10^{6}$
hepar2	*#nodes: 70, #edges: 123, avg. Markov Blanket: 4.51*								
	0	0.22	$1.310 \cdot 10^{2}$	1	1.84	$1.579 \cdot 10^{2}$	2	MO	—
win95pts	*#nodes: 76, #edges: 112, avg. Markov Blanket: 5.92*								
	0	0.20	$1.180 \cdot 10^{2}$	1	0.36	$2.284 \cdot 10^{3}$	3	0.36	$4.296 \cdot 10^{5}$
	7	0.91	$1.876 \cdot 10^{6}$	12	0.42	$3.973 \cdot 10^{7}$	17	61.73	$1.110 \cdot 10^{15}$
pathfinder	*#nodes: 135, #edges: 200, avg. Markov Blanket: 3.04*								
	0	0.37	217	1	0.53	$1.050 \cdot 10^{4}$	3	31.31	$2.872 \cdot 10^{4}$
	5	MO	—	7	5.44	∞	7	480.83	∞
andes	*#nodes: 223, #edges: 338, avg. Markov Blanket: 5.61*								
	0	0.46	$3.570 \cdot 10^{2}$	1	MO	—	3	1.66	$5.251 \cdot 10^{3}$
	5	1.41	$9.862 \cdot 10^{3}$	7	0.99	$8.904 \cdot 10^{4}$	9	0.90	$6.637 \cdot 10^{5}$
pigs	*#nodes: 441, #edges: 592, avg. Markov Blanket: 3.66*								
	0	0.57	$7.370 \cdot 10^{2}$	1	0.74	$2.952 \cdot 10^{3}$	3	0.88	$2.362 \cdot 10^{3}$
	5	0.85	$1.260 \cdot 10^{5}$	7	1.02	$1.511 \cdot 10^{6}$	8	MO	—
munin	*#nodes: 1041, #edges: 1397, avg. Markov Blanket: 3.54*								
	0	1.29	$1.823 \cdot 10^{3}$	1	1.47	$3.648 \cdot 10^{4}$	3	1.37	$1.824 \cdot 10^{7}$
	5	1.43	∞	9	1.79	$1.824 \cdot 10^{16}$	10	65.64	$1.153 \cdot 10^{18}$

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

– if interpreted as probabilistic programs – are covered by this class. Hence, exact expected sampling times of Bayesian networks can be determined automatically.

Since the original development of the expected runtime calculus (Kaminski et al., 2016), there has been ongoing research into several further directions. We conclude with a brief discussion of recent developments.

Applications. The main application of our calculus is the analysis of *randomized algorithms*. However, many randomized algorithms rely on advanced programming features that are not supported by the simplistic language considered in this chapter. There have been various extensions of weakest-precondition-style calculi for probabilistic programs that attempt to support additional features while maintaining elegant and applicable proof rules. In particular, recursion (Olmedo et al., 2016) and dynamic data structures (Batz et al., 2019) have been incorporated into our probabilistic guarded command language. An important feature that is still missing is support for concurrent randomized algorithms.

Automation. While finding invariants is a challenging task, we were usually able to find correct invariants by considering a few loop unrollings. Hence, there is hope that the proof rules presented in this chapter provide a foundation for automated reasoning about expected runtimes. As a first step, our calculus has been mechanized in the theorem prover Isabelle (Hölzl, 2016). Furthermore, the class of i.i.d. loops is a further step towards automation for a restricted class of programs. More recently, Ngo et al. (2018) extended work on automated amortized resource analysis to automatically reason about bounds on the expected runtime of a larger class of probabilistic programs. They reduce inference of upper bounds to a linear programming problem. In fact, their work can be considered an extension of our ert-calculus by specialized rules. In particular, soundness of their approach explicitly relies on the soundness of the expected runtime transformer presented here.

Almost-sure termination. Our calculus allows for proving positive almost-sure termination, i.e. finite expected runtimes. A weaker notion is plain almost-sure termination, i.e. termination with probability 1. Powerful rules for proving almost-sure termination of a large class of programs have been developed (McIver et al., 2018), and for certain subclasses of probabilistic programs even automated approaches for proving almost-sure termination exist (Esparza et al., 2012; Chatterjee et al., 2016, 2017; Agrawal et al., 2017).

https://doi.org/10.1017/9781108770750.007 Published online by Cambridge University Press

Acknowledgements

Kaminski and Katoen have been supported by the ERC Advanced Grant FRAPPANT (project number 787914) and the DFG-funded Research Training Group 2236 UnRAVeL. Matheja was supported by the DFG Project ATTESTOR. The authors thank Kevin Batz and Federico Olmedo; joint work with them provided the basis for this chapter.

References

Abramson, Bruce, Brown, John, Edwards, Ward, Murphy, Allan, and Winkler, Robert L. 1996. Hailfinder: A Bayesian system for forecasting severe weather. *International Journal of Forecasting*, **12**(1), 57–71.

