**Scientific Annals of Computer Science vol. 24 (2), 2014, pp. 177–216**
**doi: 10.7561/SACS.2014.2.177**

# Probabilistic Recursion Theory and Implicit Computational Complexity

Ugo Dal Lago[1], Sara Zuppiroli[2], Maurizio Gabbrielli[3]

**Abstract**

We show that probabilistic computable functions, i.e., those functions outputting distributions and computed by probabilistic Turing machines, can be characterized by a natural generalization of Church and Kleene's partial recursive functions. The obtained algebra, following Leivant, can be restricted so as to capture the notion of a polytime sampleable distribution, a key concept in average-case complexity and cryptography.

**Keywords:** probabilistic recursion theory, implicit computational complexity, probabilistic Turing machines

## 1 Introduction

Models of computation as introduced one after the other in the first half of the last century, were all designed around the assumption that *determinacy* is one of the key properties to be modeled: given an algorithm and an input to it, the sequence of computation steps leading to the final result is *uniquely* determined by the way an *algorithm* describes the state evolution. The great majority of the introduced models are *equivalent*, in that the classes

[1]Università di Bologna & INRIA, E-mail: dallago@cs.unibo.it
[2]Università di Bologna, E-mail: zuppirol@cs.unibo.it
[3]Università di Bologna & INRIA, E-mail: gabbri@cs.unibo.it

of functions (on, say, natural numbers) they are able to compute are the same [4].

The second half of the 20th century has seen the assumption above relaxed in many different ways. Nondeterminism, as an example, has been investigated as a way to abstract the behavior of certain classes of algorithms, this way facilitating their study without necessarily changing their expressive power: think about how NFAs [18] make the task of proving closure properties of regular languages easier.

A relatively recent step in this direction consists in allowing algorithms' internal state to evolve probabilistically: the next state is not *functionally* determined by the current one, but is obtained from it by performing a process having possibly many outcomes, each with a probability. Probabilistically evolving computation (probabilistic computation for short) can be a way to abstract over determinism, but also a way to model situations in which algorithms have access to a source of randomness[4]. Indeed, probabilistic models are nowadays more and more pervasive: not only they are a formidable tool when dealing with uncertainty and incomplete information, but they sometimes are a *necessity* rather than an option, like in computational cryptography (where, e.g., public key encryption schemes cannot be secure without being probabilistic [10]). Examples of application areas in which probabilistic computation has proved to be useful include natural language processing [15], robotics [22], computer vision [3], and machine learning [16].

But what does the presence of probabilistic choice give us in terms of expressivity? Are we strictly more expressive than usual, deterministic, computation? And what about efficiency: is it that probabilistic choice permits to solve computational problems more efficiently? These questions have been among the most central in the theory of computation, in particular in computational complexity, in the last forty years and they have received several different answers. We postpone to the next section a discussion of these answers, however we can already summarize two main points emerging from these results. First, while probability has been proved not to offer any computational advantage in the absence of resource constraints, it is not known whether probabilistic classes such as **BPP** or **ZPP** are different from **P**. Second, all the existing works on this subject follow an approach that we call *reductionist*: probabilistic computation is studied by reducing

[4] Although the physical sources of randomness algorithms have access to usually contain correlations and biases, they are modeled as sources of *perfect* randomness, in which bits are uniformly distributed and independent.

or comparing it to deterministic computation.

This work goes in a somehow different direction: as already mentioned, we want to study probabilistic computation directly, without necessarily *reducing* it to deterministic computation. In our perspective, the central assumption is the following: a probabilistic algorithm computes what we call a *probabilistic function*, i.e. a function from a discrete set (e.g. natural numbers or binary strings) to *distributions* over the same set. What we want to do is to study the set of those probabilistic functions which can be computed by algorithms, possibly with resource constraints.

In the first part of this paper we provide a characterization of computable probabilistic functions by the natural generalization of Kleene's partial recursive functions, where among the initial functions there is now a function corresponding to tossing a fair coin, thus modeling the access to a source of randomness. In the non-trivial proof of completeness for the obtained algebra, Kleene's minimization operator is used in an unusual way, making the usual proof strategy for Kleene's Normal Form Theorem (see, e.g., [21]) useless. We later hint at how to recover the latter by replacing minimization with a more powerful operator. We also mention how probabilistic recursion theory offers characterizations of concepts like the one of a computable distribution and of a computable real number.

The second part of this paper is devoted to applying the aforementioned recursion-theoretical framework to polynomial-time computation. We do that by following Bellantoni and Cook's and Leivant's works [1, 13], in which polynomial-time deterministic computation is characterized by a restricted form of recursion, called *predicative* or *ramified* recursion. Endowing Leivant's ramified recurrence with a random base function, in particular, is shown to provide a characterization of polynomial-time computable distributions, a key notion in average-case complexity [2].

This is a revised and extended version of an eponymous paper appeared in the proceedings of the 11th International Colloquium on Theoretical Aspects of Computing [6].

**Related Work.** This work is rooted in the classic theory of computation, and in particular in the definition of partial computable functions as introduced by Church and later studied by Kleene [12]. Relevant related work includes the many probabilistic computational models introduced so far. Without trying to be exhaustive we can mention that, starting from the early fifties, various forms of automata in which probabilistic choice is available

have been considered (e.g., see [17]). The inception of probabilistic choice into an universal model of computation, namely Turing machines, is due to Santos [19, 20], but is (essentially) already there in an earlier work by De Leeuw and others [7]. Some years later, Gill [8] considered probabilistic Turing machines with bounded complexity: his work has been the starting point of a florid research about the interplay between computational complexity and randomness. Among the many side effects of this research one can of course mention modern cryptography [11], in which algorithms (e.g. encryption schemes, authentication schemes, and adversaries for them) are almost invariably assumed to work in probabilistic polynomial time.

