Almost Sure Productivity

Alejandro Aguirre
IMDEA Software Institute, Madrid, Spain

Gilles Barthe
IMDEA Software Institute, Madrid, Spain

Justin Hsu
University College London, London, UK

Alexandra Silva
University College London, London, UK

Abstract

We introduce *Almost Sure Productivity (ASP)*, a probabilistic generalization of the productivity condition for coinductively defined structures. Intuitively, a probabilistic coinductive stream or tree is ASP if it produces infinitely many outputs with probability 1. Formally, we define ASP using a final coalgebra semantics of programs inspired by Kerstan and König. Then, we introduce a core language for probabilistic streams and trees, and provide two approaches to verify ASP: a syntactic sufficient criterion, and a decision procedure by reduction to model-checking LTL formulas on probabilistic pushdown automata.

2012 ACM Subject Classification Theory of computation → Denotational semantics, Theory of computation → Probabilistic computation, Theory of computation → Program reasoning

Keywords and phrases Coinduction, Probabilistic Programming, Productivity

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.113

Related Version The full version of this paper can be found at `https://arxiv.org/abs/1802.06283`.

Acknowledgements We thank Benjamin Kaminski, Charles Grellois and Ugo dal Lago for helpful discussions on preliminary versions of this work, and we are grateful to the anonymous reviewers for pointing out areas where the exposition could be improved. This work was initiated at the Bellairs Research Institute in Barbados, and was partially supported by NSF grant #1637532 and ERC grant ProFoundNet (#679127).

1 Introduction

The study of probabilistic programs has a long history, especially in connection with semantics [22] and verification [23, 17, 28]. Over the last decade the field of probabilistic programming has attracted renewed attention with the emergence of practical probabilistic programming languages and novel applications in machine learning, privacy-preserving data mining, and modeling of complex systems. On the more theoretical side, many semantical and syntactic tools have been developed for verifying probabilistic properties. For instance, significant attention has been devoted to termination of probabilistic programs, focusing on the complexity of the different termination classes [19], and on practical methods for proving that a program terminates [16, 25, 2, 27]. The latter class of works generally focuses on *almost sure termination*, which guarantees that a program terminates with probability 1.

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 113; pp. 113:1–113:15


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Coinductive probabilistic programming is a new computational paradigm that extends probabilistic programming to infinite objects, such as streams and infinite trees, providing a natural setting for programming and reasoning about probabilistic infinite processes such as Markov chains or Markov decision processes. Rather surprisingly, the study of coinductive probabilistic programming was initiated only recently [3], and little is known about generalizations of coinductive concepts and methods to the probabilistic setting. In this paper we consider *productivity*, which informally ensures that one can compute arbitrarily precise approximations of infinite objects in finite time. Productivity has been studied extensively for standard, non-probabilistic coinductive languages [18, 14, 1, 12, 7], but the probabilistic setting introduces new subtleties and challenges.

Contributions

Our first contribution is conceptual. We introduce *almost sure productivity* (ASP), a probabilistic counterpart to productivity. A probabilistic stream computation is almost surely productive if it produces an infinite stream of outputs with probability 1. For instance, consider the stream defined by the equation

$$\sigma = (a : \sigma) \oplus_p \sigma$$

Viewed as a program, this stream repeatedly flips a coin with bias $p \in (0, 1)$, producing the value a if the coin comes up heads and retrying if the coin comes up tails. This computation is almost surely productive since the probability it fails to produce outputs for n consecutive steps is $(1 - p)^n$, which tends to zero as n increases. In contrast, consider the stream

$$\sigma = \bar{a} \oplus_p \epsilon$$

This computation flips a single biased coin and returns an infinite stream of a's if the coin comes up heads, and the empty stream ϵ if the coin comes up tails. This process is *not* almost surely productive since its probability of outputting an infinite stream is only p, which is strictly less than 1.

We define almost sure productivity for any system that can be equipped with a final coalgebra semantics in the style of Kerstan and König [20] (Section 3). We instantiate our semantics on a core probabilistic language for computing over streams and trees (Section 4). Then, we consider two methods for proving almost sure productivity.

1. We begin with a syntactic method that assigns a real-valued measure to each expression e (Section 5). Intuitively, the measure represents the expected difference between the number of outputs produced and consumed per evaluation step of the expression. For instance, the computation that repeatedly flips a fair coin and outputs a value if the coin is heads has measure $\frac{1}{2}$ – with probability $1/2$ it produces an output, with probability 0 it produces no outputs. More complex terms in our language can also consume outputs internally, leading to possibly negative values for the productivity measure.
   We show that every expression whose measure is strictly positive is almost surely productive; the proof of soundness of the method uses concentration results from martingale theory. While simple to carry out, our syntactic method is incomplete – it does not yield any information for expressions with non-positive measure.
2. To give a more sophisticated analysis, we reduce the problem of deciding ASP to probabilistic model-checking (Section 6). We translate our programs to probabilistic pushdown automata and show that almost sure productivity is characterized by a logical formula in LTL. This fragment is known to be decidable [6], giving a sound and complete procedure for deciding ASP.

We consider more advanced generalizations and extensions in Section 7, survey related work in Section 8, and conclude in Section 9.

2 Mathematical Preliminaries

This section reviews basic notation and definitions from measure theory and category theory. Given a set A we will denote by $A_\perp$ the coproduct of A with a one-element set containing a distinguished element $\perp$, i.e., $A_\perp = A + \{\perp\}$.