Adleman, Leonard Max, and Huang, Ming-Deh. 1992. *Primality Testing and Abelian Varieties over Finite Fields*. *Lecture Notes in Mathematics*, **1512**.

Agrawal, Sheshansh, Chatterjee, Krishnendu, and Novotnỳ, Petr. 2017. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, **2**(POPL), 1–32.

Andreassen, Steen, Jensen, Finn V, Andersen, Stig Kjær, Falck, B, Kjærulff, U, Woldbye, M, Sørensen, AR, Rosenfalck, A, and Jensen, F. 1989. MUNIN: an expert EMG assistant. Pages 255–277 of: *Computer-aided Electromyography and Expert Systems*. Pergamon Press.

Batz, Kevin, Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, and Matheja, Christoph. 2018. How long, O Bayesian network, will I sample thee? – A program analysis perspective on expected sampling times. Pages 186–213 of: *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 10801. Springer.

Batz, Kevin, Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, Matheja, Christoph, and Noll, Thomas. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proceedings of the ACM on Programming Languages*, **3** (POPL), 1–29.

Bichsel, Benjamin, Gehr, Timon, and Vechev, Martin T. 2018. Fine-grained semantics for probabilistic programs. Pages 145–185 of: *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 10801. Springer.

Bournez, Olivier, and Garnier, Florent. 2005. Proving positive almost-sure termination. Pages 323–337 of: *International Conference on Rewriting Techniques and Applications (RTA)*. Lecture Notes in Computer Science, vol. 3467. Springer.

Chakarov, Aleksandar, and Sankaranarayanan, Sriram. 2013. Probabilistic program analysis with martingales. Pages 511–526 of: *Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 8044. Springer.

Chatterjee, Krishnendu, Fu, Hongfei, Novotný, Petr, and Hasheminezhad, Rouzbeh. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. Pages 327–342 of: *Principles of Programming Languages (POPL)*. ACM.

Chatterjee, Krishnendu, Novotný, Petr, and Zikelic, Dorde. 2017. Stochastic invariants for probabilistic termination. Pages 145–160 of: *Principles of Programming Languages (POPL)*. ACM.

Constantinou, Anthony C., Fenton, Norman E., and Neil, Martin. 2012. pi-football: a Bayesian network model for forecasting Association Football match outcomes. *Knowledge-Based Systems*, **36**, 322–339.

Cooper, Gregory F. 1990. The computational complexity of probabilistic inference using Bayesian Belief Networks. *Artificial intelligence*, **42**(2-3), 393–405.

Dagum, Paul, and Luby, Michael. 1993. Approximating probabilistic inference in Bayesian Belief Networks is NP-hard. *Artificial intelligence*, **60**(1), 141–153.

Dijkstra, Edgser W. 1976. *A Discipline of Programming*. Prentice Hall.

Esparza, Javier, Gaiser, Andreas, and Kiefer, Stefan. 2012. Proving termination of probabilistic programs using patterns. Pages 123–138 of: *Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 7358. Springer.

Freivalds, Rūsiņš Mārtiņš. 1979. Fast probabilistic algorithms. Pages 57–69 of: *Mathematical Foundations of Computer Science (MFCS)*. Lecture Notes in Computer Science, vol. 74. Springer.

Friedman, Nir, Linial, Michal, Nachman, Iftach, and Pe'er, Dana. 2000. Using Bayesian networks to analyze expression data. Pages 127–135 of: *Computational Molecular Biology (RECOMB)*. ACM.

Frohn, Florian, Naaf, Matthias, Hensel, Jera, Brockschmidt, Marc, and Giesl, Jürgen. 2016. Lower runtime bounds for integer programs. Pages 550–567 of: *International Joint Conference on Automated Reasoning (IJCAR)*. Lecture Notes in Computer Science, vol. 9706. Springer.

Gordon, Andrew D., Henzinger, Thomas A., Nori, Aditya V., and Rajamani, Sriram K. 2014. Probabilistic programming. Pages 167–181 of: *Future of Software Engineering (FOSE)*. ACM.

Hark, Marcel, Kaminski, Benjamin Lucien, Giesl, Jürgen, and Katoen, Joost-Pieter. 2020. Aiming low Is harder – inductive proof rules for lower bounds on weakest pre-expectations in probabilistic program verification. *Proceedings of the ACM on Programming Languages*, **4**(POPL), 37:1–37:28.

Hart, Sergiu, Sharir, Micha, and Pnueli, Amir. 1983. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, **5**(3), 356–380.

Heckerman, David. 2008. A tutorial on learning with Bayesian networks. Pages 33–82 of: *Innovations in Bayesian Networks*. Studies in Computational Intelligence, vol. 156. Springer.

Hoare, Charles Antony Richard. 1962. Quicksort. *The Computer Journal*, **5**(1), 10–15.