The second part of this work is related to the area of *implicit computational complexity*, which studies machine-free characterizations of complexity classes based on mathematical logic and programming language theory, and which is a relatively young research area. Its birth is traditionally made to correspond with the beginning of the nineties, when Bellantoni and Cook [1] and Leivant [13] independently proposed function algebras precisely characterizing (deterministic) polynomial time computable functions. In the last twenty years, this area has produced many interesting results, and complexity classes spanning from the logarithmic space computable functions to the elementary functions have been characterized by, e.g., function algebras, type systems [14], or fragments of linear logic [9]. Recently, some investigations on the interplay between implicit complexity and probabilistic computation have started to appear [5]. There is however an intrinsic difficulty in giving *implicit* characterizations of probabilistic classes like **BPP** or **ZPP**: these are *semantic* classes defined by imposing a polynomial bound on time, but also appropriate bounds on the probability of error. This makes the task of enumerating machines computing problems in the classes much harder and, ultimately, prevents from deriving implicit characterizations of the classes above. Again, our emphasis here is different: we do not see probabilistic algorithms as artifacts computing functions of the same kind as the one deterministic algorithms compute, but we see probabilistic algorithms as devices outputting distributions.

## 2 Probabilistic Recursion Theory

In this section we provide a characterization of the functions computed by a *probabilistic Turing machine* (PTM) in terms of a function algebra *à la* Kleene. We first define *probabilistic recursive functions*, which are the

elements of our algebra. Next we define formally the class of probabilistic functions computed by a PTM. Finally, we show the equivalence of the two introduced classes. In the following, $\mathbb{R}_{[0,1]}$ is the unit interval.

Since PTMs compute probability distributions, the functions that we consider in our algebra have domain $\mathbb{N}^k$ (the set of $k$-tuples in $\mathbb{N}$) and codomain $\mathbb{N} \to \mathbb{R}_{[0,1]}$ (rather than $\mathbb{N}$ as in the classic case). The idea is that if $f(x)$ is a function which returns $p \in \mathbb{R}_{[0,1]}$ on input $x \in \mathbb{N}$, then $p$ is the probability of getting $y \in \mathbb{N}$ as the output when feeding $f$ with the input $x$. We note that we could extend our codomain from $\mathbb{N} \to \mathbb{R}_{[0,1]}$ to $\mathbb{N}^m \to \mathbb{R}_{[0,1]}$, however we use $\mathbb{N} \to \mathbb{R}_{[0,1]}$ in order to simplify the presentation, at the same time being consistent with the classic literature on recursion theory.

**Definition 1 (Pseudodistributions, Probabilistic Functions)** *A* pseudodistribution *on* $\mathbb{N}$ *is a function* $\mathcal{D} : \mathbb{N} \to \mathbb{R}_{[0,1]}$ *such that* $\sum_{n\in\mathbb{N}} \mathcal{D}(n) \leq 1$. $\sum_{n\in\mathbb{N}} \mathcal{D}(n)$ *is often denoted by* $\sum \mathcal{D}$. *Let* $\mathbb{P}_{\mathbb{N}}$ *be the set of all pseudodistributions on* $\mathbb{N}$. *A* probabilistic function *(PF) is a function from* $\mathbb{N}^k$ *to* $\mathbb{P}_{\mathbb{N}}$.

In the following we use the expression $\{n_1^{p_1}, \ldots, n_k^{p_k}\}$ to denote the pseudodistribution $\mathcal{D}$ defined as $\mathcal{D}(n) = \sum_{n_i=n} p_i$. Observe that $\sum \mathcal{D} = \sum_{i=1}^{k} p_i$. When this does not cause ambiguity, a pseudodistribution is simply called a distribution. Please notice that probabilistic functions are always *total* functions, but their codomain is a set of distributions which do not necessarily sum to 1, but rather to a real number *smaller* or equal to 1, this way modeling the probability of divergence. For example, the nowhere-defined partial function $\Omega : \mathbb{N} \rightharpoonup \mathbb{N}$ of classic recursion theory becomes a probabilistic function which returns the empty distribution $\emptyset$ on any input. The first step towards defining our function algebra consists in giving a set of functions to start from:

**Definition 2 (Basic Probabilistic Functions)** *The* basic probabilistic functions *(BPFs) are defined as follows:*

- *The* zero function $z : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $z(x)(0) = 1$ *for every* $x \in \mathbb{N}$*;*
- *The* successor function $s : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $s(x)(x+1) = 1$ *for every* $x \in \mathbb{N}$*;*
- *The* projection function $\Pi_m^n : \mathbb{N}^n \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $\Pi_m^n(x_1, \ldots, x_n)(x_m) = 1$ *for every positive* $n, m \in \mathbb{N}$ *such that* $1 \leq m \leq n$*;*

- *The* fair coin function $r : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *that is defined as:*

$$r(x)(y) = \begin{cases} 1/2 & \text{if } y = x; \\ 1/2 & \text{if } y = x + 1; \\ 0 & \text{otherwise.} \end{cases}$$

The first three BPFs are essentially the same as the basic functions from classic recursion theory, while $r$ is the only truly probabilistic BPF: it behaves like the identity or like the successor, each with probability $\frac{1}{2}$. It is worth noting that, as we will show in Example 1, this definition of $r$ allow us to obtain probabilistic choice.

