Reasoning about Recursive Probabilistic Programs [*]

Federico Olmedo Benjamin Lucien Kaminski Joost-Pieter Katoen Christoph Matheja
RWTH Aachen University, Germany
{federico.olmedo, benjamin.kaminski, katoen, matheja}@cs.rwth-aachen.de

Abstract

This paper presents a wp–style calculus for obtaining expectations on the outcomes of (mutually) recursive probabilistic programs. We provide several proof rules to derive one– and two–sided bounds for such expectations, and show the soundness of our wp–calculus with respect to a probabilistic pushdown automaton semantics. We also give a wp–style calculus for obtaining bounds on the expected runtime of recursive programs that can be used to determine the (possibly infinite) time until termination of such programs.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs.

Keywords recursion · probabilisitic programming · program verification · weakest pre–condition calculus · expected runtime.

1. Introduction

Uncertainty is nowadays more and more pervasive in computer science. Applications have to process inexact data from, e.g., unreliable sources such as wireless sensors, machine learning methods, or noisy biochemical reactors. Approximate computing saves resources such as e.g. energy by sacrificing "strict" correctness for applications like image processing that can tolerate some defects in the output by running them on unreliable hardware, circuits that every now and then (deliberately) produce incorrect results [4]. *Probabilistic programming* [29] is a key technique for dealing with uncertainty. Put in a nutshell, a probabilistic program takes a (prior) probability distribution as input and obtains a (posterior) distribution. Probabilistic programs are not new at all; they have been investigated by Kozen [20] and others in the early eighties. In the last years, the interest in these programs has rapidly grown. In particular, the incentive by the AI community to use probabilistic programs for describing complex Bayesian networks has boosted the field of probabilistic programming [10]. Probabilistic programs are used in, amongst others, machine learning, systems biology, security, planning and control, quantum computing, and software–defined networks. Indeed almost all programming languages, either being functional, object–oriented, logical, or imperative, in the meanwhile have a probabilistic variant.

This paper focuses on *recursive* probabilistic programs. Recursion in Bayesian networks where a variable associated with a particular domain entity can depend probabilistically on the same variable associated to a different entity, is "common and natural" [30]. Recursive probability models occur in gene regulatory networks that describe (possibly recursive) rule–based dependencies between genes. Finally, programs describing randomized algorithms are often recursive by nature. "Sherwood" algorithms exploit randomization to increase efficiency by avoiding or reducing the probability of worst–case behavior. Varying quicksort by selecting the pivot randomly (rather than doing this deterministically) avoids very uneven splits of the input array. Its worst–case runtime is the same as the average–case runtime of Hoare's deterministic quicksort since the likelihood of obtaining a quadratic worst–case is significantly lowered [24, Sec. 2.5]. A "Sherwood" variant of binary search splits the input array at a random position, and yields a similar effect—expected runtimes of worst–, average– and best–case are aligned [21, Sec. 11.4.4]. "Sherwood" techniques are also useful in selection, median finding, and hashing (such as Bloom filters).

The purpose of this paper is to provide a framework for enabling *formal reasoning about recursive probabilistic programs*. This rigorous reasoning is important to prove the *correctness* of such programs. This includes statements about the expected outcomes of recursive probabilistic programs, as well as assertions about their termination probability. These are challenging problems. For instance, consider the (at first sight simple) recursive program:

$$P_{\mathsf{rec}_3} \;\triangleright\; \{\mathsf{skip}\}\;[1/2]\;\{\mathsf{call}\;P_{\mathsf{rec}_3};\;\mathsf{call}\;P_{\mathsf{rec}_3};\;\mathsf{call}\;P_{\mathsf{rec}_3}\}$$

which terminates immediately with probability $1/2$ or invokes itself three times otherwise. It turns out that this program terminates with (irrational) probability $\frac{\sqrt{5}-1}{2}$ —the reciprocal of the golden ratio.

Correctness proofs of the "Sherwood" versions of quicksort and binary search do exist but typically rely on mathematical ad–hoc reasoning about expected values. The aim of this paper is to enable such proofs by means of formal verification of the algorithm itself.

Besides correctness, our interest is in analyzing the *expected runtime* of recursive probabilistic programs in a rigorous manner. This enables obtaining insight in their *efficiency* and moreover provides a method to show whether the expected time until termination is finite or infinite—a crucial difference for probabilistic programs [9, 17]. Again, analyses of expected runtimes of recursive randomized algorithms do exist using standard mathematics [24, Sec. 2.5], probabilistic recurrence relations [19], or dedicated techniques for divide–and–conquer algorithms [6], usually taking for granted—far from trivial—relationships between the underlying random variables. Here the aim is to do this from first principles by formal verification techniques, directly on the program code.

To accomplish these goals, this paper presents two weakest pre–condition–style calculi for reasoning about recursive probabilistic programs. The first calculus is an extension of McIver and Mor-

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

LICS '16, July 05–08, 2016, New York, NY, USA.
Copyright © 2016 ACM ISBN 978-1-4503-4391-6/16/07…$15.00.
http://dx.doi.org/10.1145/2933575.2935317

gan's calculus [23] for non–recursive programs and enables obtaining expectations on the outcomes of (mutually) recursive probabilistic programs. Compared to an existing extension with recursion [22], our approach provides a clear separation between syntax and semantics. We prove the soundness of our wp–calculus with respect to a probabilistic pushdown automaton semantics. This is complemented by a set of proof rules to derive one– and two–sided bounds for expected outcomes of recursive programs. We illustrate the usage of these proof rules by analyzing the termination probability of the example program above. Subsequently, we provide a variant of our wp–style calculus for obtaining bounds on the expected runtime of probabilistic programs. This extends our recent approach [18] towards treating recursive programs. The application of this calculus includes proving positive almost–sure termination, i.e., does a program terminate with probability one in finite expected time? Our framework enables (in a very succinct way) establishing a (well–known) relationship between the expected runtime of a probabilistic program with its termination behavior: If an (abort–free) program has finite expected runtime, then it terminates almost–surely. We provide a set of proof rules for expected runtimes and show the applicability of our approach by proving several correctness properties as well as the expected runtime of the 'Sherwood' variant of binary search.

Organization of the paper. Section 2 presents our probabilistic programming language with recursion. Section 3 presents the wp–style semantics for reasoning about program correctness. Section 4 introduces several proof rules for reasoning about the correctness of recursive programs. Section 5 presents the expected runtime transformer together with proof rules for recursive programs. Section 6 describes an operational probabilistic pushdown automata semantics and relates it to the wp–style semantics. Section 7 discusses some extensions of the results presented in the previous sections. Section 8 presents a detailed analysis of the 'Sherwood' variant of binary search. Finally, Section 9 discusses related work and Section 10 concludes. Detailed proofs of all the results are provided in an extended version of the paper [28].

2. Programming Model

To model our probabilistic recursive programs we consider a simple imperative language à la Dijkstra's Guarded Command Language (GCL) [7] with two additional features: First, a (binary) probabilisitic choice operator to endow our programs with a probabilistic behavior. For instance, the program

$$\{x \coloneqq x{+}1\}\ [1/3]\ \{x \coloneqq x{-}1\}$$

either increases x with probability $1/3$ or decreases it with probability $2/3 = 1 - 1/3$. Second, we allow for procedure calls. For simplicity, our development assumes the presence of only a single procedure, say P. We defer the treatment of multiple (possibly mutually recursive) procedures to Section 7.

Formally, a *command* of our language, coined pRGCL, is defined by the following grammar:

$$\begin{array}{rll}
\mathcal{C} ::= & \mathsf{skip} & \text{no–op} \\
\mid & \mathcal{V} \coloneqq \mathcal{E} & \text{assignment} \\
\mid & \mathsf{abort} & \text{abortion} \\
\mid & \mathsf{if}\ (\mathcal{E})\ \{\mathcal{C}\}\ \mathsf{else}\ \{\mathcal{C}\} & \text{conditional branching} \\
\mid & \{\mathcal{C}\}\ [p]\ \{\mathcal{C}\} & \text{probabilistic choice} \\
\mid & \mathsf{call}\ P & \text{procedure call} \\
\mid & \mathcal{C};\ \mathcal{C} & \text{sequential composition}
\end{array}$$

We assume a set $\mathcal{V}$ of program *variables* and a set $\mathcal{E}$ of *expressions* over program variables. As usual, we assume that program *states* are variable valuations, *i.e.* mappings from variables to values; let $\mathcal{S}$ be the set of program states. Finally, we also assume an interpretation function $[\![\mathcal{E}]\!]$ for expressions that maps program states to values.

No–op, assignments, conditionals and sequential composition are standard. $\{c_1\}\ [p]\ \{c_2\}$ represents a probabilistic choice: it behaves as c_1 with probability p and as c_2 with probability $1{-}p$.[1] Finally call P makes a (possibly recursive) call to procedure P.

For our development we assume that procedure P manipulates the global program state and we thus dispense with parameters and return statements for passing information across procedure calls. The declaration of P consists then of its body and we use $P \triangleright c$ to denote that $c \in \mathcal{C}$ is the body of P. We say that a command is *closed* if it contains no procedure calls.

A pRGCL *program* is then given by a pair $\langle c, \mathcal{D}\rangle$, where $c \in \mathcal{C}$ is the "main" command and $\mathcal{D}\colon \{P\} \to \mathcal{C}$ is the declaration of P.[2] In order not to clutter the notation, when c is closed we simply write c for program $\langle c, \mathcal{D}\rangle$, for any declaration $\mathcal{D}$.

Example 1. To illustrate the use of our language consider the following declaration of a (faulty) recursive procedure for computing the factorial of a natural number stored in x:

$$\begin{array}{l}
P_{\mathsf{fact}} \triangleright\ \mathsf{if}\ (x \leq 0)\ \{y \coloneqq 1\}\ \mathsf{else} \\
\qquad\qquad \big\{\ \{x \coloneqq x{-}1;\ \mathsf{call}\ P_{\mathsf{fact}};\ x \coloneqq x{+}1\}\ [5/6] \\
\qquad\qquad\quad \{x \coloneqq x{-}2;\ \mathsf{call}\ P_{\mathsf{fact}};\ x \coloneqq x{+}2\};\ y \coloneqq y \cdot x \big\}
\end{array}$$

In each recursive call x is decreased either by one or two, with probability $5/6$ and $1/6$, respectively. Therefore some factors might be missing in the computation of the factorial of x. △

As a final remark, observe that the language does not support guarded loops in a native way because they can be simulated. Concretely, the usual guarded loop while (E) do $\{c\}$ is simulated by the recursive procedure $P_{\mathsf{while}} \triangleright \mathsf{if}\ (E)\ \{c;\ \mathsf{call}\ P_{\mathsf{while}}\}\ \mathsf{else}\ \{\mathsf{skip}\}$.

3. Weakest Pre–Expectation Semantics

Inspired by Kozen [20], McIver and Morgan [22] generalized Dijkstra's weakest pre–condition semantics to (a variant of) pRGCL. In particular, they defined the semantics of recursive programs using fixed point techniques. In this section we present a different approach where the behavior of a recursive program is defined as the limit of its finite approximations (or truncations) and prove it equivalent to their definition based on fixed points.

