Almost Sure Productivity

Alexandra Silva

joint work with Alejandro Aguirre, Gilles Barthe, and Justin Hsu

arXiV:1802.06283

Productivity

every finite part of the infinite structure can be evaluated in finite time

$$
s\cdot\mathbb{R}^\omega
$$

$$
s=1:s \qquad \qquad 1:1:1:\cdots
$$

$$
s = 1 : (ones + s)
$$

\n $ones = 1 : ones$
\n $1 : 2 : 3 : 4 : ...$

$$
s = 0 : zip(inv(even(s)), tail(s))
$$

\n
$$
inv(0 : s) = 1 : inv(s) \quad inv(1 : s) = 0 : inv(s)
$$

\n
$$
even(x : s) = x : odd(s) \quad odd(x : s) = even(s)
$$

\n
$$
zip(1 : s, t) = 1 : zip(t, s)
$$

$$
s = 0 : zip(inv(even(s)), tail(s))
$$

$$
inv(0:s) = 1 : inv(s) \quad inv(1:s) = 0 : inv(s)
$$

$$
even(x:s) = x : odd(s) \quad odd(x:s) = even(s)
$$

$$
zip(1:s,t) = 1 : zip(t,s)
$$

 $0:1:1:0:1:0:0:1: \cdots$ *Thue - Morse Sequence*

$$
s = 0 : zip(inv(even(s)), tail(s))
$$

$$
inv(0:s) = 1 : inv(s) \quad inv(1:s) = 0 : inv(s)
$$

$$
even(x:s) = x : odd(s) \quad odd(x:s) = even(s)
$$

$$
zip(1:s,t) = 1 : zip(t,s)
$$

 $0:1:1:0:1:0:0:1: \cdots$ *Thue - Morse Sequence*

 $\{0 \to 01, 1 \to 10\}$

 $s = 0:1: even(s)$ $even(x:s) = x: odd(s)$ $odd(x : s) = even(s)$

 $s = 0:1: even(s)$ $even(x:s) = x: odd(s)$ $odd(x : s) = even(s)$

 $0:1:0:0:$ *even*

Strategies to check for productivity

- Syntactic restriction cf Coq conductive format
- Term rewriting productivity via termination

Probabilistic programs p_{max} surely productive if it produces an infinite stream of outputs with probability 1. For instance,

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

Probabilistic programs p_{max} surely productive if it produces an infinite stream of outputs with probability 1. For instance,

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

A probabilistic stream computation is **almost surely productive (ASP)** if it produces an infinite stream of **productive (ASP)** if it produces an infinite stream of outputs with probability 1.

Probabilistic programs p_{max} surely productive if it produces an infinite stream of outputs with probability 1. For instance, rohabilietie prodra *‡* = (*a* : *‡*) ü*^p ‡*

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

Prob of no output = $(1-p)^n$ $\text{output} = (1 - p)^n$ Prob of no output =

A probabilistic stream computation is **almost surely productive (ASP)** if it produces an infinite stream of **productive (ASP)** if it produces an infinite stream of outputs with probability 1.

Probabilistic programs p_{max} surely productive if it produces an infinite stream of outputs with probability 1. For instance, rohabilietie prodra *‡* = (*a* : *‡*) ü*^p ‡*

$$
\sigma=(a:\sigma)\oplus_p\sigma\quad\text{ASP}\;\checkmark
$$

Prob of no output = $(1-p)^n$ $\text{output} = (1 - p)^n$ Prob of no output =

A probabilistic stream computation is **almost surely productive (ASP)** if it produces an infinite stream of **productive (ASP)** if it produces an infinite stream of outputs with probability 1.

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

 $\overline{a} = a : a : a : a : a : \cdots$

$$
\sigma = \bar{a} \oplus_p \epsilon \qquad \mathsf{ASP}
$$

 $\overline{a} = a : a : a : a : a : \cdots$

$$
\sigma = \bar{a} \oplus_p \epsilon \qquad \mathsf{ASP}
$$

 $\overline{a} = a : a : a : a : a : \cdots$

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

$$
\sigma = \overline{a} \bigoplus_{p} \epsilon \qquad \mathbf{ASP}
$$

$$
\overline{a} = a \cdot a \cdot a \cdot a \cdot \overline{a \cdot a \cdot \overline{a \cdot a \cdot \overline{a \cdot a \cdot \overline{a \cdot \overline{a \cdot a \cdot \overline{a \cdot \over
$$

$$
\omega \quad \omega \quad \omega \quad \omega \quad \omega \quad \omega
$$

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

$$
p \le 1/2 \qquad \text{ASP} \quad \checkmark
$$

$$
p > 1/2 \qquad \text{ASP}
$$

Almost surely productivity

- Formal definition that is language oblivious
- Can be generalised for different conductive types — streams, trees, etc
- Methods to verify ASP

Formal definition

A probabilistic stream computation is **almost surely productive (ASP)** if it produces an infinite stream of outputs with probability 1. productive (ASP) if it produces an infinite stream of outputs with probability 1.