The next step consists in defining how PFs *compose*. Function composition of course cannot be used here, because when composing two PFs $f$ and $g$ the codomain of $g$ does not match with the domain of $f$. Indeed, $g$ returns a distribution $\mathbb{N} \to \mathbb{R}_{[0,1]}$ while $f$ expects a natural number as input. What we have to do here is the following. Given an input $x \in \mathbb{N}$ and an output $y \in \mathbb{N}$ for the composition $f \bullet g$, we apply the distribution $g(x)$ to a generic value $z \in \mathbb{N}$. This gives a probability $g(x)(z)$ which is then multiplied by the probability that the distribution $f(z)$ associates to the value $y \in \mathbb{N}$. If we then consider the sum of the obtained product $g(x)(z) \cdot f(z)(y)$ on all possible $z \in \mathbb{N}$ we obtain the probability of $f \bullet g$ returning $y$ when fed with $x$. The sum is due to the fact that two different values, say $z_1, z_2 \in \mathbb{N}$, which provide two different distributions $f(z_1)$ and $f(z_2)$ must both contribute to the same probability value $f(z_1)(y) + f(z_2)(y)$ for a fixed $y$. In other words, we are doing nothing more than lifting $f$ to a function from distributions to distributions, then composing it with $g$. Formally:

**Definition 3 (Composition)** *We define the* composition $f \bullet g : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *of two functions* $f : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *and* $g : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *as:*

$$((f \bullet g)(x))(y) = \sum_{z \in \mathbb{N}} f(z)(y) \cdot g(x)(z).$$

Please note that function composition as defined above is precisely the same as convolution from functional analysis. The previous definition can be generalized to functions taking more than one parameter in the expected way:

**Definition 4 (Generalized Composition)** *We define the* generalized composition *of functions* $f : \mathbb{N}^n \to \mathbb{P}_{\mathbb{N}}$, $g_1 : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}, \ldots, g_n : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ *as the*

*function* $f \odot (g_1, \ldots, g_n) : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$ *defined as follows:*

$$((f \odot (g_1, \ldots, g_n))(\mathbf{x}))(y) = \sum_{z_1,\ldots,z_n \in \mathbb{N}} f(z_1, \ldots, z_n)(y) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{x})(z_i).$$

With a slight abuse of notation, we can treat probabilistic functions as ordinary functions when forming expressions. Suppose, as an example, that $x \in \mathbb{N}$ and that $f : \mathbb{N}^3 \to \mathbb{P}_\mathbb{N}$, $g : \mathbb{N} \to \mathbb{P}_\mathbb{N}$, $h : \mathbb{N} \to \mathbb{P}_\mathbb{N}$. Then the expression $f(g(x), x, h(x))$ stands for the distribution in $\mathbb{P}_\mathbb{N}$ defined as follows: $(f \odot (g, id, h))(x)$, where $id = \Pi^1_1$ is the identity PF.

The way we have defined probabilistic functions and their composition is reminiscent of, and indeed inspired by, the way one defines the Kleisli category for the Giry monad, starting from the category of partial functions on sets. This categorical way of seeing the problem can help a lot in finding the right definition, but by itself is not adequate to proving the existence of a correspondence with machines like the one we want to give here.

Primitive recursion is defined as in Kleene's algebra, provided one uses composition as previously defined:

**Definition 5 (Primitive Recursion)** *Given functions* $g : \mathbb{N}^{k+2} \to \mathbb{P}_\mathbb{N}$, *and* $f : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$, *the function* $h : \mathbb{N}^{k+1} \to \mathbb{P}_\mathbb{N}$ *defined as*

$$h(\mathbf{x}, 0) = f(\mathbf{x}); \qquad h(\mathbf{x}, y+1) = g(y, \mathbf{x}, h(\mathbf{x}, y));$$

*is said to be defined* by primitive recursion *from* $f$ *and* $g$, *and is denoted as* $rec(f, g)$.

We now turn our attention to the minimization operator which, as in the deterministic case, is needed in order to obtain the full expressive power of (P)TMs. The definition of this operator is in our case delicate and requires some explanation. Recall that, in the classic case, given a partial function $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$, the minimization operator allows one to define another partial function, call it $\mu f$, which computes from $\mathbf{x} \in \mathbb{N}^k$ the least value of $y$ such that $f(\mathbf{x}, y)$ is equal to 0 (and $f(\mathbf{x}, z)$ is defined and different from 0 for all $z < y$), if such a value exists (and is undefined otherwise). In our case, again, we are concerned with distributions, hence we cannot simply consider the least value on which $f$ returns 0, since functions return 0 *with a certain probability*. The idea is then to define the minimization $\mu f$ as a function which, given an input $\mathbf{x} \in \mathbb{N}^k$, returns a distribution associating to each natural $y$ the probability that the result of $f(\mathbf{x}, y)$ is 0 *and* the result of $f(\mathbf{x}, z)$ is strictly positive for every $z < y$. Formally:

**Definition 6 (Minimization)** *Given a PF* $f : \mathbb{N}^{k+1} \to \mathbb{P}_{\mathbb{N}}$*, we define another PF* $\mu f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ *as follows:*

$$(\mu f)(\mathbf{x})(y) = f(\mathbf{x}, y)(0) \cdot \left( \prod_{z<y} \sum_{k>0} f(\mathbf{x}, z)(k) \right).$$

We are finally able to define the class of functions we are interested in as follows.