3.1 Definition

The wp-semantics over pRGCL generalizes Dijkstra's weakest precondition semantics over GCL twofold: First, instead of being predicates over program states, pre– and post–conditions are now (non–negative) real–valued functions over program states. Secondly, instead of merely evaluating a (boolean–valued) post–condition in the final state(s) of a program, we now *measure* the expected value of a (real–valued) post–condition w.r.t. the distribution of final states.[3] Formally, if $f\colon \mathcal{S} \to \mathbb{R}^{\geq 0}$ we let

$$\mathsf{wp}[c, \mathcal{D}](f) \ \triangleq\ \lambda s\boldsymbol{.}\, \mathbf{E}_{[\![c,\mathcal{D}]\!](s)}\,(f)\ ,$$

where $[\![c, \mathcal{D}]\!](s)$ denotes the distribution of final states from executing $\langle c, \mathcal{D}\rangle$ in initial state s and $\mathbf{E}_{[\![c,\mathcal{D}]\!](s)}\,(f)$ denotes the expected value of f w.r.t. the distribution of final states $[\![c, \mathcal{D}]\!](s)$. Consider

[1] Due to the discrete nature of binary probabilistic choices, pRGCL is restricted to discrete probability spaces.

[2] We chose the declaration of P to be a mapping from a singleton and not the mere body of P because this minimizes the changes to accommodate the subsequent treatment to multiple procedures.

[3] Strictly speaking, we consider *sub*–distributions of final states, where the missing mass captures the probability of non-termination.

for instance program

$$c_{\mathsf{coins}}\colon \quad \{x \coloneqq 0\}\;[1/2]\;\{x \coloneqq 1\}; \{y \coloneqq 0\}\;[1/3]\;\{y \coloneqq 1\}$$

that flips a pair of fair and biased coins. We have

$$\begin{aligned}\mathsf{wp}[c_{\mathsf{coins}}](f) \;=\; \lambda s\boldsymbol{.}\; &\tfrac{1}{6}\, f(s[x,y/0,0]) + \tfrac{1}{3}\, f(s[x,y/0,1]) \\ &+ \tfrac{1}{6}\, f(s[x,y/1,0]) + \tfrac{1}{3}\, f(s[x,y/1,1])\ ,\end{aligned}$$

where $s[x_1,\ldots,x_n/v_1,\ldots,v_n]$ represents the state obtained by updating in s the value of variables $x_1,\ldots,x_n$ to $v_1,\ldots,v_n$, respectively. As above, when c is closed, we usually write $\mathsf{wp}[c]$ instead of $\mathsf{wp}[c,\mathcal{D}]$, as a declaration $\mathcal{D}$ plays no role.

Observe that, in particular, if $[A]$ denotes the indicator function of a predicate A over program states, $\mathsf{wp}[c,\mathcal{D}]([A])(s)$ gives the probability of (terminating and) establishing A after executing $\langle c,\mathcal{D}\rangle$ from state s. For instance we can determine the probability that the above program c_{coins} establishes $x = y$ from state s through

$$\mathsf{wp}[c_{\mathsf{coins}}]([x{=}y])(s) \;=\; \tfrac{1}{6}\cdot 1 + \tfrac{1}{3}\cdot 0 + \tfrac{1}{6}\cdot 0 + \tfrac{1}{3}\cdot 1 \;=\; \tfrac{1}{2}\ .$$

Moreover, for a deterministic program c that from state s terminates in state s', $[\![c,\mathcal{D}]\!](s)$ is the Dirac distribution that concentrates all its mass in s' and $\mathsf{wp}[c,\mathcal{D}]\big([A]\big)(s)$ reduces to $1\cdot[A]\,(s')$, which gives 1 if $s' \models A$ and 0 otherwise. This yields the classical weakest pre–condition semantics of ordinary sequential programs.

To reason about partial program correctness, pRGCL also admits a liberal version of the transformer $\mathsf{wp}[\,\cdot\,]$, namely $\mathsf{wlp}[\,\cdot\,]$. In the same vein as for ordinary sequential programs, $\mathsf{wp}[c,\mathcal{D}]([A])(s)$ gives the probability that program $\langle c,\mathcal{D}\rangle$ terminates and establishes event A from state s, while $\mathsf{wlp}[c,\mathcal{D}]([A])(s)$ gives the probability that $\langle c,\mathcal{D}\rangle$ terminates and establishes A, or diverges.

Formally, the transformer wp operates on unbounded, so–called *expectations* in $\mathbb{E} \triangleq \{f \mid f\colon \mathcal{S} \to [0,\,\infty]\}$, while the transformer wlp operates on bounded expectations in $\mathbb{E}_{\leq 1} \triangleq \{f \mid f\colon \mathcal{S} \to [0,1]\}$. Our expectation transformers have thus type $\mathsf{wp}[\,\cdot\,]\colon \mathbb{E} \to \mathbb{E}$ and $\mathsf{wlp}[\,\cdot\,]\colon \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$.[4] In the probabilistic setting pre– and post–conditions are thus referred to as *pre–* and *post–expectations*.

Notation. We use boldface for constant expectations, *e.g.* $\mathbf{1}$ denotes the constant expectation $\lambda s\boldsymbol{.}\,1$. Given an arithmetical expression E over program variables we write E for the expectation that in states s returns $[\![E]\!](s)$. Given a Boolean expression G over program variables let $[G]$ denote the $\{0,1\}$–valued expectation that on state s returns 1 if $[\![G]\!](s) = \mathsf{true}$ and 0 if $[\![G]\!](s) = \mathsf{false}$. Finally, given variable x, expression E and expectation f we use $f[x/E]$ to denote the expectation that on state s returns $f(s[x/[\![E]\!](s)])$. Moreover, "$\preceq$" denotes the pointwise order between expectations, *i.e.* $f_1 \preceq f_2$ iff $f_1(s) \leq f_2(s)$ for all states $s \in \mathcal{S}$.

3.2 Inductive Characterization

McIver and Morgan [22] showed that the expectation transformers wp and wlp can be defined by induction on the program's structure. We now recall their result, taking an alternative approach to handle recursion: While McIver and Morgan use fixed point techniques, we follow *e.g.* Hehner [13] and define the semantics of a recursive procedure as the limit of an approximation sequence. We believe that this approach is sometimes more intuitive and closer to the operational view of programs.

In the same way as the semantics of loops is defined as the limit of their finite unrollings, we define the semantics of recursive procedures as the limit of their finite inlinings. Formally, the *n-th inlining* $\mathsf{call}^{\mathcal{D}}_{n}\,P$ of procedure P w.r.t. declaration $\mathcal{D}$ is defined inductively by

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

$\boldsymbol{c}$	$\mathsf{wp}[\boldsymbol{c},\boldsymbol{\mathcal{D}}](\boldsymbol{f})$
skip	f
$x \coloneqq E$	$f[x/E]$
abort	$\mathbf{0}$
if $(G)\,\{c_1\}$ else $\{c_2\}$	$[G]\cdot\mathsf{wp}[c_1,\mathcal{D}](f) + [\neg G]\cdot\mathsf{wp}[c_2,\mathcal{D}](f)$
$\{c_1\}\;[p]\;\{c_2\}$	$p\cdot\mathsf{wp}[c_1,\mathcal{D}](f) + (1{-}p)\cdot\mathsf{wp}[c_2,\mathcal{D}](f)$
call P	$\sup_n \mathsf{wp}[\mathsf{call}^{\mathcal{D}}_{n}\,P](f)$
$c_1; c_2$	$\mathsf{wp}[c_1,\mathcal{D}]\,\big(\mathsf{wp}[c_2,\mathcal{D}](f)\big)$
$\boldsymbol{c}$	$\mathsf{wlp}[\boldsymbol{c},\boldsymbol{\mathcal{D}}](\boldsymbol{f})$
abort	$\mathbf{1}$
call P	$\inf_n \mathsf{wlp}[\mathsf{call}^{\mathcal{D}}_{n}\,P](f)$

Figure 1. Expectation transformer semantics of pRGCL programs. The $\mathsf{wlp}[\,\cdot\,]$ transformer follows the same rules as $\mathsf{wp}[\,\cdot\,]$, expect for abort and procedure calls. Sum, product, supremum and infimum over expectations are all defined pointwise.

where $c[\mathsf{call}\,P/c']$ denotes the syntactic replacement of every occurrence of $\mathsf{call}\,P$ in c by c'.[5] The family of commands $\mathsf{call}^{\mathcal{D}}_{n}\,P$ define a sequence of approximations to $\mathsf{call}\,P$ where $\mathsf{call}^{\mathcal{D}}_{0}\,P$ is the "poorest" approximation, while the larger the n, the more precise the approximation becomes. Observe that, in general, $\mathsf{call}^{\mathcal{D}}_{n+1}\,P$ mimics the exact behavior of $\mathsf{call}\,P$ for all executions that finish after at most n recursive calls.

The expectation transformer semantics over pRGCL is provided in Figure 1. The action of transformers on procedure calls is defined as the limit of their action over the n-th inlining of the procedures. For the rest of the language constructs, we follow McIver and Morgan [22]. Let us briefly explain each of the rules. $\mathsf{wp}[\mathsf{skip},\mathcal{D}]$ behaves as the identity since skip has no effect. The pre–expectation of an assignment is obtained by updating the program state and then applying the post–expectation, *i.e.* $\mathsf{wp}[x \coloneqq E,\mathcal{D}]$ takes post–expectation f to pre–expectation $f[x/E] = \lambda s\boldsymbol{.}\, f(s[x/[\![E]\!](s)])$. $\mathsf{wp}[\mathsf{abort},\mathcal{D}]$ maps any post–expectation to the constant pre–expectation $\mathbf{0}$. Observe that expectation $\mathbf{0}$ is the probabilistic counterpart of predicate false. $\mathsf{wp}[\mathsf{if}\,(G)\,\{c_1\}\,\mathsf{else}\,\{c_2\},\mathcal{D}]$ behaves either as $\mathsf{wp}[c_1,\mathcal{D}]$ or $\mathsf{wp}[c_2,\mathcal{D}]$ according to the evaluation of G. $\mathsf{wp}[\{c_1\}\,[p]\,\{c_2\},\mathcal{D}]$ is obtained as a convex combination of $\mathsf{wp}[c_1,\mathcal{D}]$ and $\mathsf{wp}[c_2,\mathcal{D}]$, weighted according to p. $\mathsf{wp}[\mathsf{call}\,P,\mathcal{D}]$ behaves as the limit of wp on the sequence of finite truncations (or inlinings) of P. We take the supremum because the sequence is increasing. Observe that we advertently include no declaration in $\mathsf{wp}[\mathsf{call}^{\mathcal{D}}_{n}\,P](f)$ because $\mathsf{call}^{\mathcal{D}}_{n}\,P$ is a closed command for every n. Finally, $\mathsf{wp}[c_1; c_2,\mathcal{D}]$ is obtained as the functional composition of $\mathsf{wp}[c_1,\mathcal{D}]$ and $\mathsf{wp}[c_2,\mathcal{D}]$. The wlp transformer follows the same rules as wp, except for the abort statement and procedure calls. $\mathsf{wlp}[\mathsf{abort},\mathcal{D}]$ takes any post–expectation to pre–expectation $\mathbf{1}$. (Expectation $\mathbf{1}$ is the probabilistic counterpart of predicate true.) $\mathsf{wlp}[\mathsf{call}\,P,\mathcal{D}]$ also behaves as the limit of wlp on the sequence of finite truncations of P. This time we take the infimum because the sequence is decreasing.