Formal definition Γ

A probabilistic stream computation is **almost surely productive (ASP)** if it produces an infinite stream of foutputs with probability 1. productive (ASP) if it produces an infinite stream of outputs with probability 1. as infinite stream computation is almost surely **inary streams in the set of simplicity of the simulation of**

Fi*cht internet, we in the semantic semantics of the semantics of programs. Rather than fix a concrete program-denomination and the semantics of* ϵ

 $[\![-]\!]: \mathbb{T} \to \mathcal{D}((A_\bot)^\omega).$ $\llbracket - \rrbracket$: $\mathbb{T} \to \mathcal{D}((A_\perp)^\omega)$

(deterministic) Coinductive definitions

Probabilistic Coinductive definitions in the category. We take the work of t

I **Theorem 1** (Finality for streams [22])**.** *Given a set* T *of programs endowed with a probabilistic* $P(A^{\omega})$ is a final coalgebra but \dots *K* $l(\mathcal{D})$ $\mathcal{D}(A^{\omega})$ is a final coalgebra but … $Kl(\mathcal{D})$

Probabilistic Coinductive definitions and the space of probabilities of probabilities of probabilities of probabilities of probabilities in the category. We take the work of t First, we introduce the semantics of programs. Rather than fix a concrete programming at the point of the terms of terms of the

I **Theorem 1** (Finality for streams [22])**.** *Given a set* T *of programs endowed with a probabilistic* $P(A^{\omega})$ is a final coalgebra but \dots *K* $l(\mathcal{D})$ $\begin{array}{ccc} \nabla & \mathbb{R} & \mathbb{R} \nabla & \mathbb{R} \nabla \mathbb{R} & \mathbb$ *commutes in the Kleisli category K¸*(*D*)*:* \mathbb{T} $\begin{array}{ccc} & \mathbb{L} & \mathbb{C} \ & \mathbb{C} & \mathbb{C} \end{array}$ ∐−∐
—∩— / st ϕ ✏✏ \rightarrow $(A_{\perp})^{\frac{1}{2}}$ ¶*<*head*,*tail*>* ✏✏ $A_1 \times T$ $\frac{id\times[-]}{S}$ \cdot A_{\perp} \times $(A_{\perp})^{\omega}$ $\mathcal{D}(A^{\omega})$ is a final coalgebra but … $Kl(\mathcal{D})$ $\mathcal{D}(A^{\omega})$ is a final coalgebra but \ldots $Kl(\mathcal{D})$ is ASP if its probability of probability of probability of producing unbounded \mathcal{N} $\llbracket - \rrbracket \colon \mathbb{T} \to \mathcal{D}((A_\perp)^\omega)$ from a given one-step semantics function that maps each term to an output in *A*‹ and the $\llbracket - \rrbracket$ is probabilities function is probabilities for the Kleisli category for the Kleisli distribution monad; this introduces some computations when computing the final computations when computing the final coalgebras some computations when coalgebras some coalgebras some coalgebras some coalgebras some coalgeb $\text{st}\, \diamondsuit$ in this category. We take the work on probabilistic streams by \diamondsuit head, tail $>$ $\begin{array}{cc} \vee & & id \times \llbracket - \rrbracket & & \vee & & \vee & \ 1 & 1 & 1 \end{array}$

Probabilistic Coinductive definitions in the category. We take the work of t

I **Theorem 1** (Finality for streams [22])**.** *Given a set* T *of programs endowed with a probabilistic* $P(A^{\omega})$ is a final coalgebra but \dots *K* $l(\mathcal{D})$ $\mathcal{D}(A^{\omega})$ is a final coalgebra but … $Kl(\mathcal{D})$