Hölzl, Johannes. 2016. Formalising semantics for expected running time of probabilistic programs. Pages 475–482 of: *Interactive Theorem Proving (ITP)*. Lecture Notes in Computer Science, vol. 9807. Springer.

Hur, Chung-Kil, Nori, Aditya V, Rajamani, Sriram K, and Samuel, Selva. 2014. Slicing probabilistic programs. Pages 133–144 of: *ACM SIGPLAN Notices*, vol. 49. ACM.

Ibe, Oliver C. 2013. *Elements of Random Walk and Diffusion Processes*. John Wiley & Sons.

Jiang, Xia, and Cooper, Gregory F. 2010. A Bayesian spatio-temporal method for disease outbreak detection. *Journal of the American Medical Informatics Association*, **17**(4), 462–471.

Kaminski, Benjamin Lucien. 2019. *Advanced Weakest Precondition Calculi for Probabilistic Programs*. Ph.D. thesis, RWTH Aachen University, Germany.

Kaminski, Benjamin Lucien, and Katoen, Joost-Pieter. 2015. On the hardness of almost-sure termination. Pages 307–318 of: *Mathematical Foundations of Computer Science (MFCS), Part I*. Lecture Notes in Computer Science, vol. 9234. Springer.

Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, Matheja, Christoph, and Olmedo, Federico. 2016. Weakest precondition reasoning for expected runtimes of probabilistic programs. Pages 364–389 of: *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 9632. Springer.

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

Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, Matheja, Christoph, and Olmedo, Federico. 2018b. Weakest precondition reasoning for expected runtimes of randomized Algorithms. *Journal of the ACM*, **65**(5), 30:1–30:68.

Katoen, Joost-Pieter, Gretz, Friedrich, Jansen, Nils, Kaminski, Benjamin Lucien, and Olmedo, Federico. 2015. Understanding probabilistic programs. Pages 15–32 of: *Correct System Design – Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science, vol. 9360. Springer.

Kleene, Stephen Cole. 1952. *Introduction to Metamathematics*. North-Holland.

Koller, Daphne, and Friedman, Nir. 2009. *Probabilistic Graphical Models – Principles and Techniques*. MIT Press.

McIver, Annabelle, and Morgan, Carroll. 2004. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.

McIver, Annabelle, Morgan, Carroll, Kaminski, Benjamin Lucien, and Katoen, Joost-Pieter. 2018. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages*, **2**(POPL), 33:1–33:28.

Minka, Tom, and Winn, John. 2017. *Infer.NET*. Accessed online at http://infernet.azurewebsites.net July 30, 2018.

Neapolitan, Richard E, and Jiang, Xia. 2010. *Probabilistic Methods for Financial and Marketing Informatics*. Morgan Kaufmann.

Ngo, Van Chan, Carbonneaux, Quentin, and Hoffmann, Jan. 2018. Bounded expectations: resource analysis for probabilistic programs. Pages 496–512 of: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Olmedo, Federico, Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, and Matheja, Christoph. 2016. Reasoning about recursive probabilistic programs. Pages 672–681 of: *Logic in Computer Science (LICS)*. ACM.

Olmedo, Federico, Gretz, Friedrich, Jansen, Nils, Kaminski, Benjamin Lucien, Katoen, Joost-Pieter, and McIver, Annabelle. 2018. Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems*, **40**(1), 4:1–4:50.

Onisko, Agnieszka, Druzdzel, Marek J, and Wasyluk, Hanna. 1998. A probabilistic causal model for diagnosis of liver disorders. Pages 379–387 of: *Intelligent Information Systems (IIS)*.

Pearl, Judea. 1985. Bayesian networks: a model of self-activated memory for evidential reasoning. Pages 329–334 of: *Conference of the Cognitive Science Society*.

Rabin, Michael Oser. 1976. Probabilistic algorithms. Pages 21–39 of: Traub, Joseph Frederick (ed), *Algorithms and Complexity: New Directions and Recent Results*. Academic Press.

Ramanna, Sheela, Jain, Lakhmi C, and Howlett, Robert J. 2013. *Emerging Paradigms in Machine Learning*. Springer.

Scutari, Marco. 2017. *Bayesian Network Repository*. Accessed online at http://www.bnlearn.com July 30, 2018.

Smid, Michiel. 2000. Closest-point problems in computational geometry. Pages 877–935 of: Sack, Jörg-Rüdiger, and Urrutia, Jorge (eds), *Handbook of Computational Geometry*. North–Holland.

Solovay, Robert, and Strassen, Volker. 1977. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, **6**(1), 84–85.

Tarski, Alfred, et al. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**(2), 285–309.

Zweig, Geoffrey, and Russell, Stuart J. 1998. Speech recognition with dynamic Bayesian networks. Pages 173–180 of: *AAAI/IAAI*. AAAI Press / The MIT Press.