Example 2. Reconsider $c_{\mathsf{coins}} = c_1;\ c_2$ from Section 3.1 with

$$c_1\colon\; \{x \coloneqq 0\}\;[1/2]\;\{x \coloneqq 1\} \quad\text{and}\quad c_2\colon\; \{y \coloneqq 0\}\;[1/3]\;\{y \coloneqq 1\}\ .$$

We use our weakest pre–expectation calculus to formally determine the probability that the outcome of the two coins coincide:

$$\begin{aligned}&\mathsf{wp}[c_{\mathsf{coins}}]([x{=}y]) \\ &\quad = \mathsf{wp}[c_1]\big(\mathsf{wp}[c_2]([x{=}y])\big)\end{aligned}$$

[4] The transformer wlp is well–typed because $\mathsf{wlp}[c,\mathcal{D}](f)(s) \leq \sup_{s'} f(s')$ for every state s.

[5] The formal definition of this syntactic replacement proceeds by a routine induction on the structure of c.

$$\begin{aligned}
&= \mathsf{wp}[c_1]\big(\tfrac{1}{3}\cdot \mathsf{wp}[y := 0]([x{=}y]) + \tfrac{2}{3}\cdot \mathsf{wp}[y := 1]([x{=}y])\big)\\
&= \mathsf{wp}[c_1]\big(\tfrac{1}{3}\cdot [x{=}0] + \tfrac{2}{3}\cdot [x{=}1]\big)\\
&= \tfrac{1}{2}\cdot \mathsf{wp}[x := 0]\big(\tfrac{1}{3}\cdot [x{=}0] + \tfrac{2}{3}\cdot [x{=}1]\big)\\
&\quad + \tfrac{1}{2}\cdot \mathsf{wp}[x := 1]\big(\tfrac{1}{3}\cdot [x{=}0] + \tfrac{2}{3}\cdot [x{=}1]\big)\\
&= \tfrac{1}{2}\cdot \big(\tfrac{1}{3}\cdot [0{=}0] + \tfrac{2}{3}\cdot [0{=}1]\big) + \tfrac{1}{2}\cdot \big(\tfrac{1}{3}\cdot [1{=}0] + \tfrac{2}{3}\cdot [1{=}1]\big)\\
&= \tfrac{1}{2}\cdot \tfrac{\mathbf{1}}{\mathbf{3}} + \tfrac{1}{2}\cdot \tfrac{\mathbf{2}}{\mathbf{3}} \;=\; \tfrac{\mathbf{1}}{\mathbf{2}} \qquad \triangle
\end{aligned}$$

The transformers wp and wlp enjoy several appealing algebraic properties, which we summarize below.

Lemma 3.1 (Basic properties of w(l)p**).** *For every program $\langle c, \mathcal{D}\rangle$, every f_1, f_2, and increasing ω–chain $f_0 \preceq f_1 \preceq \cdots$ in $\mathbb{E}$, g_1, g_2, and every decreasing ω–chain $g_0 \succeq g_1 \succeq \cdots$ in $\mathbb{E}_{\leq 1}$, and scalars $\alpha_1, \alpha_2 \in \mathbb{R}_{\geq 0}$ it holds:*

Continuity:	$\sup_n \mathsf{wp}[c,\mathcal{D}](f_n) \;=\; \mathsf{wp}[c,\mathcal{D}](\sup_n f_n)$
	$\inf_n \mathsf{wlp}[c,\mathcal{D}](g_n) \;=\; \mathsf{wlp}[c,\mathcal{D}](\inf_n g_n)$
Monotonicity:	$f_1 \preceq f_2 \implies \mathsf{wp}[c,\mathcal{D}](f_1) \preceq \mathsf{wp}[c,\mathcal{D}](f_2)$
	$g_1 \preceq g_2 \implies \mathsf{wlp}[c,\mathcal{D}](g_1) \preceq \mathsf{wlp}[c,\mathcal{D}](g_2)$
Linearity:	$\mathsf{wp}[c,\mathcal{D}](\alpha_1\cdot f_1 + \alpha_2 \cdot f_2)$
	$=\; \alpha_1\cdot \mathsf{wp}[c,\mathcal{D}](f_1) + \alpha_2\cdot \mathsf{wp}[c,\mathcal{D}](f_2)$
Preserv. of $\mathbf{0}$,$\mathbf{1}$:	$\mathsf{wp}[c,\mathcal{D}](\mathbf{0}) = \mathbf{0}$ *and* $\mathsf{wlp}[c,\mathcal{D}](\mathbf{1}) = \mathbf{1}$

Program termination. Since the termination behavior of a program is given by the probability that it establishes true, we can readily use the transformer wp to reason about program termination. It suffices to consider the weakest pre–expectation of the program w.r.t. post–expectation $[\mathsf{true}] = \mathbf{1}$. Said otherwise, $\mathsf{wp}[c,\mathcal{D}](\mathbf{1})(s)$ gives the termination probability of program $\langle c, \mathcal{D}\rangle$ from state s. In particular, if the program terminates with probability 1, we say that it *terminates almost–surely*.

3.3 Characterization based on Fixed Points

Next we use a continuity argument on the transformer w(l)p to prove that its action on recursive procedures can also be defined using fixed point techniques. This alternative characterization rests on a subsidiary transformer $\mathsf{w(l)p}[\,\cdot\,]^\sharp_\theta$, which is a slight variant of $\mathsf{w(l)p}[\,\cdot\,]$. The main difference between these transformers is the mechanism that they use to give semantics to procedure calls: $\mathsf{w(l)p}[\,\cdot\,]$ relies on a declaration $\mathcal{D}$, while $\mathsf{w(l)p}[\,\cdot\,]^\sharp_\theta$ relies on a so–called (*liberal*) *semantic environment* $\theta\colon \mathbb{E}\to\mathbb{E}$ ($\theta\colon \mathbb{E}_{\leq 1}\to\mathbb{E}_{\leq 1}$) which is meant to directly encode the semantics of procedure calls. Then $\mathsf{w(l)p}[\mathsf{call}\, P]^\sharp_\theta(f)$ gives $\theta(f)$, while for all other program constructs c, $\mathsf{w(l)p}[c]^\sharp_\theta(f)$ agrees with $\mathsf{w(l)p}[c](f)$; see [28] for details. For technical reasons, in the remainder of our development we will consider only continuous semantic environments in $\mathsf{SEnv} \triangleq \{f \mid f\colon \mathbb{E}\to\mathbb{E} \text{ is upper continuous}\}$ and $\mathsf{LSEnv} \triangleq \{f \mid f\colon \mathbb{E}_{\leq 1}\to\mathbb{E}_{\leq 1} \text{ is lower continuous}\}$.[6] This is a natural assumption since we are interested only in semantic environments that are obtained as the w(l)p–semantics of a pRGCL program, which are continuous by Lemma 3.1.

The semantics of recursive procedures can now be readily given as the fixed point of a semantic environment transformer.

Theorem 3.1 (Fixed point characterization for procedure calls)**.** *Given a declaration $\mathcal{D}\colon \{P\}\to\mathcal{C}$ for procedure P,*

$$\mathsf{wp}[\mathsf{call}\, P, \mathcal{D}] \;=\; \mathit{lfp}_{\sqsubseteq}\left(\lambda\theta{:}\mathsf{SEnv}.\,\mathsf{wp}[\mathcal{D}(P)]^\sharp_\theta\right)$$

$$\mathsf{wlp}[\mathsf{call}\, P, \mathcal{D}] \;=\; \mathit{gfp}_{\sqsubseteq}\left(\lambda\theta{:}\mathsf{LSEnv}.\,\mathsf{wlp}[\mathcal{D}(P)]^\sharp_\theta\right)\;.$$

[6] A (liberal) semantic environment θ is *upper* (*lower*) *continuous* iff for every increasing ω-chain $f_0 \preceq f_1 \preceq \cdots$ (decreasing ω-chain $f_0 \succeq f_1 \succeq \cdots$), $\sup_n \theta(f_n) = \theta(\sup_n f_n)$ ($\inf_n \theta(f_n) = \theta(\inf_n f_n)$).

The fixed points above are taken w.r.t. the pointwise order "$\sqsubseteq$" over semantic environments: given $\theta_1, \theta_2 \in \mathsf{SEnv}$ (resp. $\theta_1, \theta_2 \in \mathsf{LSEnv}$), $\theta_1 \sqsubseteq \theta_2$ iff $\theta_1(f) \preceq \theta_2(f)$ for all $f\in\mathbb{E}$ (resp. $f\in\mathbb{E}_{\leq 1}$).

Theorem 3.1 reveals an inherent difference between the complexities of reasoning about loops and general recursion: The semantics of loops can be given as the fixed point of an expectation transformer (see *e.g.* [25]), while the semantics of recursion requires the fixed point of a (*higher order*) environment transformer. As a consequence, proving our results for recursive programs requires new insights not present in the proofs for while–programs. This fact was already noticed by Dijkstra [7, p. xvii] and later on confirmed by Nelson [26, p. 517] for non–probabilisitic programs.

4. Correctness of Recursive Programs

In this section we introduce some proof rules for effectively reasoning about the behavior of recursive programs. For that we require the notion of *constructive derivability*. Given logical formulae A and B, we use $A \Vdash B$ to denote that B can be derived assuming A. In particular, we will consider claims of the form

$$\mathsf{w(l)p}[\mathsf{call}\, P](f_1) \bowtie g_1 \;\Vdash\; \mathsf{w(l)p}[c](f_2) \bowtie g_2\;,$$

where $\bowtie \in \{\preceq, \succeq\}$, f_1, g_1 give the specification of call P and f_2, g_2 the specification of c. Notice that in such a claim we omit any procedure declaration as the derivation is independent of P's body.

Our first two rules are extensions of well–known rules for ordinary recursive programs (see *e.g.* [15]) to a probabilistic setting:

$$\frac{\mathsf{wp}[\mathsf{call}\, P](f) \preceq g \;\Vdash\; \mathsf{wp}[\mathcal{D}(P)](f) \preceq g}{\mathsf{wp}[\mathsf{call}\, P, \mathcal{D}](f) \preceq g}\;[\text{wp-rec}]$$

$$\frac{g \preceq \mathsf{wlp}[\mathsf{call}\, P](f) \;\Vdash\; g \preceq \mathsf{wlp}[\mathcal{D}(P)](f)}{g \preceq \mathsf{wlp}[\mathsf{call}\, P, \mathcal{D}](f)}\;[\text{wlp-rec}]$$

So for proving that a procedure call satisfies a specification (given by f, g), it suffices to show that the procedure's body satisfies the specification, assuming that the recursive calls in the body do, too.