Coalgebra, Monads, Kleisli categories

We assume that the reader is familiar with the notions of objects, morphisms, functors and natural transformations (see, for instance, [5]). Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$, a *coalgebra* of F is a pair (X, f) of an object $X \in \mathcal{C}$ and a morphism $f : X \to F(x)$ in $\mathcal{C}$. A *monad* is a triple (T, η, μ) of an endofunctor $T : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\eta : 1_\mathcal{C} \to T$ (the *unit*) and $\mu : T^2 \to T$ (the *multiplication*) such that $\mu \circ \mu_T = \mu \circ T\mu$ and $\mu \circ \eta_T = 1_T = \mu \circ T\eta$. Given a category $\mathcal{C}$ and a monad (T, η, μ), the *Kleisli category* $\mathcal{K}\ell(T)$ of T has as objects the objects of $\mathcal{C}$ and as morphisms $X \to Y$ the morphisms $X \to T(Y)$ in $\mathcal{C}$.

Streams, Trees, Final Coalgebra

We will denote by O^ω the set of infinite streams of elements of O (alternatively characterized as functions $\mathbb{N} \to O$). We have functions $\mathsf{head} \colon O^\omega \to O$ and $\mathsf{tail} \colon O^\omega \to O^\omega$ that enable observation of the elements of the stream. In fact, they provide O^ω with a one-step structure that is canonical: given any set S and any two functions $h \colon S \to O$ and $t \colon S \to S$ (i.e., a coalgebra $(S, \langle h, t\rangle)$ of the functor $F(X) = O \times X$) there exists a *unique* stream function associating semantics to elements of S:

$$\begin{array}{ccc} S & \overset{[\![-]\!]}{\dashrightarrow} & O^\omega \\ {\scriptstyle \langle h,t\rangle}\downarrow & & \downarrow{\scriptstyle \langle \mathsf{head},\mathsf{tail}\rangle} \\ O \times S & \overset{id\times[\![-]\!]}{\dashrightarrow} & O \times O^\omega \end{array}$$

Formally, this uniqueness property is known as *finality*: O^ω is the *final coalgebra* of the functor $F(X) = O \times X$ and the above diagram gives rise to a coinductive definition principle. A similar principle can be obtained for infinite binary trees and other algebraic datatypes. The above diagrams are in the category of sets and functions, but infinite streams and trees have a very rich algebraic structure and they are also the carrier of final coalgebras in other categories. For the purpose of this paper, we will be particularly interested in a category where the maps are probabilistic – the Kleisli category of the distribution (or *Giry*) monad.

Probability Distributions, σ-algebras, Measurable Spaces

To model probabilistic behavior, we need some basic concepts from measure theory (see, e.g., [30]). Given an arbitrary set X we call a set Σ of subsets of X a *σ-algebra* if it contains the empty set and is closed under complement and countable union. A *measurable space* is a pair (X, Σ). A *probability measure* or distribution μ over a measurable space is a function $\mu : \Sigma \to [0, 1]$ assigning probabilities $\mu(A) \in [0, 1]$ to the *measurable sets* $A \in \Sigma$ such that $\mu(X) = 1$ and $\mu(\bigcup_{i\in I} A_i) = \sum_{i\in I} \mu(A_i)$ whenever $\{A_i\}_{i\in I}$ is a countable collection of disjoint measurable sets. The collection $\mathcal{D}(X)$ of probability distributions over a measurable space

X forms the so-called Giry monad. The monad unit $\eta\colon X \to \mathcal{D}(X)$ maps $a \in X$ to the point mass (or Dirac measure) δ_a, i.e., the measure assigning 1 to any set containing a and 0 to any set not containing a. The monad multiplication $m\colon \mathcal{D}\mathcal{D}(X) \to \mathcal{D}(X)$ is given by integration:

$$m(P)(S) = \int ev_S dP, \text{ where } ev_S(\mu) = \mu(S).$$

Given measurable spaces (X, Σ_X) and (Y, Σ_Y), a *Markov kernel* is a function $P : X \times \Sigma_Y \to [0,1]$ (equivalently, $X \to \Sigma_Y \to [0,1]$) that maps each source state $x \in X$ to a distribution over target states $P(x,-) : \Sigma_Y \to [0,1]$.

Markov kernels form the arrows in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$ of the $\mathcal{D}$ monad; we denote such arrows by $X \xrightarrow{P} Y$. Composition in the Kleisli category is given by integration:

$$X \xrightarrow{P} Y \xrightarrow{Q} Z \qquad (P \circ Q)(x, A) = \int_{y \in Y} P(x, dy) \cdot Q(y, A)$$

Associativity of composition is essentially Fubini's theorem.

3 Defining Almost Sure Productivity

We will consider programs that denote probability distributions over coinductive types, such as infinite streams or trees. In this section, we focus on the definitions for programs producing streams and binary trees for simplicity, but our results should extend to arbitrary polynomial functors (see Section 7).

First, we introduce the semantics of programs. Rather than fix a concrete programming language at this point, we let $\mathbb{T}$ denote an abstract state space (e.g., the terms of a programming language or the space of program memories). The state evolves over an infinite sequence of discrete time steps. At each step, we will probabilistic observe either a concrete output ($a \in A$) or nothing ($\bot$), along with a resulting state. Intuitively, $p \in \mathbb{T}$ is ASP if its probability of producing unboundedly many outputs is 1. Formally, we give states in $\mathbb{T}$ a denotational semantics $[\![-]\!]\colon \mathbb{T} \to \mathcal{D}((A_\bot)^\omega)$ defined coinductively, starting from a given one-step semantics function that maps each term to an output in $A_\bot$ and the resulting term. Since the step function is probabilistic, we work in the Kleisli category for the distribution monad; this introduces some complications when computing the final coalgebras in this category. We take the work on probabilistic streams by Kerstan and König [20] as our starting point, and then generalize to probabilistic trees.