Probabilistic Coinductive definitions in the category. We take the work of t

I **Definition 2** (ASP for streams)**.** A stream program *p* œ T is *almost surely productive* (ASP) Composition of \sim o \rightarrow arrows uses monad multiplication

ASP *each program a probability distribution of output streams such that the following diagram commutes in the Kleisli category K¸*(*D*)*:*

$\text{error of } \mathcal{D} \text{ and } \mathcal{D} \text{ and } \mathcal{D} \text{ is a } \text{error of } \mathcal{D} \text{ and } \mathcal{D} \text{ are } \mathcal{D} \text{ and } \mathcal{D} \text{$ A stream program $p:$ \mathbb{T} is **almost surely productive (ASP)** if

✏✏

/
A
A
A^{*(A}· <i>A*·⁾
^A</sup>

✏✏

A‹ ◊ T ¶

id◊J≠^K

Pr $\sigma \sim \llbracket p \rrbracket$ [σ has infinitely many concrete output elements $a \in A$] = 1.

For this to be a sensible definition, the event "*‡* has infinitely many concrete output

ASP *each program a probability distribution of output streams such that the following diagram commutes in the Kleisli category K¸*(*D*)*:*

$\text{error of } \mathcal{D} \text{ and } \mathcal{D} \text{ and } \mathcal{D} \text{ is a } \text{error of } \mathcal{D} \text{ and } \mathcal{D} \text{ are } \mathcal{D} \text{ and } \mathcal{D} \text{$ A stream program $p:$ \mathbb{T} is **almost surely productive (ASP)** if

✏✏

/
A
A
A^{*(A}· <i>A*·⁾
^A</sup>

✏✏

A‹ ◊ T ¶

id◊J≠^K

Pr $\sigma \sim \llbracket p \rrbracket$ [σ has infinitely many concrete output elements $a \in A$] = 1.

For this to be a sensible definition, the event "*‡* has infinitely many concrete output

(*A*‹)*^Ê [|] ^u* prefix of *v, u* ^œ (*A*‹)^ú*}*. Our definition evidently depends on the definition of definition parametrised on [[-]] *or* st : T —> A x T

Some remarks

- Definition of monad D involves basic measure theory
- Multiplication is given by integration
- All *events* need to be measurable

two children whereas a one children whereas a one children whereas a one can construct the construct these trees these trees trees to a construct the construction of the construction of the construction of the construction

in the tree. A bit more formally, let *^w* ^œ *{L, R}^Ê* be an infinite word on alphabet *{L, R}*.

Given any tree *t* œ Trees(*A*‹), *w* induces a single path *t^w* in the tree: from the root, the path

·) =

‡ ·

A tree program $p:$ T is **almost surely productive (ASP)** if I **Definition 5** (ASP for trees)**.** A tree program *p* œ T is *almost surely productive* (ASP) if $\forall w \in \{L, R\}^{\omega}$. Pr $t \sim \llbracket p \rrbracket$ $[t_w$ has infinitely many concrete output nodes $a \in A] = 1$.

‡

unf(*‡*) =

‡

Example: a language for streams **4 A Calculus for Probabilistic Streams and Trees** Now that we have introduced almost sure productivity, we consider how to verify this property. $strapms$ for producing streams and trees respectively. We suppose that outputs are drawn from some

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

Example: a language for streams **4 A Calculus for Probabilistic Streams and Trees** Now that we have introduced almost sure productivity, we consider how to verify this property. $strapms$ for producing streams and trees respectively. We suppose that outputs are drawn from some **A. Aguirre, G. Barthe, J. Hsu, A. Silva 374:7**

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

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

Example: a language for streams **4 A Calculus for Probabilistic Streams and Trees** Now that we have introduced almost sure productivity, we consider how to verify this property. $strapms$ for producing streams and trees respectively. We suppose that outputs are drawn from some **A. Aguirre, G. Barthe, J. Hsu, A. Silva 374:7** Γ restricts recursion to be global in nature, i.e., our language does not support mutual or

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

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

 $\mathsf{st}_e(\mathsf{tail}^k(a : e)) \triangleq \mathsf{st}_e(\mathsf{tail}^{k-1}(e))$ $e^{(2\pi i)^2}$ $e^{(2\pi i)^2}$ $e^{(2\pi i)^2}$ $e^{(2\pi i)^2}$ $e^{(2\pi i)^2}$ $\mathsf{st}_e(\mathsf{tail}^k(e_1 \oplus_p e_2)) \triangleq \mathsf{st}_e(\mathsf{tail}^k(e_1) \oplus_p \mathsf{tail}^k(e_2))$