Example 3. Reconsider the procedure P_{rec_3} with declaration

$$\mathcal{D}(P_{\mathsf{rec}_3}):\quad \{\mathsf{skip}\}\;[1/2]\;\{\mathsf{call}\, P_{\mathsf{rec}_3};\ \mathsf{call}\, P_{\mathsf{rec}_3};\ \mathsf{call}\, P_{\mathsf{rec}_3}\}$$

presented in the introduction. We prove that it terminates with probability *at most* $\varphi = \frac{\sqrt{5}-1}{2}$ from any initial state. Formally, this is captured by $\mathsf{wp}[\mathsf{call}\, P, \mathcal{D}](\mathbf{1}) \preceq \boldsymbol{\varphi}$. To prove this, we apply rule [wp-rec]. We must then establish the derivability claim

$$\mathsf{wp}[\mathsf{call}\, P](\mathbf{1}) \preceq \boldsymbol{\varphi} \;\Vdash\; \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1}) \preceq \boldsymbol{\varphi}\;.$$

The derivation goes as follows:

$$\begin{aligned}
&\mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1})\\
=\;&\quad\{\text{def. of wp}\}\\
&\tfrac{1}{2}\cdot \mathsf{wp}[\mathsf{skip}](\mathbf{1}) + \tfrac{1}{2}\cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3};\ \mathsf{call}\, P_{\mathsf{rec}_3};\ \mathsf{call}\, P_{\mathsf{rec}_3}](\mathbf{1})\\
=\;&\quad\{\text{def. of wp}\}\\
&\tfrac{\mathbf{1}}{\mathbf{2}} + \tfrac{1}{2}\cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3};\ \mathsf{call}\, P_{\mathsf{rec}_3}]\big(\mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}](\mathbf{1})\big)\\
\preceq\;&\quad\{\text{assumption, monot. of wp}\}\\
&\tfrac{\mathbf{1}}{\mathbf{2}} + \tfrac{1}{2}\cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3};\mathsf{call}\, P_{\mathsf{rec}_3}](\boldsymbol{\varphi})\\
=\;&\quad\{\text{def. of wp, scalab. of wp twice}\}\\
&\tfrac{\mathbf{1}}{\mathbf{2}} + \tfrac{1}{2}\,\varphi\cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}]\big(\mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}](\mathbf{1})\big)\\
\preceq\;&\quad\{\text{assumption, monot. of wp}\}\\
&\tfrac{\mathbf{1}}{\mathbf{2}} + \tfrac{1}{2}\,\varphi\cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}](\boldsymbol{\varphi})
\end{aligned}$$

$$\begin{aligned}
= \quad & \{\text{scalab. of } \mathsf{wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi^2 \cdot \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}](\mathbf{1}) \\
\preceq \quad & \{\text{assumption, monot. of } \mathsf{wp}\} \\
& \tfrac{1}{2} + \tfrac{1}{2}\,\varphi^3 = \boldsymbol{\varphi}
\end{aligned}$$

An appealing feature of our approximation semantics is that to prove the following soundness result we do not need to resort to a continuity argument on the expectation transformers.

Theorem 4.1 (Soundness of rules [w(l)p-rec]). *Rules* [wp-rec] *and* [wlp-rec] *are sound w.r.t. the* w(l)p *semantics in Figure 1.*

Rules [w(l)p-rec] allow deriving only one–sided bounds for the weakest (liberal) pre–expectation of a procedure call. It is also possible to derive two–sided bounds using the following rules:

$$\frac{l_0 = \mathbf{0}, \qquad u_0 = \mathbf{0}, \qquad l_n \preceq \mathsf{wp}[\mathsf{call}\, P](f) \preceq u_n \;\Vdash\; l_{n+1} \preceq \mathsf{wp}[\mathcal{D}(P)](f) \preceq u_{n+1}}{\sup_n l_n \preceq \mathsf{wp}[\mathsf{call}\, P, \mathcal{D}](f) \preceq \sup_n u_n}\,[\mathsf{wp\text{-}rec}_\omega]$$

$$\frac{l_0 = \mathbf{1}, \qquad u_0 = \mathbf{1}, \qquad l_n \preceq \mathsf{wlp}[\mathsf{call}\, P](f) \preceq u_n \;\Vdash\; l_{n+1} \preceq \mathsf{wlp}[\mathcal{D}(P)](f) \preceq u_{n+1}}{\inf_n l_n \preceq \mathsf{wlp}[\mathsf{call}\, P, \mathcal{D}](f) \preceq \inf_n u_n}\,[\mathsf{wlp\text{-}rec}_\omega]$$

In constrast to rules [w(l)p-rec], these rules require exhibiting two sequences of expectations $\langle l_n \rangle$ and $\langle u_n \rangle$ rather than a single expectation g to bound the weakest (liberal) pre–expectation of a procedure call. Intuitively l_n (u_n) represents a lower (upper) bound for the weakest pre–expectation of the n-inlining of the procedure, *i.e.* from the premises of the rules we will have $l_n \preceq \mathsf{w(l)p}[\mathsf{call}^{\mathcal{D}}_n\, P](f) \preceq u_n$ for all $n \in \mathbb{N}$.

Observe that both rules can be specialized to reason about one–sided bounds. For instance, by setting $u_{n+1} = \boldsymbol{\infty}$ in [wp-rec$_\omega$] we can reason about lower bounds of $\mathsf{wp}[\mathsf{call}\, P, \mathcal{D}](f)$, which is not supported by rule [wp-rec]. Similarly, by taking $l_n = \mathbf{0}$ in rule [wlp-rec$_\omega$] we can reason about upper bounds of $\mathsf{wlp}[\mathsf{call}\, P, \mathcal{D}](f)$.

Example 4. Reconsider the procedure P_{rec_3} from Example 3. Now we prove that the procedure terminates with probability *at least* $\varphi = \frac{\sqrt{5}-1}{2}$ from any initial state. To this end, we rely on the fact that φ can be characterized by the asymptotic behavior of the sequence $\langle \varphi_n \rangle$, where $\varphi_0 = 0$ and $\varphi_{n+1} = \frac{1}{2} + \frac{1}{2}\,\varphi_n^3$. In symbols, $\varphi = \sup_n \varphi_n$. We wish then to prove that

$$\sup_n \boldsymbol{\varphi_n} \;\preceq\; \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}, \mathcal{D}](\mathbf{1})\;.$$

To establish this formula we apply the one side variant of rule [wp-rec$_\omega$] to reason about lower bounds of $\mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}, \mathcal{D}](\mathbf{1})$, that is, we implicitly take $u_{n+1} = \boldsymbol{\infty}$. We must then establish

$$\boldsymbol{\varphi_n} \preceq \mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}](\mathbf{1}) \;\Vdash\; \boldsymbol{\varphi_{n+1}} \preceq \mathsf{wp}[\mathcal{D}(P_{\mathsf{rec}_3})](\mathbf{1})\;.$$

The derivation follows the same steps as those taken in Example 3 to give upper bounds on $\mathsf{wp}[\mathsf{call}\, P_{\mathsf{rec}_3}, \mathcal{D}](\mathbf{1})$. Combining the result proved with that in Example 3, we conclude that $\varphi = \frac{\sqrt{5}-1}{2}$ is the exact termination probability of $\langle \mathsf{call}\, P_{\mathsf{rec}_3}, \mathcal{D} \rangle$. △

Lastly, we can establish the correctness our rules.

Theorem 4.2 (Soundness of rules [w(l)p-rec$_\omega$]). *Rules* [w(l)p-rec$_\omega$] *are sound w.r.t. the* w(l)p *semantics in Figure 1.*

To conclude the section we would like to point out that the rule [wp-rec$_\omega$] is related to previous work on proof rules. It can be viewed as a generalization of Jones's loop rule [16] to the case of recursion (even though Jones originally presented a one–sided version) and as an adaptation of Audebaud and Paulin-Mohring's rule [1] to our weakest pre–expectation semantics. The counterpart of the rule for partial correctness, on the other hand, is, to the best of our knowledge, novel.

c	$\mathsf{ert}[c, \mathcal{D}](t)$
skip	$\mathbf{1} + t$
$x := E$	$\mathbf{1} + t[x/E]$
abort	$\mathbf{0}$
if $(G)\,\{c_1\}$ else $\{c_2\}$	$\mathbf{1} + [G] \cdot \mathsf{ert}\,[c_1, \mathcal{D}](t) + [\neg G] \cdot \mathsf{ert}\,[c_2, \mathcal{D}](t)$
$\{c_1\}\ [p]\ \{c_2\}$	$p \cdot \mathsf{ert}\,[c_1, \mathcal{D}](t) + (1{-}p) \cdot \mathsf{ert}\,[c_2, \mathcal{D}](t)$
call P	$\mathit{lfp}_{\sqsubseteq}\left(\lambda\eta\colon \mathsf{RtEnv}.\, \underline{\mathbf{1}} \oplus \mathsf{ert}\,[\mathcal{D}(P)]^{\sharp}_{\eta}\right)(t)$
$c_1; c_2$	$\mathsf{ert}\,[c_1, \mathcal{D}]\left(\mathsf{ert}\,[c_2, \mathcal{D}](t)\right)$

Figure 2. Rules for the expected runtime transformer ert. $\mathit{lfp}_{\sqsubseteq}(F)$ denotes the least fixed point of transformer $F\colon \mathsf{RtEnv} \to \mathsf{RtEnv}$ w.r.t. the pointwise order "$\sqsubseteq$" between runtime environments.

5. The Expected Runtime of Programs

To further our study of recursive probabilistic programs we now develop a calculus for reasoning about the expected or average runtime of pRGCL programs. This calculus builds upon our previous work in [18] and is able to handle recursive procedures.

5.1 The Expected Runtime Transformer ert

We assume a runtime model where executing a skip statement, an assignment, evaluating the guard in a conditional branching and invoking a procedure[7] consumes one unit of time. On the other hand, combining two programs by means of a sequential composition or a probabilistic choice consumes no additional time other than that consumed by the original programs. Likewise, halting a program execution with an abort statement consumes no unit of time.

Since the runtime of a program varies according to the initial state from which it is executed, our aim is to associate to each program $\langle c, \mathcal{D} \rangle$ a mapping that takes each state s to the expected time until $\langle c, \mathcal{D} \rangle$ terminates on s. Such mappings will range over the set of *runtimes* $\mathbb{T} \triangleq \left\{ t \mid t\colon \mathcal{S} \to [0, \infty] \right\}$.[8]

To associate each program to its runtime we use a continuation passing style formalized by the transformer

$$\mathsf{ert}\,[\,\cdot\,]\colon \mathbb{T} \to \mathbb{T}\;.$$

If $t \in \mathbb{T}$ represents the runtime of the computation that follows program $\langle c, \mathcal{D} \rangle$, then $\mathsf{ert}\,[c, \mathcal{D}](t)$ represents the overall runtime of $\langle c, \mathcal{D} \rangle$, plus the computation following $\langle c, \mathcal{D} \rangle$. Runtime t is usually referred to as the *continuation* of $\langle c, \mathcal{D} \rangle$. In particular, by setting the continuation of a program to zero we recover the runtime of the plain program. That is, for every initial state s,

$$\mathsf{ert}\,[c, \mathcal{D}](\mathbf{0})(s)$$

gives the expected runtime of program $\langle c, \mathcal{D} \rangle$ from state s.