**Definition 7 (Probabilistic Recursive Functions)** *The class* $\mathscr{PR}$ *of* probabilistic recursive functions *is the smallest class of probabilistic functions that contains the BPFs (Definition 2) and is closed under the operations of* general composition *(Definition 4),* primitive recursion *(Definition 5) and* minimization *(Definition 6).*

**Example 1** *The following are examples of probabilistic recursive functions:*

- *The* identity function $id : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *is defined as follows: for all* $x, y \in \mathbb{N}$

$$id(x)(y) = \begin{cases} 1 & \text{if } y = x; \\ 0 & \text{otherwise.} \end{cases}$$

  *This definition means that* $id = \Pi_1^1$*. Since the latter is a BPF (Definition 2),* $id$ *is in* $\mathscr{PR}$*.*
- *The probabilistic function* $rand : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *such that, for every* $x \in \mathbb{N}$*,*

$$rand(x)(y) = \begin{cases} 1/2 & \text{if } y = 0; \\ 1/2 & \text{if } y = 1; \\ 0 & \text{otherwise;} \end{cases}$$

  *can be easily shown to be recursive, since* $rand = r \odot z$ *(and we know that both* $r$ *and* $z$ *are BPF). Actually, one can easily see that rand could itself be taken as the only genuinely probabilistic BPF, i.e.,* $r$ *can be constructed from rand and the other BPFs by composition and primitive recursion.*
- *All functions we have proved recursive so far have the property that the returned distribution is* finite *and* total *for any input. Indeed, this is true for every probabilistic* primitive *recursive function, since minimization is the only way to break finiteness and totality. Consider the function* $f : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as*

$$f(x)(y) = \begin{cases} \frac{1}{2^{y-x+1}} & \text{if } y \geq x; \\ 0 & \text{otherwise.} \end{cases}$$

*We define another function* $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *by stipulating that*

$$h(x)(y) = \frac{1}{2^{y+1}}$$

*for every* $x, y \in \mathbb{N}$. $h$ *is a probabilistic recursive function; indeed, consider the function* $k : \mathbb{N}^2 \to \mathbb{P}_{\mathbb{N}}$ *defined as* $rand \odot \Pi_1^2$ *and build* $\mu k$. *By definition,*

$$(\mu k)(x)(y) = k(x, y)(0) \cdot \prod_{z<y} \sum_{q>0} k(x, z)(q). \tag{1}$$

*Then observe that* $(\mu k)(x)(y) = \frac{1}{2^{y+1}}$*: by (1),* $(\mu k)(x)(y)$ *unfolds into a product of exactly* $y + 1$ *copies of* $\frac{1}{2}$*, each "coming from the flip of a distinct coin". Hence,* $h = \mu k$. *Then we observe that*

$$(add \odot (\mu k, id))(x)(y) = \sum_{z_1, z_2} add(z_1, z_2)(y) \cdot ((\mu k)(x)(z_1) \cdot id(x)(z_2)).$$

*But notice that* $id(x)(z_2) = 1$ *only when* $z_2 = x$ *(and in the other cases* $id(x)(z_2) = 0$*),* $(\mu\ k)(x)(z_1) = \frac{1}{2^{z_1+1}}$*, and* $add(z_1, z_2)(y) = 1$ *only when* $z_1 + z_2 = y$ *(and in the other cases,* $add(z_1, z_2)(y) = 0$*). This implies that the term in the sum is different from* 0 *only when* $z_2 = x$ *and* $z_1 + z_2 = y$*, namely when* $z_1 = y - z_2 = y - x$*, and in that case its value is* $\frac{1}{2^{y-x+1}}$. *Thus, we can claim that* $f = (add \odot (\mu k, id))$*, and that* $f$ *is in* $\mathscr{PR}$.

It is easy to show that $\mathscr{PR}$ includes all partial recursive functions, seen as probabilistic functions. This can be done by first defining extended recursive functions as follows.

**Definition 8 (Extended Recursive Functions)** *For every partial recursive function* $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ *we define the extended function* $p_f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ *as follows:*

$$p_f(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = f(\mathbf{x}); \\ 0 & \text{otherwise.} \end{cases}$$

**Proposition 1** *If* $f$ *is partial recursive function, then* $p_f$ *as defined above is in* $\mathscr{PR}$.

**Proof.** The proof goes by induction on the structure of $f$ as a partial recursive function. The proof for the base cases is immediate. As for the inductive cases, we have the following ones:

- $f$ is defined by composition from $h, g_1, \ldots, g_n$ as:

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), \ldots, g_n(\mathbf{x})),$$

where $h : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g_i : \mathbb{N}^k \rightharpoonup \mathbb{N}$ for every $1 \leq i \leq n$ are partial recursive functions. By definition, we have

$$p_f(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = h(g_1(\mathbf{x}), \ldots, g_n(\mathbf{x})); \\ 0 & \text{otherwise} \end{cases}$$

Also by definition, we have that for every $1 \leq i \leq n$, it holds that:

$$p_{g_i}(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = g_i(\mathbf{x}); \\ 0 & \text{otherwise;} \end{cases}$$
$$p_h(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = h(\mathbf{x}); \\ 0 & \text{otherwise.} \end{cases}$$