Example: a language for streams **4 A Calculus for Probabilistic Streams and Trees** Now that we have introduced almost sure productivity, we consider how to verify this property. $strapms$ for producing streams and trees respectively. We suppose that outputs are drawn from some **A. Aguirre, G. Barthe, J. Hsu, A. Silva 374:7** Evomple: o Ionounono for restricts recursion to be global in nature, i.e., our language does not support mutual or Adilipid. didiiyudyo ivi terms reduce by scaling the result of stepping *e* and the result of stepping *e*^Õ by *p* and 1 ≠ *p* respectively, and then combining the distributions:

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

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

$$
\mathsf{st}_e(\mathsf{tail}^k(a : e)) \triangleq \mathsf{st}_e(\mathsf{tail}^{k-1}(e))
$$
\n
$$
\mathsf{st}_e(\mathsf{tail}^k(e_1 \oplus_p e_2)) \triangleq \mathsf{st}_e(\mathsf{tail}^k(e_1) \oplus_p \mathsf{tail}^k(e_2))
$$
\n
$$
\mathsf{st}_e(a : e') \triangleq \delta(\mathit{inl}(a, e'))
$$
\n
$$
\mathsf{st}_e(e') \triangleq \delta(\mathit{inr}(e'[e/\sigma])) \quad \text{otherwise}
$$

ASP I: syntactic measure **5.1 A Syntactic Measure** #(*a* : *e*) , #(*e*)+1 A J \vdash 1

$$
\#(c) \stackrel{\Delta}{=} 0
$$

$$
\#(e_1 \oplus_p e_2) \stackrel{\Delta}{=} p \cdot \#(e_1) + (1 - p) \cdot \#(e_2)
$$

$$
\#(a : e) \stackrel{\Delta}{=} \#(e) + 1
$$

$$
\#(\text{tail}(e)) \stackrel{\Delta}{=} \#(e) - 1
$$

the term. We can define a similar measure for tree terms: We can now state conditions for \mathbf{F} for \mathbf{F} for streams and trees. The streams and trees. **Theorem**

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

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

 $\sqrt{2\pi}$ $\mathcal{O} = (\mathcal{U} \cdot \mathcal{O}) \cup p$ can \mathcal{O} $\sigma = (a : \sigma) \oplus_p \mathsf{tail}(\sigma)$

I **Example 12.** Consider the stream definition *‡* = (*ı* : *‡*) ü*^p* tail(*‡*). The # measure of $\#(\sigma) = p \cdot 1 + (1-p) \cdot (-1) = 2p-1.$ $t = p \cdot 1 + (1 - p) \cdot (-1) = 2p - 1$ element *a*. Otherwise the program tries to compute the tail of the recursive call; the first $\#(\sigma) =$

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

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

I **Example 12.** Consider the stream definition *‡* = (*ı* : *‡*) ü*^p* tail(*‡*). The # measure of $\#(\sigma) = p \cdot 1 + (1-p) \cdot (-1) = 2p-1.$ $t = p \cdot 1 + (1 - p) \cdot (-1) = 2p - 1$ element *a*. Otherwise the program tries to compute the tail of the recursive call; the first $\#(\sigma) =$

ϵ $\#(0) > 0 \iff p > 1/2$ $\#(\sigma) > 0 \iff p > 1/2$

ASP II : Probabilistic model checking

Idea: Associate with a program a pPDA and reduce ASP to model checking

ASP II : Probabilistic model checking algorithm¹ is in **PSPACE**. The idea behind our encoding from terms to pPDAs is simple to describe. The states of the

The idea behind our encoding from terms to pPDAs is simple to describe. The states of the Idea: Associate with a program a pPDA and reduce ASP to model checking destructions. We use a single-letter state the number of α Idea: Associate with a program a pPDA and reduce ASP to

$$
\mathcal{A}_e = (\mathcal{S}_e, \{tl\}, \mathcal{T}_e)
$$

is the set of syntactic subterms of *e* and *T^e* is the following transition function:

 $\mathcal{T}_e((\sigma, a), (e, a)) = 1$ $\mathcal{T}_e((e_1 \oplus_p e_2, a), (e_1, a)) = p$ $\mathcal{T}_e((e_1 \oplus_p e_2, a), (e_2, a)) = 1 - p$ $\mathcal{T}_e((a': e', \bot), (e', \varepsilon)) = 1$ $\mathcal{T}_e((a': e', t\mathcal{l}), (e', \varepsilon)) = 1$ $\mathcal{T}_e((\mathsf{tail}(e'),a),(e',tl\cdot a)=1$ \mathcal{T}_{e} ((*e*)