The transformer $\mathsf{ert}\,[c, \mathcal{D}]$ is defined by induction on the structure of c, following the rules in Figure 2. The rules are defined so as to correspond to the aforementioned runtime model. That is, $\mathsf{ert}\,[c, \mathcal{D}](\mathbf{0})$ captures the expected number of assignments, guard evaluations, procedure calls and skip statements in the execution of $\langle c, \mathcal{D} \rangle$. Most rules are self–explanatory. $\mathsf{ert}\,[\mathsf{skip}, \mathcal{D}]$ adds one unit of time to the continuation since skip does not modify the program state and its execution takes one unit of time. $\mathsf{ert}\,[x := E, \mathcal{D}]$ also adds one unit of time, but to the continuation evaluated in the state resulting from the assignment. $\mathsf{ert}\,[\mathsf{abort}, \mathcal{D}]$ yields always the

[7] Loosely speaking, the overall runtime of a procedure call is then one plus the runtime of executing the procedure's body.

[8] Strictly speaking, the set of runtimes $\mathbb{T}$ coincides with the set of unbounded expectations $\mathbb{E}$ but we prefer to distinguish the two sets since they are to represent different objects. We will, however, keep the same notations for runtimes as for expectations, for example $t[x/E]$, $t_1 \preceq t_2$, etc.

constant runtime $\mathbf{0}$ since abort aborts any subsequent program execution (making their runtime irrelevant) and consumes no time. $\mathsf{ert}\,[\mathsf{if}\ (G)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, \mathcal{D}]$ adds one unit of time to the runtime of either of its branches, depending on the value of the guard. $\mathsf{ert}\,[\{c_1\}\ [p]\ \{c_2\}, \mathcal{D}]$ gives the weighted average between the runtime of its branches, each of them weighted according to its probability. $\mathsf{ert}\,[c_1; c_2, \mathcal{D}]$ first applies $\mathsf{ert}\,[c_2, \mathcal{D}]$ to the continuation and then $\mathsf{ert}\,[c_1, \mathcal{D}]$ to the resulting runtime of this application. Finally, $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]$ is defined using fixed point techniques.

To understand the intuition behind the definition of $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]$ recall that call P consumes one unit of time more than the body of P. To capture this fact we make use of the auxiliary runtime transformer $\mathsf{ert}\,[\,\cdot\,]^\sharp_\eta : \mathbb{T} \to \mathbb{T}$ (*cf.* expectation transformer $\mathsf{wp}[\,\cdot\,]^\sharp_\theta$). This transformer behaves as ert except that for defining its action on a procedure call, it relies on a so–called *runtime environment* η in $\mathsf{RtEnv} \triangleq \{\eta \mid \eta\colon \mathbb{T} \to \mathbb{T}$ is upper continuous$\}$ instead of on a procedure declaration. Concretely, $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]^\sharp_\eta$ takes continuation t to $\eta(t)$ and for all other program constructs, $\mathsf{ert}\,[\,\cdot\,]^\sharp_\eta$ follows the same rule as $\mathsf{ert}\,[\,\cdot\,]$. Using this transformer we can (implicitly) define $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]$ by the equation

$$\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}] \;=\; \underline{\mathbf{1}} \oplus \mathsf{ert}\,[\mathcal{D}(P)]^\sharp_{\mathsf{ert}[\mathsf{call}\ P, \mathcal{D}]} \ ,$$

where $\underline{\mathbf{1}} = \lambda t : \mathbb{T}.\,\mathbf{1}$ represents the constantly $\mathbf{1}$ runtime transformer and "$\oplus$" the point–wise sum between runtime transformers, *i.e.* for $\gamma_1, \gamma_2\colon \mathbb{T} \to \mathbb{T}$, we let $(\gamma_1 \oplus \gamma_2)(t) \triangleq \gamma_1(t) + \gamma_2(t)$. The above equation leads to the fixed point characterization of $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]$ in Figure 2.

We remark that, as opposed to w(l)p, it is not posible to define the action $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}]$ of ert on a procedure call in terms of its action $\mathsf{ert}\,[\mathsf{call}^{\mathcal{D}}_n\ P]$ on the finite inlinings. This is because when computing $\mathsf{ert}\,[\mathsf{call}^{\mathcal{D}}_n\ P](t)$, to be correct the transformer should add one unit of time each time a procedure call was inlined, and this is not recoverable from $\mathsf{call}^{\mathcal{D}}_n\ P$.[9]

This concludes our definition of the transformer ert. We devote the remainder of the section to study several of its properties. We begin with Theorem 5.1 summarizing some algebraic properties.

Theorem 5.1 (Basic properties of ert)**.** *For any program $\langle c, \mathcal{D}\rangle$, any constant runtime $\mathbf{k} = \lambda s.\, k$ for $k \in \mathbb{R}_{\geq 0}$, any $t, u \in \mathbb{T}$, and any increasing ω–chain $t_0 \preceq t_1 \preceq \cdots$ of runtimes, it holds:*

Continuity:	$\sup_n \mathsf{ert}\,[c, \mathcal{D}](t_n) \;=\; \mathsf{ert}\,[c, \mathcal{D}](\sup_n t_n)$;
Monotonicity:	$t \preceq u \implies \mathsf{ert}\,[c, \mathcal{D}](t) \;\preceq\; \mathsf{ert}\,[c, \mathcal{D}](u)$;
Propagation of constants:	$\mathsf{ert}\,[c, \mathcal{D}](\mathbf{k} + t) \;=\; \mathbf{k} + \mathsf{ert}\,[c, \mathcal{D}](t)$ *provided $\langle c, \mathcal{D}\rangle$ is* abort*–free;*
Preservation of infinity:	$\mathsf{ert}\,[c, \mathcal{D}](\infty) \;=\; \infty$ *provided $\langle c, \mathcal{D}\rangle$ is* abort*–free.*

The next result establishes a connection between ert and wp.

Theorem 5.2. *For every program $\langle c, \mathcal{D}\rangle$ and runtime t,*

$$\mathsf{ert}\,[c, \mathcal{D}](t) \;=\; \mathsf{ert}\,[c, \mathcal{D}](\mathbf{0}) + \mathsf{wp}[c, \mathcal{D}](t)\ .$$

Proof. By induction on the program structure, considering the stronger version of the statement

$$\mathsf{ert}\,[c, \mathcal{D}](t_1 + t_2) \;=\; \mathsf{ert}\,[c, \mathcal{D}](t_1) + \mathsf{wp}[c, \mathcal{D}](t_2)\ . \qquad \square$$

Theorem 5.2 allows giving a very short proof of a well–known result relating expected runtimes and termination probabilities: If a program has finite expected runtime, it terminates almost surely.

[9] If we adopt a model where the runtime of a procedure call coincides with the runtime of its body, we could just take $\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}](t) = \sup_n \mathsf{ert}\,[\mathsf{call}^{\mathcal{D}}_n\ P](t)$.

Theorem 5.3. *For every* abort*–free program $\langle c, \mathcal{D}\rangle$ and initial state s of the program,*

$$\mathsf{ert}\,[c, \mathcal{D}](\mathbf{0})(s) < \infty \implies \mathsf{wp}[c, \mathcal{D}](\mathbf{1})(s) = 1\ .$$

Proof. By instantiating Theorem 5.2 with $t = \mathbf{1}$ and using the propagation of constants property of ert (Theorem 5.1) to decompose $\mathsf{ert}\,[c, \mathcal{D}](\mathbf{1})$ as $\mathbf{1} + \mathsf{ert}\,[c, \mathcal{D}](\mathbf{0})$. $\square$

Observe that in Theorem 5.3 we cannot drop the abort–free requirement on the program. To see this, consider the program $c = \{\mathsf{skip}\}\,[1/2]\,\{\mathsf{abort}\}$. The program has a finite runtime ($\mathsf{ert}\,[c](\mathbf{0}) = \mathbf{1/2} < \infty$) and terminates, however, with probability less than one ($\mathsf{wp}[c](\mathbf{1}) = \mathbf{1/2} < \mathbf{1}$). Moreover, observe that Theorem 5.3 is only valid on the stated direction: A probabilistic program can terminate almost–surely and require, still, an expected infinite time to reach termination. This phenomenon is illustrated, for instance, by the one dimensional random walk; see *e.g.* [18, §7].

Even though Theorem 5.3 constitutes a well–known and natural result on probabilistic programs (closely related to the Borel–Cantelli lemma), our contribution here is to give the first fully formal proof of such a result.

5.2 Proof Rules for Recursive Programs

The runtime of procedure calls, which includes, in particular, recursive programs, is defined using fixed points. To avoid reasoning about fixed points we propose some proof rules based on invariants.

We show that an adaptation of the proof rules for procedure calls from our wp–calculus is sound for the ert–calculus. The rules are:

$$\frac{\mathsf{ert}\,[\mathsf{call}\ P](t) \preceq \mathbf{1}{+}u \;\Vdash\; \mathsf{ert}\,[\mathcal{D}(P)](t) \preceq u}{\mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}](t) \preceq \mathbf{1}{+}u}\ [\mathsf{eet\text{-}rec}]$$

$$\frac{\begin{array}{c} l_0 = \mathbf{0}, \qquad\qquad u_0 = \mathbf{0}, \\ \mathbf{1}{+}l_n \preceq \mathsf{ert}\,[\mathsf{call}\ P](t) \preceq \mathbf{1}{+}u_n \\ \Vdash\; l_{n+1} \preceq \mathsf{ert}\,[\mathcal{D}(P)](t) \preceq u_{n+1} \end{array}}{\mathbf{1}{+}\sup_n l_n \preceq \mathsf{ert}\,[\mathsf{call}\ P, \mathcal{D}](t) \preceq \mathbf{1}{+}\sup_n u_n}\ [\mathsf{eet\text{-}rec}_\omega]$$

Compared to the proof rules from the wp–calculus, these proof rules require incrementing by one unit some of the bounds. Loosely speaking, this is because the runtime of a procedure call is one plus the runtime of its body, whereas the semantics of a procedure call fully agrees with the semantics of its body.