▶ **Theorem 1** (Finality for streams [20]). *Given a set $\mathbb{T}$ of programs endowed with a probabilistic step function $\mathsf{st}\colon \mathbb{T} \to \mathcal{D}(A_\bot \times \mathbb{T})$, there is a* unique *semantics function $[\![-]\!]$ assigning to each program a probability distribution of output streams such that the following diagram commutes in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$:*

$$\begin{array}{ccc} \mathbb{T} & \xrightarrow{[\![-]\!]} & (A_\bot)^\omega \\ {\scriptstyle \mathsf{st}}\downarrow & & \downarrow{\scriptstyle <\mathsf{head},\mathsf{tail}>} \\ A_\bot \times \mathbb{T} & \xrightarrow{id \times [\![-]\!]} & A_\bot \times (A_\bot)^\omega \end{array}$$

▶ **Definition 2** (ASP for streams). A stream program $p \in \mathbb{T}$ is *almost surely productive* (ASP) if

$$\Pr_{\sigma \sim [\![p]\!]}[\sigma \text{ has infinitely many concrete output elements } a \in A] = 1.$$

For this to be a sensible definition, the event "σ has infinitely many concrete output elements $a \in A$" must be a measurable set in some σ-algebra on $(A_\perp)^\omega$. Following Kerstan and König, we take the σ-algebra generated by *cones*, sets of the form $uA^\omega = \{v \in (A_\perp)^\omega \mid u \text{ prefix of } v, u \in (A_\perp)^*\}$. Our definition evidently depends on the definition of $[\![-]\!] \colon \mathbb{T} \to \mathcal{D}(A_\perp)^\omega$; our coinductively defined semantics will be useful later for showing soundness when verifying ASP, but our definition of ASP is sensible for any semantics $[\![-]\!]$.

▶ **Example 3.** Let us consider the following program defining a stream σ recursively, in which each recursion step is determined by a coin flip with bias p:

$$\sigma = (a : \sigma) \oplus_p \mathsf{tail}(\sigma)$$

In the next section we will formally introduce this programming language, but intuitively the program repeatedly flips a coin. If the coin flip results in heads the program produces an element a. Otherwise the program tries to compute the tail of the recursive call; the first element produced by the recursive call is dropped (consumed), while subsequent elements produced (if any) are emitted as output.

To analyze the productivity behavior of this probabilistic program, we can reason intuitively. Each time the second branch is chosen, the program must choose the first branch strictly more than once in order to produce one output (since, e.g., $\mathsf{tail}(a : \sigma) = \sigma$). Accordingly, the productivity behavior of this program depends on the value of p. When p is less than $1/2$, the program chooses the first branch less often than the second branch and the program is not ASP. On the other hand, when $p > 1/2$ the program will tend to produce more elements a than are consumed by the destructors, and the above program is ASP. In the sequel, we will show two methods to formally prove this fact.

It will be convenient to represent the functor as $F(X) = A_\perp \times X$ as $A \times X + X$. In the rest of this paper we will often use the latter representation and refer to the final coalgebra as *observation streams* $\mathsf{OS} = (A_\perp)^\omega$ with structure $\mathsf{OS} \xleftarrow[\cong]{\langle \mathsf{out},\mathsf{unf}\rangle} A \times \mathsf{OS} + \mathsf{OS}$ given by $\mathsf{out}(a, \sigma) = a : \sigma$ and $\mathsf{unf}(\sigma) = \perp : \sigma$.

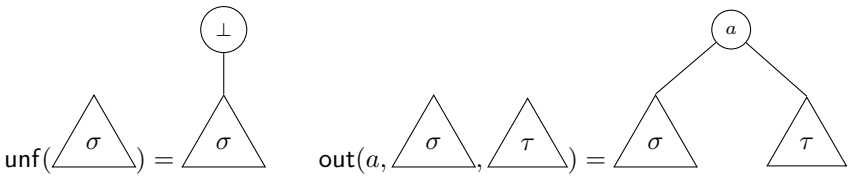
Streams are not the only coinductively defined data; infinite binary trees are another classical example. To generate trees, we can imagine that a program produces an output value – labeling the root node – and two child programs, which then generate the left and right child of a tree of outputs. Much like we saw for streams, probabilistic programs generating these trees may sometimes step to a single new program without producing outputs. Accordingly we will work with the functor $F(X) = A \times X \times X + X$, where the left summand can be thought of as the result of an output step, while the right summand gives the result of a non-output step.

▶ **Theorem 4** (Finality for trees). *Given a set of programs $\mathbb{T}$ endowed with a probabilistic step function $\mathsf{st} \colon \mathbb{T} \to \mathcal{D}(A \times \mathbb{T} \times \mathbb{T} + \mathbb{T})$, there is a* unique *semantics function $[\![-]\!]$ assigning to each program a probability distribution of output trees such that the following diagram commutes in the Kleisli category $\mathcal{K}\ell(\mathcal{D})$.*

$$\begin{array}{ccc} \mathbb{T} & \xrightarrow{\quad [\![-]\!] \quad} & \mathsf{Trees}(A_\perp) \\ {\scriptstyle \mathsf{st}}\downarrow & & \downarrow{\scriptstyle \langle \mathsf{out},\mathsf{unf}\rangle^{-1}} \\ A \times \mathbb{T} \times \mathbb{T} + \mathbb{T} & \xrightarrow{id \times [\![-]\!] \times [\![-]\!] + [\![-]\!]} & A \times \mathsf{Trees}(A_\perp) \times \mathsf{Trees}(A_\perp) + \mathsf{Trees}(A_\perp) \end{array}$$

$\mathsf{Trees}(A_\perp)$ are infinite trees where the nodes are either elements of A or $\perp$. An a-node has two children whereas a $\perp$-node only has one child. Formally, we can construct these trees

with the two maps out and unf:



Defining ASP for trees is a bit more subtle than for streams. Due to measurability issues, we can only refer to the probability of infinitely many outputs along one path at a time in the tree. A bit more formally, let $w \in \{L, R\}^\omega$ be an infinite word on alphabet $\{L, R\}$. Given any tree $t \in \mathsf{Trees}(A_\perp)$, w induces a single path t_w in the tree: from the root, the path follows the left/right child of a-nodes as indicated by w, and the single child of $\perp$-nodes.