By induction hypothesis, we observe that $p_{g_1}, \ldots, p_{g_n}, p_h \in \mathscr{P}\mathscr{R}$ and

$$\begin{aligned} &((p_h \odot (p_{g_1}, \ldots, p_{g_n}))(\mathbf{x}))(y) \\ &\quad = \sum_{z_1,\ldots,z_n \in \mathbb{N}} p_h(z_1, \ldots, z_n)(y) \cdot \left( \prod_{1 \leq i \leq n} p_{g_i}(\mathbf{x})(z_i) \right) \\ &\quad = p_f(\mathbf{x})(y). \end{aligned}$$

by using the definitions above. Thus $p_f = (p_h \odot (p_{g_1}, \ldots, p_{g_n})) \in \mathscr{P}\mathscr{R}$.
- $f$ is defined by primitive recursion so $f : \mathbb{N}^k \times \mathbb{N} \to \mathbb{N}$ is defined as:

$$f(\mathbf{x}, 0) = h(\mathbf{x});$$
$$f(\mathbf{x}, y+1) = g(y, \mathbf{x}, f(\mathbf{x}, y));$$

where $g : \mathbb{N}^{k+2} \to \mathbb{N}$ and $h : \mathbb{N}^k \to \mathbb{N}$ are partial recursive functions. By induction hypothesis, we have that $p_g, p_h \in \mathscr{P}\mathscr{R}$. The fact that $rec(p_g, p_h)(\mathbf{x}, y) = p_f(\mathbf{x}, y)$ can be proved by an induction on $y$. Thus $p_f$ is in $\mathscr{P}\mathscr{R}$ because $p_f = rec(p_h, p_g)$.
- Suppose $f : \mathbb{N}^k \to \mathbb{N}$ is defined by minimization, i.e. $f = \mu g$. By definition of $f(\mathbf{x})$ we have:

$$p_f(\mathbf{x})(z) = \begin{cases} 1 & \text{if } z = \mu y.(g(\mathbf{x}, y) = 0); \\ 0 & \text{otherwise;} \end{cases}$$

By hypothesis $p_g \in \mathscr{PR}$. We observe that:

$$\begin{aligned}
(\mu p_g)(\mathbf{x})(z) &= p_g(\mathbf{x}, z)(0) \cdot \left( \prod_{n<z} \sum_{k>0} p_g(\mathbf{x}, n)(k) \right) \\
&= p_g(\mathbf{x}, z)(0) \cdot \left( \prod_{n<z} \sum_{k=g(\mathbf{x},n)>0} 1 \right) \\
&= \begin{cases} 1 & \text{if } g(\mathbf{x}, z) = 0 \text{ and } \forall n < z . g(\mathbf{x}, z) > 0; \\ 0 & \text{otherwise;} \end{cases} \\
&= \begin{cases} 1 & \text{if } z = \mu y . (g(\mathbf{x}, y) = 0) \\ 0 & \text{otherwise;} \end{cases} \\
&= p_f(\mathbf{x})(z).
\end{aligned}$$

Thus $p_f$ is in $\mathscr{PR}$ because $p_f = \mu\, p_g$.
This concludes the proof. □

## 2.1 Probabilistic Turing Machines and Computable Functions

In this section we introduce *computable* probabilistic functions as those probabilistic functions which can be computed by probabilistic Turing machines. As previously mentioned, probabilistic computation models have received a wide interest in computer science already in the fifties [7] and early sixties [17]. A natural question which arose was then to see what happened if random elements were allowed in a Turing machine. This question led to several formalizations of probabilistic Turing machines [7, 19] — which, essentially, are Turing machines which have the ability to flip coins in order to make uniform, fair, decisions — and to several results concerning the computational complexity of problems when solved by PTMs [8].

Following [8], a probabilistic Turing machine (PTM) can be seen as a Turing machine with two transition functions $\delta_0, \delta_1$. At each computation step, either $\delta_0$ or $\delta_1$ can be applied, each with probability $\frac{1}{2}$. Then, in a way analogous to the deterministic case, we can define a notion of a (initial, final) configuration for a PTM. In the following, $\Sigma_b$ denotes the set of possible symbols on the tape, including a blank symbol $\Box$; $Q$ denotes the set of states; $Q_f \subseteq Q$ denotes the set of final states and $q_s \in Q$ denotes the initial state.

**Definition 9 (Probabilistic Turing Machine)** *A probabilistic Turing machine (PTM) is a Turing machine endowed with two transition functions*

*$\delta_0, \delta_1$. At each computation step the transition function $\delta_0$ can be applied with probability $\frac{1}{2}$ and the transition $\delta_1$ can be applied with probability $\frac{1}{2}$.*

**Definition 10 (Configuration of a PTM)** *Let $M$ be a PTM. We define a PTM* configuration *as a 4-tuple $\langle s, a, t, q\rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that:*

- *The first component, $s \in \Sigma_b^*$, is the portion of the tape lying on the left of the head.*
- *The second component, $a \in \Sigma_b$, is the symbol the head is reading.*
- *The third component, $t \in \Sigma_b^*$, is the portion of the tape lying on the right of the head.*
- *The fourth component, $q \in Q$ is the current state.*

*Moreover we define the set of all configurations as $\mathcal{C}_M = \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$.*

**Definition 11 (Initial and Final Configurations of a PTM)** *Let $M$ be a PTM. We define the* initial configuration *of $M$ for the string $s$ as the configuration $\langle \varepsilon, a, v, q_s\rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that $s = a \cdot v$ and the fourth component, $q_s \in Q$, is the initial state. We denote it with $\mathcal{IN}_M^s$. Similarly, we define a* final configuration *of $M$ for $s$ as a configuration $\langle s, \Box, \varepsilon, q_f\rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q_f$. The set of all such final configurations for $M$ is denoted by $\mathcal{FC}_M^s$.*

For a function $T : \mathbb{N} \to \mathbb{N}$, we say that a PTM $M$ *runs in time bounded by* $T$ if for any input $x$, $M$ halts on input $x$ within $T(|x|)$ steps *independently* of the random choices it makes. Thus, $M$ *works in polynomial time* if it runs in time bounded by $P$, where $P$ is any polynomial.