Example 5. To illustrate the use of the rules, consider the faulty factorial procedure with declaration

$$\mathcal{D}(P_{\mathsf{fact}})\colon\ \mathsf{if}\ (x \leq 0)\ \{\{y := 1\}\ \mathsf{else}\ \{\{c_1\}\ [5/6]\ \{c_2\};\ y := y{\cdot}x\}\ ,$$

where $c_1 = x := x{-}1$; call P_{fact}; $x := x{+}1$ and $c_2 = x := x{-}2$; call P_{fact}; $x := x{+}2$. We prove that on input $x = k \geq 0$, the expected runtime of the procedure is $2 + \alpha_k$, where

$$\alpha_k \;=\; \frac{1}{49}\Big(121 + 210k + 432\big(-\tfrac{1}{6}\big)^{k+1}\Big)\ .$$

Since the term $432(-1/6)^{k+1}$ is negligible, we can approximate the procedure's runtime by $4.5 + 4.3k$. We can formally capture our exact runtime assertion by

$$\mathsf{ert}\,[\mathsf{call}\ P_{\mathsf{fact}}, \mathcal{D}](\mathbf{0}) \;=\; \mathbf{1} + \sup_n t_n\ ,$$

where $t_n = \mathbf{1}{+}[x < 0]{\cdot}\mathbf{1}{+}[0 \leq x \leq n]{\cdot}\alpha_x + [x > n]{\cdot}\alpha_{n+1}$. To see this, observe that the sequence $\langle \alpha_k \rangle$ is increasing and therefore, $\sup_n t_n = \mathbf{1} + [x < 0] \cdot \mathbf{1} + [0 \leq x] \cdot \alpha_x$. We prove the runtime assertion using rule $[\mathsf{eet\text{-}rec}_\omega]$ with instantiations $t = \mathbf{0}$ and $l_n = u_n = t_n$ for $n \geq 1$. We have to discharge the premise

$$\mathsf{ert}\,[\mathsf{call}\ P_{\mathsf{fact}}](\mathbf{0}) = \mathbf{1} + t_n \;\Vdash\; \mathsf{ert}\,[\mathcal{D}(P_{\mathsf{fact}})](\mathbf{0}) = t_{n+1}\ .$$

Since some simple calculations yield

$$\frac{\mathsf{stmt}(\ell) = \mathsf{skip} \quad \mathsf{succ}_1(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma} \langle \ell',\, s\rangle}\,[\text{skip}] \qquad \frac{\mathsf{stmt}(\ell) = x \coloneqq E \quad \mathsf{succ}_1(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma} \langle \ell',\, s[x \mapsto s(E)]\rangle}\,[\text{assign}] \qquad \frac{\mathsf{stmt}(\ell) = \mathsf{abort}}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma} \langle \ell,\, s\rangle}\,[\text{abort}]$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\,(G)\,\{c_1\}\,\mathsf{else}\,\{c_2\} \quad s \models G \quad \mathsf{succ}_1(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma} \langle \ell',\, s\rangle}\,[\text{if1}] \qquad \frac{\mathsf{stmt}(\ell) = \mathsf{if}\,(G)\,\{c_1\}\,\mathsf{else}\,\{c_2\} \quad s \not\models G \quad \mathsf{succ}_2(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma} \langle \ell',\, s\rangle}\,[\text{if2}]$$

$$\frac{\mathsf{stmt}(\ell) = \{c_1\}\,[p]\,\{c_2\} \quad \mathsf{succ}_1(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,p,\,\gamma} \langle \ell',\, s\rangle}\,[\text{prob1}] \qquad \frac{\mathsf{stmt}(\ell) = \{c_1\}\,[p]\,\{c_2\} \quad \mathsf{succ}_2(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1-p,\,\gamma} \langle \ell',\, s\rangle}\,[\text{prob2}]$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{call}\,P \quad \mathsf{succ}_1(\ell) = \ell'}{\langle \ell,\, s\rangle \xrightarrow{\gamma,\,1,\,\gamma\cdot\ell'} \langle \mathsf{init}(\mathcal{D}(P)),\, s\rangle}\,[\text{call}] \qquad \frac{}{\langle \downarrow,\, s\rangle \xrightarrow{\ell',\,1,\,\varepsilon} \langle \ell',\, s\rangle}\,[\text{return}] \qquad \frac{}{\langle \downarrow,\, s\rangle \xrightarrow{\gamma_0,\,1,\,\gamma_0} \langle \mathsf{Term},\, s\rangle}\,[\text{terminate}]$$

Figure 3. Rules for defining an operational semantics for pRGCL programs. For sequential composition there is no dedicated rule as the control flow is encoded via the succ_1 and the succ_2 functions.

$$\begin{aligned}\mathsf{ert}\,[\mathcal{D}(P_{\mathsf{fact}})](\mathbf{0}) \;&=\; \mathbf{1} + [x \leq 0]\cdot\mathbf{1} \\ &\quad + [x > 0]\cdot\left(\tfrac{5}{6}\cdot\mathsf{ert}\,[c_1](\mathbf{1}) + \tfrac{1}{6}\cdot\mathsf{ert}\,[c_2](\mathbf{1})\right)\,,\end{aligned}$$

our next step is to compute $\mathsf{ert}\,[c_1](\mathbf{1})$ (the calculations are identical for $\mathsf{ert}\,[c_2](\mathbf{1})$). To do so, we rely on assumption $\mathsf{ert}\,[\mathsf{call}\,P](\mathbf{0}) = \mathbf{1} + t_n$ and the propagation of constants property of ert.

$$\begin{aligned}\mathsf{ert}\,[c_1](\mathbf{1}) \;&=\; \mathsf{ert}\,[x \coloneqq x{-}1;\ \mathsf{call}\,P_{\mathsf{fact}}]\left(\mathsf{ert}\,[x \coloneqq x{+}1](\mathbf{1})\right) \\ &=\; \mathbf{2} + \mathsf{ert}\,[x \coloneqq x{-}1;\ \mathsf{call}\,P_{\mathsf{fact}}](\mathbf{0}) \\ &=\; \mathbf{2} + \mathsf{ert}\,[x \coloneqq x{-}1](\mathbf{1} + t_n) \\ &=\; \mathbf{4} + t_n[x/x+1]\end{aligned}$$

The derivation then concludes by showing that

$$\begin{aligned} t_{n+1} \;&=\; \mathbf{1} + [x \leq 0]\cdot\mathbf{1} \\ &\quad + [x > 0]\cdot\left(\tfrac{5}{6}\left(\mathbf{4} + t_n[x/x{+}1]\right) + \tfrac{1}{6}\left(\mathbf{4} + t_n[x/x{+}2]\right)\right)\,,\end{aligned}$$

which after some term reordering reduces to proving that $\alpha_0 = 1$, $\alpha_1 = 7$ and $\alpha_{k+2} = 5 + \frac{5}{6}\alpha_{k+1} + \frac{1}{6}\alpha_k$. △

We conclude the section establishing the soundness of the rules.

Theorem 5.4 (Soundness of rules [eet-rec], [eet-rec$_\omega$])**.** *Rules* [eet-rec] *and* [eet-rec$_\omega$] *are sound w.r.t. the* ert*–calculus in Figure 2.*

6. Operational Semantics

We provide an operational semantics for pRGCL programs in terms of pushdown Markov chains with rewards (PRMC) [3] and prove the transformer wp to be sound with respect to this semantics. Due to space limitations, this section contains an informal introduction only. Corresponding formal definitions are found in [28].

For simplicity, we assume a canonical labeling for each command $c \in \mathcal{C}$ together with auxiliary functions init, succ_1, succ_2 and stmt determining the initial location, the first and second successor of a location and the program statement corresponding to a label. As an example, the labels attached to each statement of program c from Example 3 are as follows:

$$c:\quad \{\mathsf{skip}^1\}\ [1/2]^2\ \{\mathsf{call}\,P^3;\ \mathsf{call}\,P^4;\ \mathsf{call}\,P^5\ \}\,.$$

The definition of the auxiliary functions is straightforward. For instance, we have $\mathsf{init}(c) = 2$, $\mathsf{succ}_1(1) = \downarrow$, $\mathsf{succ}_2(2) = 3$, and $\mathsf{stmt}\,(2) = c$, where $\downarrow$ is a special symbol indicating termination of a procedure. Moreover, label Term stands for termination of the whole program.

Our operational semantics of pRGCL programs is given as an execution relation, where each step is of the form

$$\langle \ell,\, s\rangle \xrightarrow{\gamma,\,p,\,\gamma'} \langle \ell',\, s'\rangle\,.$$

Here, ℓ, ℓ' are program labels, $s, s' \in \mathcal{S}$ are program states, γ is a program label being popped from and γ' a finite sequence of labels being pushed on the stack, respectively. $p \in [0,1]$ denotes the probability of executing this step.

This execution relation corresponds to the transition relation of a PRMC, where each pair $\langle \ell,\, s\rangle$ is a state and the stack alphabet is given by the set of all labels of a given pRGCL program. Moreover, given $f \in \mathbb{E}$, a reward of $f(s)$ is assigned to each state of the form $\langle \mathsf{Term},\, s\rangle$. Otherwise, the reward of a state is 0. Figure 3 shows the rules defining the operational semantics of pRGCL programs. The rules in Figure 3 are self–explanatory. In case of a procedure call, the calls successor label is pushed on the stack and execution continues with the called procedure. Whenever a procedure terminates, i.e. reaches a state $\langle \downarrow,\, s\rangle$, and the stack is non–empty, a return address is popped from and execution continues at this address.

Figure 4 shows the PRMC of example program c. The initial state is 2 (the probabilistic choice). Say the right branch is chosen; we move to 3. The statement at 3 is a call, and the address after the call is 4; so 4 is pushed and the procedure body is reentered. Say now the left branch is chosen; we move to 1 (the skip) and then terminate, i.e. we move to $\downarrow$. Recall that return address 4 is on top of the stack; 4 is popped, we move to 4 to continue execution.

The expected reward that PRMC $\mathfrak{P}$ associated to program $\langle c, \mathcal{D}\rangle$ reaches a set of target states $\mathcal{T}$ from initial state $\langle \ell,\, s\rangle$ is defined as

$$\mathsf{ExpRew}^{\mathfrak{P}_s^f[\![c,\mathcal{D}]\!]}(\mathcal{T}) \;=\; \sum_{\pi \in \Pi(\langle \ell,\, s\rangle, \mathcal{T})} \mathsf{Prob}^{\mathfrak{P}}(\pi)\cdot\mathsf{rew}(\pi)\,,$$

where π is a path from $\langle \ell,\, s\rangle$ to some target state, $\mathsf{Prob}^{\mathfrak{P}}(\pi)$ is the probability of π and $\mathsf{rew}(\pi)$ is the reward collected along π.

We are now in a position to state the relationship between the operational model and the denotational semantics:

Theorem 6.1 (Correspondence Theorem)**.** *Let $c \in \mathcal{C}$, $f \in \mathbb{E}$, and $\mathcal{T} = \{\langle \mathsf{Term},\, s\rangle \mid s \in \mathcal{S}\}$* [10]*. Then for each $s \in \mathcal{S}$, we have*

$$\mathsf{ExpRew}^{\mathfrak{P}_s^f[\![c,\mathcal{D}]\!]}(\mathcal{T}) \;=\; \mathsf{wp}[c,\mathcal{D}](f)(s)\,.$$

In the spirit of [11] a similar result can be obtained for wlp. For that one needs a liberal expected reward being defined as the expected reward plus the probability of not reaching the target states at all. One can then show a similar correspondence to wlp.

7. Extensions

Mutual recursion. Both our wp– and ert–calculus can be extended to handle multiple procedures. Say we want to handle m

[10] $\mathcal{T}$ denotes the set of states representing successful termination of the pushdown automaton.

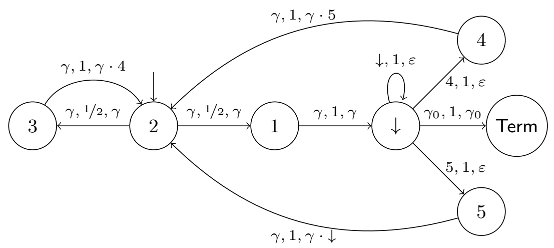
$$\begin{tikzcd}
3 \arrow[r, bend left, "\gamma, 1, \gamma\cdot 4"] & 2 \arrow[l, "\gamma, 1/2, \gamma"'] \arrow[r, "\gamma, 1/2, \gamma"] & 1 \arrow[r, "\gamma, 1, \gamma"] & \downarrow \arrow[loop above, "\downarrow, 1, \varepsilon"] \arrow[ur, "4, 1, \varepsilon"] \arrow[r, "\gamma_0, 1, \gamma_0"] \arrow[dr, "5, 1, \varepsilon"'] & \mathsf{Term} \\
 & & & & 4 \arrow[ulll, bend right, "\gamma, 1, \gamma\cdot 5"'] \\
 & & & & 5 \arrow[ulll, bend left, "\gamma, 1, \gamma\cdot\downarrow"]
\end{tikzcd}$$

Figure 4. PRMC of program c from Example 3. Since c affects no variables, the second component of states is omitted.