▶ **Definition 5** (ASP for trees)**.** A tree program $p \in \mathbb{T}$ is *almost surely productive* (ASP) if

$$\forall w \in \{L, R\}^\omega.\ \Pr_{t \sim [\![p]\!]}[t_w \text{ has infinitely many concrete output nodes } a \in A] = 1.$$

We have omitted the σ-algebra structure on $\mathsf{Trees}(A_\perp)$ for lack of space, but it is quite similar to the one for streams: it is generated by the cones $u\mathsf{Trees}(A_\perp) = \{t \in \mathsf{Trees}(A_\perp) \mid t \text{ is an extension of the finite tree } u\}$.

▶ **Example 6.** Consider the probabilistic tree defined by the following equation:

$$\tau = \mathsf{mk}(a, \tau, \tau) \oplus_p \mathsf{left}(\tau)$$

The $\mathsf{mk}(a, t_1, t_2)$ constructor produces a tree with the root labeled by a and children t_1 and t_2, while the $\mathsf{left}(t)$ destructor consumes the output at the root of t and steps to the left child of t. While this example is more difficult to work out informally, it has similar ASP behavior as the previous example we saw for streams: when $p > 1/2$ this program is ASP, since it has strictly higher probability of constructing a node (and producing an output) than destructing a node (and consuming an output).

4 A Calculus for Probabilistic Streams and Trees

Now that we have introduced almost sure productivity, we consider how to verify this property. We work with two variants of a simple calculus for probabilistic coinductive programming, for producing streams and trees respectively. We suppose that outputs are drawn from some finite alphabet A. The language for streams considers terms of the following form:

$$e \in \mathbb{T} ::= \sigma \mid e \oplus_p e \mid a : e\ (a \in A) \mid \mathsf{tail}(e)$$

The distinguished variable σ represents a recursive occurrence of the stream so that streams can be defined via equations $\sigma = e$. The operation $e_1 \oplus_p e_2$ selects e_1 with probability p and e_2 with probability $1 - p$. The constructor $a : e$ builds a stream with head a and tail e. The destructor $\mathsf{tail}(e)$ computes the tail of a stream, discarding the head.

The language for trees is similar, with terms of the following form:

$$e \in \mathbb{T} ::= \tau \mid e \oplus_p e \mid \mathsf{mk}(a, e, e)\ (a \in A) \mid \mathsf{left}(e) \mid \mathsf{right}(e)$$

The variable τ represents a recursive occurrence of the tree, so that trees are defined as $\tau = e$. The constructor $\mathsf{mk}(a, e_1, e_2)$ builds a tree with root labeled a and children e_1 and e_2. The destructors $\mathsf{left}(e)$ and $\mathsf{right}(e)$ extract the left and right children of e, respectively.

We interpret these terms coalgebraically by first giving a step function from $\mathsf{st}_e : \mathbb{T} \to \mathcal{D}(F(\mathbb{T}))$ for an appropriate functor, and then taking the semantics as the map to the final coalgebra. For streams, we take the functor $F(X) = A \times X + X$: a term steps to a distribution over either an output in A and a resulting term, or just a resulting term (with no output). To describe how the recursive occurrence σ steps, we parametrize the step function st_e by the top level stream term e; this term remains fixed throughout the evaluation. This choice restricts recursion to be global in nature, i.e., our language does not support mutual or nested recursion. Supporting more advanced recursion is also possible, but we stick with the simpler setting here; we return to this point in Section 7.

The step relation is defined by case analysis on the syntax of terms. Probabilistic choice terms reduce by scaling the result of stepping e and the result of stepping e' by p and $1-p$ respectively, and then combining the distributions:

$$\mathsf{st}_e(e_1 \oplus_p e_2) \triangleq p \cdot \mathsf{st}_e(e_1) + (1-p) \cdot \mathsf{st}_e(e_2)$$

The next cases push destructors into terms:

$$\mathsf{st}_e(\mathsf{tail}^k(a : e)) \triangleq \mathsf{st}_e(\mathsf{tail}^{k-1}(e))$$
$$\mathsf{st}_e(\mathsf{tail}^k(e_1 \oplus_p e_2)) \triangleq \mathsf{st}_e(\mathsf{tail}^k(e_1) \oplus_p \mathsf{tail}^k(e_2))$$

Here and below, we write tail^k as a shorthand for $k > 0$ applications of tail.

The remaining cases return point distributions. If we have reached a constructor then we produce a single output. Otherwise, we replace σ by the top level stream term, unfolding a recursive occurrence.

$$\mathsf{st}_e(a : e') \triangleq \delta(inl(a, e'))$$
$$\mathsf{st}_e(e') \triangleq \delta(inr(e'[e/\sigma])) \quad \text{otherwise}$$

Note that a single evaluation step of a stream may lead to multiple constructors at top level of the term, but only one output can be recorded each step – the remaining constructors are preserved in the term and will give rise to outputs in subsequent steps.

The semantics is similar for trees. We take the functor $F(X) = (A \times X \times X) + X$: a term reduces to a distribution over either an output in A and two child terms, or a resulting term and no output. The main changes to the step relation are for constructors and destructors. The constructor $\mathsf{mk}(a, e_1, e_2)$ reduces to $\delta(inl(a, e_1, e_2))$, representing an output a this step. Destructors are handle like tail for streams, where $\mathsf{left}(\mathsf{mk}(a, e_1, e_2))$ reduces to e_1 and $\mathsf{right}(\mathsf{mk}(a, e_1, e_2))$ reduces to e_2, and $\mathsf{tail}^k(-)$ is generalized to any finite combination of $\mathsf{left}(-)$ and $\mathsf{right}(-)$.