Intuitively, the function computed by a PTM $M$ associates to each input $s \in \Sigma^*$ a pseudodistribution which indicates the probability of reaching a final configuration of $M$ from $\mathcal{IN}_M^s$. It is worth noticing that, differently from the deterministic case, since in a PTM the same configuration can be obtained along different computation paths, the probability of reaching a given final configuration is the *sum* of the probabilities of reaching the configuration along all computation paths, of which there can be (even infinitely) many. It is thus convenient to define the function computed by a PTM through a fixpoint construction, as follows. First, we can define a partial order on the string distributions.

**Definition 12** *A* string (pseudo)distribution *on $\Sigma^*$ is a function $\mathcal{D} : \Sigma^* \to \mathbb{R}_{[0,1]}$ such that $\sum_{s\in\Sigma^*} \mathcal{D}(s) \leq 1$. $\mathbb{P}_{\Sigma^*}$ denotes the set of all string (pseudo) distributions on $\Sigma^*$.*

Next we can define a partial order on string distributions by a pointwise extension of the usual order on $\mathbb{R}$:

**Definition 13** *The relation $\sqsubseteq_{\mathbb{P}_{\Sigma^*}} \subseteq \mathbb{P}_{\Sigma^*} \times \mathbb{P}_{\Sigma^*}$ is defined by stipulating that $A \sqsubseteq_{\mathbb{P}_{\Sigma^*}} B$ if and only if, for all $s \in \Sigma^*$, $A(s) \leq B(s)$.*

The proof of the following is immediate.

**Proposition 2** *The structure $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$ is a poset.*

It is time to define the domain $\mathcal{CEV}$:

**Definition 14** *The set $\mathcal{CEV}$ is defined as $\{f \mid f : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}\}$.*

The set $\mathcal{CEV}$ will be used as the domain[5] of the functional whose least fixpoint gives the function computed by a PTM. To this aim, inheriting the structure on $\mathbb{P}_{\Sigma^*}$, we can define a partial order on $\mathcal{CEV}$ as follows.

**Definition 15** *The relation $\sqsubseteq_{\mathcal{CEV}} \subseteq \mathcal{CEV} \times \mathcal{CEV}$ is defined for $A, B \in \mathcal{CEV}$ as $A \sqsubseteq_{\mathcal{CEV}} B$ if and only if, for all $c \in \mathcal{C}_M$, $A(c) \sqsubseteq_{\mathbb{P}_{\Sigma^*}} B(c)$.*

The proof of the following is also immediate.

**Proposition 3** *The structure $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$ is a poset.*

Given a poset, the notions of least upper bound, denoted by $\bigsqcup$, and of an ascending chain are defined as usual. Next, the bottom elements of the posets of our interest can be obtained as follows.

**Lemma 1** *Let $d_\bot : \Sigma^* \to \mathbb{R}_{[0,1]}$ be defined by stipulating that $d_\bot(s) = 0$ for all $s \in \Sigma^*$. Then, $d_\bot$ is the bottom element of the poset $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$.*

**Lemma 2** *Let $b_\bot : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}$ be defined by stipulating that $b_\bot(c) = d_\bot$ for all $c \in \mathcal{C}_M$. Then, $b_\bot$ is the bottom element of the poset $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$.*

Now, it is time to prove that the posets at hand are also $\omega$-complete, that is, that each ascending chain in the poset has a least upper bound in it.

**Proposition 4** *The poset $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$ is a $\omega CPO$.*

[5] Of course $\mathcal{CEV}$ is a proper superset of the functions computed by PTMs.

**Proof.** We need to prove that for each chain $c_1 \sqsubseteq_{\mathbb{P}_{\Sigma^*}} c_2 \sqsubseteq_{\mathbb{P}_{\Sigma^*}} c_3 \ldots$ the least upper bound $\bigsqcup_i c_i$ exists. First note that since $\sum_{s\in\Sigma^*} c_i(s) \leq 1$, from definition of $\sqsubseteq_{\mathbb{P}_{\Sigma^*}}$ it follows that, for each $s \in \Sigma^*$, $c_1(s) \leq c_2(s) \leq \ldots \leq 1$ holds. This implies that, for each $s \in \Sigma^*$, the limit $\lim_{i\to\infty} c_i(s)$ exists. Hence, defining $c_{\mathcal{L}}$ as the distribution such that $c_{\mathcal{L}}(s) = \lim_{i\to\infty} c_i(s)$, we have that $c_{\mathcal{L}} = \bigsqcup_i c_i$. Indeed, $c_{\mathcal{L}} \sqsupseteq_{\mathbb{P}_{\Sigma^*}} c_i$, and any upper bound of the family $\{c_i\}_{i\in\mathbb{N}}$ is clearly greater or equal to $c_{\mathcal{L}}$. □