(possibly mutually recursive) procedures $P_1, \ldots, P_m$ with declaration $\mathcal{D} \in \mathcal{C}^m$. The definition of $\mathsf{wp}[\mathsf{call}\, P_i, \mathcal{D}]$ remains the same, we only need to adapt the definition of the n-inlining $\mathsf{call}_n^{\mathcal{D}}\, P_i$ of procedure P_i as to inline the calls of all procedures:

$$\mathsf{call}_{n+1}^{\mathcal{D}}\, P_i \;=\; \mathcal{D}(P)[\mathsf{call}\, P_1/\mathsf{call}_n^{\mathcal{D}}\, P_1, \ldots, \mathsf{call}\, P_m/\mathsf{call}_n^{\mathcal{D}}\, P_m] \;.$$

As for the ert–calculus, a runtime environment is now a tuple $\eta = (\eta_1, \ldots, \eta_m)$, where η_i is meant to provide the behavior of procedure P_i in $\mathsf{ert}\,[\cdot]^\sharp_\eta$, *i.e.* $\mathsf{ert}\,[\mathsf{call}\, P_i]^\sharp_\eta = \eta_i$. The action of ert on procedure calls is then defined simultaneously as[11]

$$\big(\mathsf{ert}\,[\mathsf{call}\, P_1, \mathcal{D}]\,, \ldots, \mathsf{ert}\,[\mathsf{call}\, P_m, \mathcal{D}]\big) \;=\; \mathit{lfp}\Big(\lambda\eta\textbf{.}\; \big(\underline{\mathbf{1}} \oplus \mathsf{ert}\,[\mathcal{D}(P_1)]^\sharp_\eta\,, \ldots, \underline{\mathbf{1}} \oplus \mathsf{ert}\,[\mathcal{D}(P_m)]^\sharp_\eta\big)\Big) \;.$$

The proof rules for reasoning about procedure calls in both calculi are easily adapted. We show only the case of [wp-rec]; the others admit a similar adaptation.

$$\frac{\begin{array}{c}\mathsf{wp}[\mathsf{call}\, P_1](f_1) \preceq g_1, \ldots, \mathsf{wp}[\mathsf{call}\, P_m](f_m) \preceq g_m \;\Vdash\; \mathsf{wp}[\mathcal{D}(P_1)](f_1) \preceq g_1 \\ \vdots \\ \mathsf{wp}[\mathsf{call}\, P_1](f_1) \preceq g_1, \ldots, \mathsf{wp}[\mathsf{call}\, P_m](f_m) \preceq g_m \;\Vdash\; \mathsf{wp}[\mathcal{D}(P_m)](f_m) \preceq g_m\end{array}}{\mathsf{wp}[\mathsf{call}\, P_i, \mathcal{D}](f_i) \preceq g_i \quad \text{for all } i = 1 \ldots m}$$

The rule reasons about all the procedures simultaneously. Roughly speaking, the rule premise requires deriving the specification g_i for the body of each procedure P_i, assuming the corresponding specification for each procedure call in it. The rule conclusion establishes the specification of the set of procedures altogether.

Random samplings. All our results remain valid if the pRGCL language allows for random samplings (from distributions with discrete support). In a random sampling $x := \mu$, μ represents a probability distribution which is sampled and its outcome is assigned to program variable x. In Section 8 we exploit this extension to model a probabilistic variant of the binary search.

Alternative runtime models. The ert–calculus can be easily adapted to capture alternative runtime models. For instance we can capture the model where we are interested in counting only the number of procedure calls and also more fine–grained models such as that where the time consumed by an assignment (or guard evaluation) depends on some notion of *size* of the expression being assigned (guard being evaluated). Likewise, the ert–calculus can be easily adapted so as to take into account the costs of flipping the (possibly biased) coin from probabilistic choices.

Soundness of the* ert*–calculus. We can also establish the soundness of the ert–calculus w.r.t. the operational semantics based on PRMC. This only requires changes in the reward function.

[11] For determining the *least* fixed point, environments are compared component-wise, *i.e.* $(\eta_1, \ldots, \eta_m) \sqsubseteq (\nu_1, \ldots, \nu_m)$ iff $\eta_i \sqsubseteq \nu_i$ for all $i = 1 \ldots m$.

8. Case Study

In this section we show the applicability of our approach analyzing a probabilistic, so–called Sherwood [21], variant of the binary search. The main difference w.r.t. the classical version is that in each recursive call the pivot element is picked uniformly at random from the remaining array, aligning this way worst–, best– and average–case of the algorithm runtime.

The algorithm we analyze searches for value val in array $a[\mathit{left}\,..\,\mathit{right}]$. It is encoded by procedure B with declaration $\mathcal{D}$ presented in Figure 5. We use random assignment $\mathit{mid} := \mathsf{uniform}(\mathit{left},\, \mathit{right})$ to model the random election of the pivot. For simplicity, we assume that the random assignment is performed in constant time 1 if $\mathit{left} \leq \mathit{right}$ and that it diverges if $\mathit{left} > \mathit{right}$.

Partial correctness. We verify the following partial correctness property: When B is invoked in a state where $\mathit{left} \leq \mathit{right}$, $a[\mathit{left}\,..\,\mathit{right}]$ is sorted, and val occurs in $a[\mathit{left}\,..\,\mathit{right}]$, then the invocation of B stores in mid the index where val lies. Formally,

$$\begin{aligned} g &\preceq \mathsf{wlp}[\mathsf{call}\, B, \mathcal{D}](f), \text{ with} \\ g &= [\mathit{left} \leq \mathit{right}] \cdot \big[\mathsf{sorted}(\mathit{left},\, \mathit{right})\big] \\ &\quad \cdot \big[\exists x \in [\mathit{left},\, \mathit{right}] \colon a[x] = \mathit{val}\big] \\ f &= \big[a[\mathit{mid}] = \mathit{val}\big]\;, \end{aligned}$$

where $\big[\mathsf{sorted}(y,\, z)\big]$ is the indicator function of $a[y\,..\,z]$ being sorted. In order to prove $g \preceq \mathsf{wlp}[\mathsf{call}\, B](f)$ we apply rule [wlp-rec]. We are then left to prove

$$g \preceq \mathsf{wlp}[\mathsf{call}\, B](f) \;\Vdash\; g \preceq \mathsf{wlp}[\mathcal{D}(B)](f)\;.$$

The way in which we propagate post–expectation f from the exit point of the procedure till its entry point, obtaining pre–expectation g, is fully detailed in Figure 5. To do so we use assumption $g \preceq \mathsf{wlp}[\mathsf{call}\, B](f)$ and monotonicity of wlp.

Dually, we can verify that when val is not in the array, the value of $a[\mathit{mid}]$ after termination of B is different from val. A detailed derivation of this property is provided in [28].

Expected runtime. We perform a runtime analysis of the algorithm for those inputs where val does not occur in the array. Under this assumption we can distinguish two cases: either val is smaller than every element in the array or larger than all of them.

For the first case we show that the expected runtime of the algorithm is upper bounded by $\mathbf{1} + u$, with

$$\begin{aligned} u \;=\; &[\mathit{left} > \mathit{right}] \cdot \boldsymbol{\infty} + \mathbf{3} \\ &+ [\mathit{left} < \mathit{right}] \cdot (\mathbf{5} \cdot H_{\mathit{right}-\mathit{left}+1} - \mathbf{5/2})\;, \end{aligned}$$

and H_k being the k-th harmonic number. Formally, we show that

$$\mathsf{ert}\,[\mathsf{call}\, B](\mathbf{0}) \;\preceq\; \mathbf{1} + u$$

applying rule [eet-rec]. We must then establish

$$\mathsf{ert}\,[\mathsf{call}\, B](\mathbf{0}) \preceq \mathbf{1} + u \;\Vdash\; \mathsf{ert}\,[\mathcal{D}](\mathbf{0}) \preceq u\;.$$

The details of this derivation are provided in Figure 6.

Similarly, when val is greater than every element in the array, the expected runtime is upper bounded by $\mathbf{1} + u$, with

$$\begin{aligned} u \;=\; &[\mathit{left} > \mathit{right}] \cdot \boldsymbol{\infty} + \mathbf{3} \\ &+ [\mathit{left} < \mathit{right}] \cdot (\mathbf{6} \cdot H_{\mathit{right}-\mathit{left}+1} - \mathbf{3})\;. \end{aligned}$$

The verification for this case is analogous therefore omitted.

Combining the two cases we conclude that when the sought–after value does not occur in the array, the algorithm terminates in expected time in $\Theta\big(\log n\big)$, where $n = \mathit{right} - \mathit{left} + 1$ is the size of the array, since $H_k \in \Theta(\log k)$.

```
$g \preceq \frac{[left<right]}{right-left+1} \sum_{i=left}^{right} \left( \begin{array}{l} [a[i]<val] \cdot g[left/\mathsf{min}(i+1, right)] \\ + [a[i]>val] \cdot g[right/\mathsf{max}(i-1, left)] \\ + [a[i]=val] \end{array} \right)$
    $+ [left = right] \cdot [a[left] = val]$
$\boldsymbol{mid} := \mathbf{uniform}(\boldsymbol{left}, \boldsymbol{right})$;
$[left < right] \cdot \Big( [a[mid] < val] \cdot g[left/\cdots]$
    $+ [a[mid] > val] \cdot g[right/\cdots]$
    $+ [a[mid] = val] \Big) + [left \geq right] \cdot f$
if ($\boldsymbol{left} < \boldsymbol{right}$){
    $[a[mid]<val] \cdot g[left/\cdots] + [a[mid]>val] \cdot g[right/\cdots]$
        $+ [a[mid]=val] \cdot f$
    if ($\boldsymbol{a}[\boldsymbol{mid}] < \boldsymbol{val}$){
        $g[left/\mathsf{min}(mid+1, right)]$
        $\boldsymbol{left} := \mathbf{min}(\boldsymbol{mid} + \mathbf{1}, \boldsymbol{right})$;
        $g$
        call $B$
        $f$
    } else {
        $[a[mid] > val] \cdot g[right/\cdots] + [a[mid] \leq val] \cdot f$
        if ($\boldsymbol{a}[\boldsymbol{mid}] > \boldsymbol{val}$){
            $g[right/\mathsf{max}(mid-1, left)]$
            $\boldsymbol{right} := \mathbf{max}(\boldsymbol{mid} - \mathbf{1}, \boldsymbol{left})$;
            $g$
            call $B$
            $f$
        } else { $f$ skip $f$ } $f$
    } $f$
} else { $f$ skip $f$ } $f$
```