Concretely, let $C[e]$ be any (possibly empty) combination of left and right applied to e. We have the following step rules:

$$\mathsf{st}_e(C[\mathsf{left}(\mathsf{mk}(a, e_l, e_r))]) \triangleq \mathsf{st}_e(C[e_l])$$
$$\mathsf{st}_e(C[\mathsf{right}(\mathsf{mk}(a, e_l, e_r))]) \triangleq \mathsf{st}_e(C[e_r])$$
$$\mathsf{st}_e(C[e_1 \oplus_p e_2]) \triangleq p \cdot \mathsf{st}_e(C[e_1]) + (1-p) \cdot \mathsf{st}_e(C[e_2])$$
$$\mathsf{st}_e(\mathsf{mk}(a, e_l, e_r)) \triangleq \delta(inl(a, e_l, e_r))$$
$$\mathsf{st}_e(C[\tau]) \triangleq \delta(inr(C[e]))$$

5 Syntactic Conditions for ASP

With the language and semantics in hand, we now turn to proving ASP. While it is theoretically possible to reason directly on the semantics using our definitions from Section 3, in practice it

is much easier to reason about the language. In this section we present a syntactic sufficient condition for ASP. Intuitively, the idea is to approximate the expected number of outputs every step; if this measure is strictly positive, then the program is ASP.

5.1 A Syntactic Measure

We define a syntactic measure $\#(-) : \mathbb{T} \to \mathbb{R}$ by induction on stream terms:

$$\begin{aligned} \#(\sigma) &\triangleq 0 \\ \#(e_1 \oplus_p e_2) &\triangleq p \cdot \#(e_1) + (1-p) \cdot \#(e_2) \\ \#(a : e) &\triangleq \#(e) + 1 \\ \#(\mathsf{tail}(e)) &\triangleq \#(e) - 1 \end{aligned}$$

The measure # describes the expected difference between the number of outputs produced (by constructors) and the number of outputs consumed (by destructors) in each unfolding of the term. We can define a similar measure for tree terms:

$$\begin{aligned} \#(\tau) &\triangleq 0 \\ \#(e_1 \oplus_p e_2) &\triangleq p \cdot \#(e_1) + (1-p) \cdot \#(e_2) \\ \#(\mathsf{mk}(a, e_1, e_2)) &\triangleq \min(\#(e_1), \#(e_2)) + 1 \\ \#(\mathsf{left}(e)) = \#(\mathsf{right}(e)) &\triangleq \#(e) - 1 \end{aligned}$$

We can now state conditions for ASP for streams and trees.

▶ **Theorem 7.** *Let e be a stream term with $\gamma = \#(e)$. If $\gamma > 0$, e is ASP.*

▶ **Theorem 8.** *Let e be a tree term with $\gamma = \#(e)$. If $\gamma > 0$, e is ASP.*

The main idea behind the proof for streams is that by construction of the step relation, each step either produces an output or unfolds a fixed point (if there is no output). In unfolding steps, the expected measure of the term plus the number of outputs increases by γ. By defining an appropriate martingale and applying the Azuma-Hoeffding inequality, the sum of the measure and the number of outputs must increase linearly as the term steps when $\gamma > 0$. Since the measure is bounded above – when the measure is large the stream outputs instead of unfolding – the number of outputs must increase linearly and the stream is ASP.

The proof for trees is similar, showing that on any path through the observation tree there are infinitely many output steps with probability 1. We present detailed proofs in the full version of this paper.

5.2 Examples

We consider a few examples of our analysis. The alphabet A does not affect the ASP property; without loss of generality, we can let the alphabet A be the singleton $\{\star\}$.

▶ **Example 9.** Consider the stream definition $\sigma = (\star : \sigma) \oplus_p \mathsf{tail}(\sigma)$. The # measure of the stream term is $p \cdot 1 + (1-p) \cdot (-1) = 2p - 1$. By Theorem 7, the stream is ASP when $p > 1/2$.

The measure does not give useful information when # is not positive.

▶ **Example 10.** Consider the stream definition $\sigma = (\star : \sigma) \oplus_{1/2} \mathsf{tail}(\sigma)$; the # measure of the term is 0. The number of outputs can be modeled by a simple random walk on a line, where

the maximum position is the number of outputs produced by the stream. Since a simple random walk has probability 1 of reaching every $n \in \mathbb{N}$ [26], the stream term is ASP.

In contrast, $\#(\sigma)$ is 0 but the stream definition $\sigma = \sigma$ is clearly non-productive.

We can give similar examples for tree terms.

▶ **Example 11.** Consider the tree definitions $\tau = e_i$, where

- $e_1 \triangleq \mathsf{left}(\tau) \oplus_{1/4} \mathsf{mk}(\star, \tau, \tau)$
- $e_2 \triangleq \mathsf{left}(\tau) \oplus_{1/4} \mathsf{mk}(\star, \tau, \mathsf{left}(\tau))$.

We apply Theorem 8 to deduce ASP. We have $\#(e_1) = (1/4) \cdot (-1) + (3/4) \cdot (+1) = 1/2$, so the first term is ASP. For the second term, $\#(e_2) = (1/4) \cdot (-1) + (3/4) \cdot 0 = -1/4$, so our analysis does not give any information.

6 Probabilistic Model-Checking for ASP

The syntactic analysis for ASP is simple, but it is not complete – no information is given if the measure is not positive. In this section we give a more sophisticated, complete analysis by first modeling the operational semantics of a term by a *Probabilistic Pushdown Automaton* (pPDA), then deciding ASP by reduction to model-checking.

6.1 Probabilistic Pushdown Automata and LTL

A pPDA is a tuple $\mathcal{A} = (S, \Gamma, \mathcal{T})$ where S is a finite set of states and Γ is a finite *stack alphabet*. The *transition function* $\mathcal{T} : S \times (\Gamma \cup \{\bot\}) \times S \times \Gamma^* \to [0, 1]$ looks at the top symbol on the stack (which might be empty, denoted $\bot$), consumes it, and pushes a (possibly empty, denoted ε) string of symbols onto the stack, before transitioning to the next state. A *configuration* of $\mathcal{A}$ is an element of $\mathcal{C} = S \times \Gamma^*$, and represents the state of the pPDA and the contents of its stack (with the top on the left) at some point of its execution. Given a configuration, the transition function $\mathcal{T}$ specifies a distribution over configurations in the next step. Given an initial state s and an initial stack $\gamma \in \Gamma^*$, $\mathcal{T}$ induces a distribution $\mathrm{Paths}(s, \gamma)$ over the infinite sequence of configurations starting in (s, γ).