**Proposition 5** *The poset $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$ is a $\omega$CPO.*

**Proof.** Analogous to the previous one. □

We can now define a functional $F_M$ on $\mathcal{CEV}$ which will be used to define the function computed by $M$ via a fixpoint construction. Intuitively, the application of the functional $F_M$ describes *one* computation step. Formally:

**Definition 16** *Given a PTM $M$, we define a functional $F_M : \mathcal{CEV} \to \mathcal{CEV}$ as:*

$$F_M(f)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}^s_M; \\ \frac{1}{2}f(\delta_0(C)) + \frac{1}{2}f(\delta_1(C)) & \text{otherwise.} \end{cases}$$

Note that, according to the notation introduced after Definition 1, $\{s^1\}$ is the distribution which assigns probability 1 to $s$. The following proposition is needed in order to apply the usual fixpoint result.

**Proposition 6** *The functional $F_M$ is continuous.*

**Proof.** We prove that $F_M(\bigsqcup_{i\in\mathbb{N}} f_i) = \bigsqcup_{i\in\mathbb{N}} F_M(f_i)$, or, saying another way, that for every configuration $C$, $F_M(\bigsqcup_{i\in\mathbb{N}} f_i)(C) = \bigsqcup_{i\in\mathbb{N}}(F_M(f_i))(C)$. Now, notice that for every $C$,

$$F_M(\bigsqcup_{i\in\mathbb{N}} f_i)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}^s_M; \\ \frac{1}{2}((\bigsqcup_{i\in\mathbb{N}} f_i)(C_1)) + \frac{1}{2}((\bigsqcup_{i\in\mathbb{N}} f_i)(C_2)) & \text{if } C \to C_1, C_2; \end{cases}$$

where $C \to C_1, C_2$ means that, from the configuration $C$, the machine with one computation step evolves to the configurations $C_1$ and $C_2$ and, similarly, that:

$$\bigsqcup_{i\in\mathbb{N}}(F_M(f_i))(C) = \bigsqcup_{i\in\mathbb{N}} \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}^s_M; \\ \frac{1}{2}f_i(C_1) + \frac{1}{2}f_i(C_2) & \text{if } C \to C_1, C_2. \end{cases}$$

Now, given any $C$, we distinguish two cases:

- If $C \in \mathcal{FC}^s_M$, then

$$F_M(\bigsqcup_{i\in\mathbb{N}} f_i)(C) = \{s^1\} = \bigsqcup_{i\in\mathbb{N}} \{s^1\} = \bigsqcup_{i\in\mathbb{N}} (F_M(f_i))(C).$$

- If $C \to C_1, C_2$, then

$$\begin{aligned} F_M(\bigsqcup_{i\in\mathbb{N}} f_i)(C) &= \frac{1}{2}((\bigsqcup_{i\in\mathbb{N}} f_i)(C_1)) + \frac{1}{2}((\bigsqcup_{i\in\mathbb{N}} f_i)(C_2)) \\ &= \frac{1}{2}(\bigsqcup_{i\in\mathbb{N}} f_i(C_1)) + \frac{1}{2}(\bigsqcup_{i\in\mathbb{N}} f_i(C_2)) \\ &= \bigsqcup_{i\in\mathbb{N}} \frac{1}{2} f_i(C_1) + \bigsqcup_{i\in\mathbb{N}} \frac{1}{2} f_i(C_2) = \bigsqcup_{i\in\mathbb{N}} (\frac{1}{2} f_i(C_1) + \frac{1}{2} f_i(C_2)) \\ &= \bigsqcup_{i\in\mathbb{N}} (F_M(f_i))(C). \end{aligned}$$

This concludes the proof. □

**Theorem 1** *The functional $F_M$ from Definition 16 has a least fixpoint which is equal to $\bigsqcup_{n\geq 0} F^n_M(b_\perp)$.*

**Proof.** Immediate from the well-known fixpoint theorem for continuous maps on a $\omega$CPO. □

Such a least fixpoint is, once composed with a function returning $\mathcal{IN}^s_M$ from $s$, the *function computed by the machine* $M$, which is denoted as $\mathcal{IO}_M : \Sigma^* \to \mathbb{P}_{\Sigma^*}$. A probabilistic function is *computable* if it is the function computed by any PTM $M$. The set of computable functions is $\mathscr{PC}$.

The fixpoint construction delineated above is an appropriate way to *define* what a PTM computes, although working with it can be quite cumbersome in proofs. A better, equivalent, definition consists in working with *computation trees*, each of which represents all probabilistic computation paths of a machine $M$ when fed with a given input string $x$. We define such a tree as follows. Each node is labelled by a configuration of the machine and each edge represents a computation step. The root is labelled with the initial configuration $\mathcal{IN}^x_M$ and each node labelled with $C$ has either no child (if $C$ is final) or 2 children (otherwise) labelled with $\delta_0(C)$ and $\delta_1(C)$. Please notice that the same configuration may be duplicated across