Figure 5. Declaration $\mathcal{D}$ (boldface) of the probabilistic binary search procedure B together with the proof (lightface) that call B finds the index of val when started in a sorted array $a[left\,..\,right]$ which contains value val. We write j $\boldsymbol{C}$ h for $j \preceq \mathsf{wlp}[C]\,(h)$.

```
$u = [left > right] \cdot \infty + \mathbf{3} + [left < right]$
    $\cdot \left( \mathbf{5} + \sum_{i=left}^{right} \left( \begin{array}{l} \frac{[\mathsf{min}(i+1, right) < right]}{right - left + 1} \\ \cdot (\mathbf{5} \cdot H_{right - \mathsf{min}(i+1, right) < right+1} \\ \quad - \mathbf{5/2}) \end{array} \right) \right)$
$\boldsymbol{mid} := \mathbf{uniform}(\boldsymbol{left}, \boldsymbol{right})$;
$\mathbf{2} + [left < right] \cdot \Big( \mathbf{2} + [a[mid] < val] \cdot (\mathbf{3}$
        $+ [\mathsf{min}(mid+1, right) < right]$
            $\cdot \big( \mathbf{5} \cdot H_{right - \mathsf{min}(mid+1, right)+1} - \mathbf{5/2} \big)$
    $+ [a[mid] > val] \cdot (\cdots) \Big)$
if ($\boldsymbol{left} < \boldsymbol{right}$){
    $\mathbf{3} + [a[mid] < val] \cdot u[left/\mathsf{min}(mid+1, right)]$
    $+ [a[mid] > val] \cdot (\cdots)$
    if ($\boldsymbol{a}[\boldsymbol{mid}] < \boldsymbol{val}$){
        $\mathbf{2} + u[left/\mathsf{min}(mid+1, right)]$
        $\boldsymbol{left} := \mathbf{min}(\boldsymbol{mid} + \mathbf{1}, \boldsymbol{right})$;
        $\mathbf{1} + u$
        call $B$
        $\mathbf{0}$
    } else {
        $\mathbf{2} + [a[mid] > val] \cdot (\cdots)$
        if ($\boldsymbol{a}[\boldsymbol{mid}] > \boldsymbol{val}$){
            $\mathbf{2} + u[right/\mathsf{max}(mid-1, left)]$
            $\boldsymbol{right} := \mathbf{max}(\boldsymbol{mid} - \mathbf{1}, \boldsymbol{left})$;
            $\mathbf{1} + u$
            call $B$
            $\mathbf{0}$
        } else { $\mathbf{1}$ skip $\mathbf{0}$ } $\mathbf{0}$
    } $\mathbf{0}$
} else { $\mathbf{1}$ skip $\mathbf{0}$ } $\mathbf{0}$
```

Figure 6. Runtime analysis of the probabilistic binary search procedure for the case that every value occurring in $a[left\,..\,right]$ is smaller than val. We write j $\boldsymbol{C}$ h for $j \succeq \mathsf{ert}\,[C]\,(h)$.

9. Related Work

wp–style reasoning for recursive programs. Recursion has been treated for non–probabilistic programs. Hesselink [15] provided several proof rules for recursive procedures, both for total and partial correctness. Our first two proof rules are extensions of his rules to the probabilistic setting. Predicate transformer semantics for recursive non–deterministic procedures has been provided by Bonsangue and Kok [2] and Hesselink [14]. Nipkow [27] provides an operational semantics and a Hoare logic for recursive (parameterless) non–deterministic procedures. Zhang *et al.* [32] establishes the equivalence between an operational semantics and a weakest pre–condition semantics for recursive programs in Coq. To some extent our transfer theorem between probabilistic pushdown automata and the wp–semantics can be considered as a probabilistic extension of this work.

Deductive reasoning for recursive probabilistic programs. Jones provided several proof rules for recursive probabilistic programs in her Ph.D. dissertation [16]. One of our proof rules is a generalisation of Jones' proof rule to general recursion. McIver and Morgan [22] also provide a wp–semantics of probabilistic recursive programs. While [22] use fixed point techniques, we follow *e.g.* Hehner [13] and define the semantics of a recursive procedure as the limit of an approximation sequence. In contrast to our approach based on procedures, [22] introduced recursion through the language constructor $\mathbf{rec}\ \mathcal{B}$, where $\mathcal{B}$ is a program–semantics transformer. (Intuitively $\mathcal{B}$ encodes how the recursive procedure defined (and invoked) by $\mathbf{rec}\ \mathcal{B}$ transforms the outcome of its recursive calls). Our approach provides a strict separation between program syntax and semantics. Moreover our approach based on procedure calls can model mutual recursion in a natural way (see Section 7), while the approach in [22] approach does not accommodate so naturally to such cases. Audebaud and Paulin-Mohring [1] present a mechanized method for proving properties of randomized algorithms in the Coq proof assistant. Their approach is based on higher–order logic, in particular using a monadic interpretation of programs as probabilistic distributions. Our proof rule for obtaining two–sided bounds on recursive programs is directly adapted from their work. They however do neither relate their work to an operational model nor support the analysis of expected runtimes.

Semantics of recursive probabilistic programs. Gupta *et al.* [12] consider the interplay between constraints, probabilistic choice, and recursion in the context of a (concurrent) constraint–based probabilistic programming language. They provide an operational semantics using labeled transition systems and (weak) bisimulation as well as a denotational semantics. Recursion is treated operationally by considering the limit of syntactic finite approximations. In the denotational semantics, the mixture of probabilities and constraints violates basic monotonicity properties for a standard treatment of recursion. Their main result is that the transition system semantics modulo weak bisimulation is fully abstract with respect to the input–output relation of processes. They do neither consider non–determinism nor reasoning about recursive probabilistic programs. Pfeffer and Koller [30] provide a measure–

theoretic semantics of recursive Bayesian networks and show that every recursive probabilistic relational database has a probability measure as model. This is complemented by an inference algorithm that obtains approximations by basically unfolding the recursive Bayesian network. Recently, Toronto *et al.* [31] provided a measure–theoretic semantics for a probabilistic programming language with recursion. Their interpretation of recursive programs is however restricted to (almost surely) terminating programs.

Probabilistic pushdown automata. The analysis of probabilistic pushdown automata, which correspond to the model of recursive Markov chains, has been well–investigated. Key computational problems for analyzing classes of these models can be reduced to computing the least fixed point solution of corresponding classes of monotone polynomial systems of non–linear equations. For subclasses of these models termination probabilities, ω–regular properties, and expected runtimes can be algorithmically obtained. Recent surveys are provided by Etessami [8] and Brazdil *et al.* [3]. Our transfer theorem indicates that (some of) these results are transferable to obtaining weakest pre–expectations for recursive probabilistic programs having a finite–control probabilistic push–down automata. A detailed study is outside the scope of this paper and left for future work.

10. Conclusion

We have presented two wp–calculi: one for reasoning about correctness, and one for analysing expected rum-times of recursive probabilistic programs. The wp–calculi have been related, equipped with proof rules, and exemplified by analysing a Sherwood version of binary search. A relation with a straightforward operational interpretation using pushdown Markov chains has been established. We believe that this work provides a good basis for the automation of the analysis of recursive probabilistic programs. Future work consists of applying our calculi to other recursive randomized algorithms (such as quick sort with random pivot selection). Other future work includes investigating a generalisation of Colussi's technique [5] to transform a recursive program and its correctness proof into a non-recursive program with its accompanying correctness proof. This would allow to transfer—typically simpler—correctness proofs of the recursive probabilistic programs to non-recursive ones.

Acknowledgments

We would like to thank the anonymous reviewers for pointing out the duality between transformers wp and wlp: $\mathsf{wp}[c,\mathcal{D}](\mathbf{1}-f) = \mathbf{1} - \mathsf{wlp}[c,\mathcal{D}](f)$. By exploiting this relationship, we can reduce the proof effort required to establish all our results about wlp.

References

[1] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Science of Comp. Progr.*, 74(8):568 – 589, 2009.

[2] M. Bonsangue and J. Kok. The weakest precondition calculus: Recursion and duality. *Formal Aspects of Computing*, 6(1):788–800, 1994.

[3] T. Brázdil, J. Esparza, S. Kiefer, and A. Kucera. Analyzing probabilistic pushdown automata. *Formal Methods in System Design*, 43 (2):124–163, 2013.

[4] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proc. of OOPSLA*, pages 33–52. ACM, 2013.

[5] L. Colussi. Recursion as an effective step in program development. *ACM Trans. Program. Lang. Syst.*, 6(1):55–67, Jan. 1984.

[6] B. C. Dean. A simple expected running time analysis for randomized "divide and conquer" algorithms. *Discrete Appl. Math.*, 154(1):1–5, 2006.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[8] K. Etessami. Analysis of probabilistic processes and automata theory. In *Handbook of Automata Theory*. 2016. (to appear).

[9] L. M. F. Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In *Proc. of POPL*, pages 489–501. ACM, 2015.

[10] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering (FOSE)*, pages 167–181. ACM, 2014.

[11] F. Gretz, J.-P. Katoen, and A. McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.*, 73:110–132, 2014.

[12] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In A. W. Appel and A. Aiken, editors, *Proc. of POPL*, pages 189–202. ACM, 1999.

[13] E. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11(4):287–304, 1979. .

[14] W. H. Hesselink. Predicate-transformer semantics of general recursion. *Acta Informatica*, 26(4):309–332, 1989.

[15] W. H. Hesselink. Proof rules for recursive procedures. *Formal Aspects of Computing*, 5(6):554–570, 1993. .

[16] C. Jones. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, 1989.

[17] B. L. Kaminski and J. Katoen. On the hardness of almost-sure termination. In *Prof. of MFCS, Part I*, volume 9234 of *LNCS*, pages 307–318. Springer, 2015.

[18] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *Proc. of ESOP*, LNCS, 2016. To appear.

[19] R. M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):1136–1150, 1994.

[20] D. Kozen. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.

[21] J. McConnell. *Analysis of Algorithms – An Active Learning Approach*. Jones and Bartlett Publishers, Inc., 2008.

[22] A. McIver and C. Morgan. Partial correctness for probabilistic demonic programs. *Theor. Comp. Sc.*, 266(12):513 – 541, 2001.

[23] A. McIver and C. Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems*. Springer, 2004.

[24] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[25] C. Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*. Springer, 1996.

[26] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, Oct. 1989.

[27] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Proc. of CSL*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[28] F. Olmedo, B. L. Kaminski, J. Katoen, and C. Matheja. Reasoning about recursive probabilistic programs. *CoRR*, abs/1603.02922, 2016.

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

[30] A. Pfeffer and D. Koller. Semantics and inference for recursive probability models. In *Proc. of AAAI*, pages 538–544. AAAI Press / The MIT Press, 2000.

[31] N. Toronto, J. McCarthy, and D. V. Horn. Running probabilistic programs backwards. In *Proc. of ESOP*, volume 9032 of *LNCS*, pages 53–79. Springer, 2015.

[32] X. Zhang, M. Munro, M. Harman, and L. Hu. Weakest precondition for general recursive programs formalized in Coq. In *Proc. of TPHOL*, volume 2410 of *LNCS*, pages 332–348. Springer, 2002.