Linear Temporal Logic (LTL) [29] is a linear-time temporal logic that describes runs of a transition system, which in a pPDA correspond to infinite sequences of configurations. Propositions in LTL are defined by the syntax

$$\phi, \psi ::= Q \mid \neg\phi \mid \mathcal{X}\phi \mid \phi \,\mathcal{U}\, \psi \mid \Diamond\phi \mid \Box\phi$$

where ϕ, ψ are *path formulas*, which describe a particular path, and Q is a set of atomic propositions. The validity of an LTL formula on a run π of pPDA $\mathcal{A}$ is defined as follows:

$$\pi \models \Phi \Leftrightarrow \pi[0] \in Q \qquad \pi \models \phi \,\mathcal{U}\, \psi \Leftrightarrow \exists i. \pi_i \models \psi \wedge \forall j < i. \pi_j \models \phi$$
$$\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi \qquad \pi \models \Diamond\phi \Leftrightarrow \exists i. \pi_i \models \phi$$
$$\pi \models \mathcal{X}\phi \Leftrightarrow \pi_1 \models \phi \qquad \pi \models \Box\phi \Leftrightarrow \forall i. \pi_i \models \phi$$

where atomic propositions q are interpreted as $[\![q]\!] \subseteq \mathcal{C}$. As expected, path formulas are interpreted in traces of configurations $\pi \in \mathcal{C}^\omega$; $\pi[i]$ is the ith element in the path π, and π_i is the suffix of π from $\pi[i]$.

Given a pPDA $\mathcal{A}$, a starting configuration $(s, \gamma) \in \mathcal{C}$ and a LTL formula ϕ, the *qualitative model-checking problem* is to decide whether runs starting from (s, γ) satisfy ϕ almost surely, i.e., whether $\Pr_{\pi \in \mathrm{Paths}(s,\gamma)}[\pi \models \phi] = 1$. The following is known.

▶ **Theorem 12** (Brázdil, et al. [6]). *The quantitative model-checking problem for pPDAs against LTL specifications is decidable.*

Almost sure productivity states that an event – namely, producing an output – occurs infinitely often with probability 1. Such properties can be expressed in LTL.

▶ **Lemma 13.** *Let $(s,\gamma) \in \mathcal{C}$ be an initial configuration and $\mathcal{B} \subseteq \mathcal{C}$ be a set of configurations. Then $\Pr_{\pi \in \mathrm{Paths}(s,\gamma)}[\pi$ visits $\mathcal{B}$ infinitely often$] = 1$ iff $\Pr_{\pi \in \mathrm{Paths}(s,\gamma)}[\pi \models \Box\Diamond\mathcal{B}] = 1$.*

We will encode language terms as pPDAs and cast almost sure productivity as an LTL property stating that configurations representing output steps are reached infinitely often with probability 1. Theorem 12 then gives a decision procedure for ASP. In general, this algorithm[1] is in **PSPACE**.

6.2 Modeling streams with pPDAs

The idea behind our encoding from terms to pPDAs is simple to describe. The states of the pPDA will represent subterms of the original term, and transitions will model steps. In the original step relation, the only way a subterm can step to a non-subterm is by accumulating destructors. We use a single-letter stack alphabet to track the number of destructors so that a term like $\mathsf{tail}^k(e)$ can be modeled by the state corresponding to e and k counters on the stack. More formally, given a stream term e we define a pPDA $\mathcal{A}_e = (\mathcal{S}_e, \{tl\}, \mathcal{T}_e)$, where $\mathcal{S}_e$ is the set of syntactic subterms of e and $\mathcal{T}_e$ is the following transition function:

$$\begin{aligned}
\mathcal{T}_e((\sigma,a),(e,a)) &= 1 & \mathcal{T}_e((a':e',\bot),(e',\varepsilon)) &= 1\\
\mathcal{T}_e((e_1 \oplus_p e_2,a),(e_1,a)) &= p & \mathcal{T}_e((a':e',tl),(e',\varepsilon)) &= 1\\
\mathcal{T}_e((e_1 \oplus_p e_2,a),(e_2,a)) &= 1-p & \mathcal{T}_e((\mathsf{tail}(e'),a),(e',tl\cdot a) &= 1
\end{aligned}$$

Above, $\cdot$ concatenates strings and we implicitly treat a as alphabet symbol or a singleton string. All non-specified transitions have zero probability. We define the set of *outputting configurations* as $\mathcal{O} \triangleq \{s \in \mathcal{C} \mid \exists a', e'.\, s = (a' : e', \bot)\}$, that is, configurations where the current term is a constructor and there are no pending destructors. Our main result states that this set is visited infinitely often with probability 1 if and only if e is ASP. In fact, we prove something stronger:

▶ **Theorem 14.** *Let e be a stream term and let $\mathcal{A}_e$ be the corresponding pPDA. Then,*

$$\Pr_{t\sim[\![e]\!]}[t \text{ has infinitely many output nodes}] = \Pr_{\pi\sim\mathrm{Paths}(e,\varepsilon)}[\pi \models \Box\Diamond\,\mathcal{O}].$$

In particular, e is ASP if and only if for almost all runs π starting in (e,ε), $\pi \models \Box\Diamond\mathcal{O}$.

By Theorem 12, ASP is decidable for stream terms. In fact, it is also possible to decide whether a stream term is almost surely *not* productive, i.e., the probability of producing infinitely many outputs is zero.