ASP II : Probabilistic model checking
and we implicitly the strings and we implicit the strings and we implicit the single symbol or a single *Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*1*, a*)) = *p Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*2*, a*)) = 1 ≠ *p* \overline{a} *Te*((tail(*e*^Õ)*, a*)*,*(*e*^Õ *, tl · a*)=1 string. All non-specified transitions have zero probability. We define the set of *outputting* \overline{a} *Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*2*, a*)) = 1 ≠ *p Te*((*a*^Õ : *e*^Õ *, tl*)*,*(*e*^Õ *, Á*)) = 1 *Te*((tail(*e*^Õ)*, a*)*,*(*e*^Õ *, tl · a*)=1 $$ string. All non-specified transitions have zero probability. We define the set of *outputting ,* ‹)*}*, that is, configurations where the

provement stronger: the stronger: \mathbb{R} **Theorem**

I **Theorem 17.** *Let e be a stream term and let A^e be the corresponding pPDA. Then,* I Let *e* be a stream term and let A_e be the corresponding pPDA. Then,

Pr $t \sim \llbracket e \rrbracket$ $[t$ *has infinitely many output nodes* = Pr π *∼Paths(e,* ε *)* $[\pi \models \Box \Diamond \mathcal{O}].$ [*t has infinitely many output nodes*] = Pr $output$ *nod* $\left[\begin{matrix}f_{\rho}g\end{matrix}\right] = \frac{D}{\rho}$

1 Technically, this algorithm requires first encoding the LTL formula into a Deterministic Rabin Automaton and
The LTL formula into a Deterministic Rabin Automaton Automaton Automaton Automaton Automaton Automaton Automat

In particular, e is ASP if and only if for almost all runs π *starting in* (e, ε) *,* $\pi \models \Box \Diamond O$ *.* I_n particular, a is Λ SP if and only if for almost all runs π starting in (

ASP II : Probabilistic model checking
and we implicitly the strings and we implicit the strings and we implicit the single symbol or a single *Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*1*, a*)) = *p Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*2*, a*)) = 1 ≠ *p* \overline{a} *Te*((tail(*e*^Õ)*, a*)*,*(*e*^Õ *, tl · a*)=1 string. All non-specified transitions have zero probability. We define the set of *outputting* \overline{a} *Te*((*e*¹ ü*^p e*2*, a*)*,*(*e*2*, a*)) = 1 ≠ *p Te*((*a*^Õ : *e*^Õ *, tl*)*,*(*e*^Õ *, Á*)) = 1 *Te*((tail(*e*^Õ)*, a*)*,*(*e*^Õ *, tl · a*)=1 $$ string. All non-specified transitions have zero probability. We define the set of *outputting ,* ‹)*}*, that is, configurations where the

provement stronger: the stronger: \mathbb{R} **Theorem**

I **Theorem 17.** *Let e be a stream term and let A^e be the corresponding pPDA. Then,* I Let *e* be a stream term and let A_e be the corresponding pPDA. Then,

Pr $t \sim \llbracket e \rrbracket$ $[t$ *has infinitely many output nodes* = Pr π *∼Paths(e,* ε *)* $[\pi \models \Box \Diamond \mathcal{O}].$ [*t has infinitely many output nodes*] = Pr $output$ *nod* $\left[\begin{matrix}f_{\rho}g\end{matrix}\right] = \frac{D}{\rho}$

1 Technically, this algorithm requires first encoding the LTL formula into a Deterministic Rabin Automaton and
The LTL formula into a Deterministic Rabin Automaton Automaton Automaton Automaton Automaton Automaton Automat

In particular, e is ASP if and only if for almost all runs π *starting in* (e, ε) *,* $\pi \models \Box \Diamond O$ *.* I_n particular, a is Λ SP if and only if for almost all runs π starting in (

Model checking LTL formulas against pPDAs is decidable (Brazdil et al) => ASP is decidable for stream programs

Conclusions

- Simple definition non trivial formalisation
- Methods for stream can be extended to trees
- Richer languages, higher-order, other conductive types
- Different notions of ASP weaker
- (Non-)Dependency on the step relation