[1] Technically, this algorithm requires first encoding the LTL formula into a Deterministic Rabin Automaton (DRA). Even though this encoding can in general blow up the problem size exponentially, this is not the case for the simple conditions we consider.

6.3 Extending to trees

Now, we extend our approach to trees. The main difficulty can be seen in the constructors. For streams, we can encode the term $a : e$ by proceeding to the tail e. For trees, however, how can we encode $\mathsf{mk}(a, e_1, e_2)$? The pPDA cannot step to both e_1 and e_2. Since the failure of ASP may occur down either path, we cannot directly translate the ASP property on trees to LTL – ASP is a property of *all* paths down the tree. Instead, on constructors our pPDA encoding will choose a path at random to simulate. As we will show, if the probability of choosing a path that outputs infinitely often is 1, then every path will output infinitely often. Notice that in general, properties that happen with probability 1 do not necessarily happen for every path, but the structure of our problem allows us to make this generalization.

More formally, the stack alphabet will now be $\{rt, lt\}$, and on constructors we transition to each child with probability $1/2$:

$$\begin{aligned}
\mathcal{T}_e((\tau, a), (e, a)) &= 1 \\
\mathcal{T}_e((e_1 \oplus_p e_2, a), (e_1, a)) &= p \\
\mathcal{T}_e((e_1 \oplus_p e_2, a), (e_2, a)) &= 1 - p \\
\mathcal{T}_e((\mathsf{mk}(a', e_l, e_r), \bot), (e_l, \varepsilon)) &= 1/2 \\
\mathcal{T}_e((\mathsf{mk}(a', e_l, e_r), \bot), (e_r, \varepsilon)) &= 1/2
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}_e((\mathsf{mk}(a', e_l, e_r), lt), (e_l, \varepsilon)) &= 1 \\
\mathcal{T}_e((\mathsf{mk}(a', e_l, e_r), rt), (e_r, \varepsilon)) &= 1 \\
\mathcal{T}_e((\mathsf{left}(e'), a), (e', lt \cdot a)) &= 1 \\
\mathcal{T}_e((\mathsf{right}(e'), a), (e', rt \cdot a)) &= 1
\end{aligned}$$

We define $\mathcal{O} \triangleq \{s \in \mathcal{C} \mid \exists a', e_l, e_r.\, s = (\mathsf{mk}(a', e_l, e_r), \varepsilon)\}$ to be the set of outputting configurations, and we can characterize ASP with the following theorem.

▶ **Theorem 15.** *Let e be a tree term and $\mathcal{A}_e$ be the corresponding probabilistic PDA. Then $\Pr_{\pi \sim \mathrm{Paths}(e,\bot)}[\pi \models \Box\Diamond\, \mathcal{O}] = 1$ if and only if for every $w \in \{L, R\}^\omega$,*

$$\Pr_{t \sim [\![e]\!]}[t \text{ has infinitely many output nodes along } w] = 1.$$

Thus we can decide ASP by deciding a LTL formula.

7 Possible Generalizations and Extensions

Our definition of ASP and our verification approaches suggest several natural directions.

Handling Richer Languages. The most concrete direction is to consider richer languages for coinductive probabilistic programming. Starting from our core language, one might consider allowing more operations on coinductive terms, mutually recursive definitions, or conditional tests of some kind. It should also be possible to develop languages for more complex coinductive types associated with general polynomial functors (see, e.g., Kozen [24]). Note that adding more operations, e.g. pointwise + of streams would increase the expressivity of the language but raise additional challenges from the perspective of the semantics – we would have to add extra structure to the base category and re-check that the finality proof.

Developing new languages for coinductive probabilistic programming – perhaps an imperative language or a higher-order language – would also be interesting. From the semantics side, our development in Section 3 should support any language equipped with a small-step semantics producing output values, allowing ASP to be defined for many kinds of languages. The verification side appears more challenging. Natural extensions, like a pointwise addition operation, already seem to pose challenges for the analyses. We know of no general method to reasoning about ASP. This stands in contrast to almost sure termination, which can be established by where flexible criteria like decreasing probabilistic variants [17]. Considering counterparts of these methods for ASP is an interesting avenue of research.

Exploring Other Definitions. Our definition of ASP is natural, but other definitions are possible. For trees (and possibly more complex coinductive structures), we could instead require that there *exists* a path producing infinitely many outputs, rather than requiring that *all* paths produce infinitely many outputs. This weaker notion of ASP can be defined in our semantics, but it is currently unclear how to verify this kind of ASP.

Our notion of ASP also describes just the probability of generating infinitely many outputs, and does not impose any requirement on the generation rate. Quantitative strengthenings of ASP – say, requiring bounds on the expected number of steps between outputs – could give more useful information.

Understanding Dependence on Step Relation. Our coalgebraic semantics supporting our verification methods are based on a small-step semantics for programs. A natural question is whether this dependence is necessary, or if one could verify ASP with a less step-dependent semantics. Again drawing an analogy, it appears that fixing a reduction strategy is important in order to give a well-defined notion of almost sure termination for probabilistic higher-order languages (see, e.g., [25]). The situation for almost sure productivity is less clear.

8 Related Work

Our work is inspired by two previously independent lines of research: probabilistic termination and productivity of coalgebraic definitions.

Probabilistic Termination. There are a broad range of techniques for proving termination of probabilistic programs. Many of the most powerful criteria use advanced tools from probability theory [27], especially martingale theory [8, 16, 9, 10, 11]. Other works adopt more pragmatic approaches, generally with the goal of achieving automation. Arons, Pnueli and Zuck [4] reduce almost sure termination of a program P to termination of a non-deterministic program Q, using a planner that must be produced by the verifier. Esparza, Gaiser and Kiefer [15] give a CEGAR-like approach for building patterns (which play a role similar to planners) and prove that their approach is complete for a natural class of programs.

Productivity of Corecursive Definitions. There has been a significant amount of work on verifying productivity of corecursive definitions without probabilistic choice. Endrullis and collaborators [14] give a procedure for deciding productivity of an expressive class of stream definitions. In a companion work [13], they study the strength of data oblivious criteria, i.e., criteria that do not depend on values. More recently, Komendantskaya and collaborators [21] introduce observational productivity and give a semi-decision procedure for logic programs.

9 Conclusion

We introduce almost sure productivity, a counterpart to almost sure termination for probabilistic coinductive programs. In addition, we propose two methods for proving ASP for a core language for streams and infinite trees. Our results demonstrate that verification of ASP is feasible and can even be decidable for simple languages.

References

1. Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 27–38, 2013. doi:10.1145/2429069.2429075.
2. Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 2(POPL):34:1–34:32, 2018. doi:10.1145/3158122.
3. Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. Relational reasoning for Markov chains in a probabilistic guarded lambda calculus. In *European Symposium on Programming (ESOP), Thessaloniki, Greece*, Lecture Notes in Computer Science. Springer-Verlag, 2018.
4. Tamarah Arons, Amir Pnueli, and Lenore D. Zuck. Parameterized verification by probabilistic abstraction. In Andrew D. Gordon, editor, *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS), Warsaw, Poland*, volume 2620 of *Lecture Notes in Computer Science*, pages 87–102. Springer-Verlag, 2003. doi:10.1007/3-540-36576-1_6.
5. S. Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006. URL: https://books.google.es/books?id=IK_sIDI2TCwC.
6. Tomáš Brázdil, Javier Esparza, Stefan Kiefer, and Antonín Kučera. Analyzing probabilistic pushdown automata. *Formal Methods in System Design*, 43(2):124–163, Oct 2013. doi:10.1007/s10703-012-0166-0.
7. Venanzio Capretta and Jonathan Fowler. The continuity of monadic stream functions. In *IEEE Symposium on Logic in Computer Science (LICS), Reykjavik, Iceland*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005119.
8. Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia*, pages 511–526, 2013. URL: https://www.cs.colorado.edu/~srirams/papers/cav2013-martingales.pdf.
9. Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination analysis of probabilistic programs through Positivstellensatz's. In *International Conference on Computer Aided Verification (CAV), Toronto, Ontario*, volume 9779 of *Lecture Notes in Computer Science*, pages 3–22. Springer-Verlag, 2016. doi:10.1007/978-3-319-41528-4_1.
10. Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Saint Petersburg, Florida*, pages 327–342, 2016. doi:10.1145/2837614.2837639.
11. Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. Stochastic invariants for probabilistic termination. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 145–160, 2017. doi:10.1145/3009837.3009873.
12. Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, 12(3), 2016. doi:10.2168/LMCS-12(3:7)2016.
13. Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Data-oblivious stream productivity. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Doha, Qatar*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2008. doi:10.1007/978-3-540-89439-1_6.

14 Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theor. Comput. Sci.*, 411(4-5):765–782, 2010. doi:10.1016/j.tcs.2009.10.014.

15 Javier Esparza, Andreas Gaiser, and Stefan Kiefer. Proving termination of probabilistic programs using patterns. In P. Madhusudan and Sanjit A. Seshia, editors, *International Conference on Computer Aided Verification (CAV), Berkeley, California*, volume 7358 of *Lecture Notes in Computer Science*, pages 123–138. Springer-Verlag, 2012. doi:10.1007/978-3-642-31424-7_14.

16 Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In Sriram K. Rajamani and David Walker, editors, *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 489–501, 2015. doi:10.1145/2676726.2677001.

17 Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, 1983. doi:10.1145/2166.357214.

18 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 410–423, 1996. doi:10.1145/237721.240882.

19 Benjamin Lucien Kaminski and Joost-Pieter Katoen. On the hardness of almost-sure termination. In Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella, editors, *Symposium on Mathematical Foundations of Computer Science (MFCS), Milan, Italy*, volume 9234 of *Lecture Notes in Computer Science*, pages 307–318. Springer-Verlag, 2015. doi:10.1007/978-3-662-48057-1_24.

20 Henning Kerstan and Barbara König. Coalgebraic trace semantics for continuous probabilistic transition systems. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:16)2013.

21 Ekaterina Komendantskaya, Patricia Johann, and Martin Schmidt. A productivity checker for logic programming. In Manuel V. Hermenegildo and Pedro López-García, editors, *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), Edinburgh, Scotland*, volume 10184 of *Lecture Notes in Computer Science*, pages 168–186. Springer-Verlag, 2016. doi:10.1007/978-3-319-63139-4_10.

22 Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981. doi:10.1016/0022-0000(81)90036-2.

23 Dexter Kozen. A probabilistic PDL. In *ACM SIGACT Symposium on Theory of Computing (STOC), Boston, Massachusetts*, pages 291–297, 1983. doi:10.1145/800061.808758.

24 Dexter Kozen. Realization of coinductive types. *Electronic Notes in Theoretical Computer Science*, 276:237–246, 2011. doi:10.1016/j.entcs.2011.09.024.

25 Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. In Hongseok Yang, editor, *European Symposium on Programming (ESOP), Uppsala, Sweden*, volume 10201 of *Lecture Notes in Computer Science*, pages 393–419. Springer-Verlag, 2017. doi:10.1007/978-3-662-54434-1_15.

26 David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2009. URL: https://pages.uoregon.edu/dlevin/MARKOV/markovmixing.pdf.

27 Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages*, 2(POPL):33:1–33:28, 2018. doi:10.1145/3158121.

28 Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *toplas*, 18(3):325–353, 1996. doi:10.1145/229542.229547.

29 Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science (FOCS), Providence, Rhode Island*, pages 46–57, 1977. doi:10.1109/SFCS.1977.32.

30 Walter Rudin. *Real and Complex Analysis*. McGraw-Hill, New York, third edition, 1987.