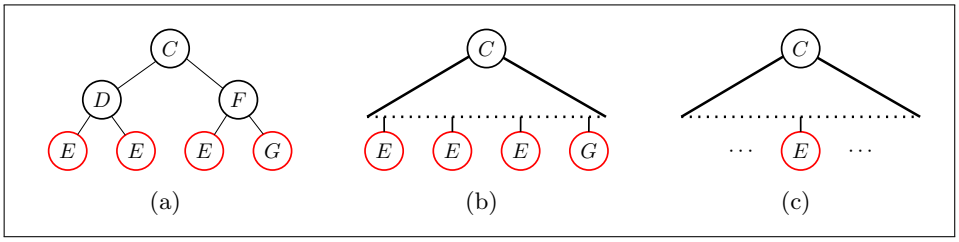


Figure 1: Computation Trees, some Examples

a single level of the tree as well as appear at different levels of the tree; nevertheless we represent each such appearance by a separate node. In Figure 1(a), an example computation trees is depicted such that $C$ is the initial configuration, $E$ and $G$ are final configurations, while $D$ and $F$ are neither initial nor final. The same computation tree can be represented more abstractly as in Figure 1(b), or even as in Figure 1(c), where we focus on one among the nodes labelled with $E$.

We can naturally associate a real number to each node of a computation tree, corresponding to the probability that the node is reached in the computation: it is $\frac{1}{2^n}$, where $n$ is the height of the node. The probability of a particular *final* configuration is the sum of the probabilities of all leaves labelled with that configuration. We also enumerate nodes in the tree, top-down and from left to right, by using binary strings in the following way: the root has associated the number $\varepsilon$. Then if $b$ is the binary string representing the node $N$, the left child of $N$ has associated the string $b \cdot 0$ while the right child has the string $b \cdot 1$. Note that from this definition it follows that each binary number associated to a node $N$ indicates a path in the tree from the root to $N$. The computation tree whose root is labelled with $\mathcal{IN}_M^x$ will be denoted as $CT_M(x)$.

It is worth noticing that the notion of computable probabilistic function we have described subsumes other key notions in probabilistic and real-number computation. As an example, *computable distributions* can be characterized as those distributions on $\Sigma^*$ which can be obtained as the result of a function in $\mathscr{PC}$ on a *fixed* input. Analogously, *computable real numbers* from the unit interval $[0, 1]$ can be seen as those elements of $\mathbb{R}$ in the form $f(0)(0)$ for a computable function $f \in \mathscr{PC}$.

## 2.2 Equivalence

In this section we prove that probabilistic *recursive* functions are the same as probabilistic *computable* functions, modulo an appropriate bijection between strings and natural numbers, which we denote (as its inverse) with $\overline{(\cdot)}$.

### 2.2.1 Soundness

In order to prove the equivalence result we first need to show that any probabilistic recursive function can be computed by a PTM. This result is not difficult and, analogously to the deterministic case, is proved by exhibiting PTMs which simulate the basic probabilistic recursive functions and by showing that $\mathscr{PC}$ is closed under composition, primitive recursion, and minimization. This is done by the following lemmata.

**Lemma 3 (Basic Functions and Computability)** *All basic probabilistic functions are computable.*

**Proof.** For every basic function from Definition 2, we can construct a probabilistic Turing machine that computes it. More specifically, the proof is immediate for the BPFs $z$, $s$ and $\Pi_n^m$: they are deterministic, thus we can use the usual Turing machines (seen as a PTMs) for them. As for the function $r$ it can be simulated by a PTM $M$ which writes $\overline{1}$ or $\overline{0}$ on the tape, both with probability $\frac{1}{2}$, and then halts. □

The composition of two computable probabilistic functions is itself computable:

**Lemma 4 (Composition and Computability)** *Given computable $f : \mathbb{N}^n \to \mathbb{P}_\mathbb{N}$ and $g_1 : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}, \ldots, g_n : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$, the function $f \odot (g_1, \ldots, g_n) : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$ is itself computable.*

**Proof.** We give the proof for the case when $n = 1$, i.e., the case in which the function to be proved computable is $f \bullet g : \mathbb{N}^k \to \mathbb{P}_\mathbb{N}$ (the general case is analogue, if only a bit more tedious). By hypothesis, $f$ and $g$ are both computable, and thus there are PTMs which compute them, say $N$ and $M$ respectively. We define a PTM $L$, working on 2 tapes[6], and which computes $f \bullet g$. On the first tape $L$ simulates $M$ by computing the value of $g$ on the

[6]The equivalence of multi tape PTMs with single tape PTMs can be proved in a way which is analogous to the classic case.

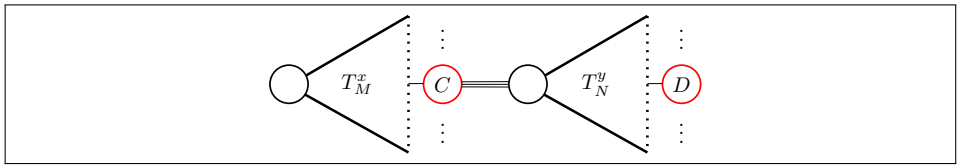


Figure 2: Computation Tree for Composition

input, while on the second tape $L$ simulates $N$ by computing the result of $f$ on the result of $g$. A bit more in detail, the machine $L$ operates as follows on a input $x$:

1. it first computes $g(x)$ by simulating $M$ on the first tape;
2. it then copies the content $y$ of the first tape to the second tape;
3. it finally computes the function $f(y)$ by simulating $N$, and obtaining $z$.

But is it that $L$ correctly computes $f \bullet g$? On the one hand, one can observe that the computation tree $CT_L(x)$ has a structure like the one in Figure 2, where $T_M^x$ has the same structure as $CT_M(x)$, $C$ corresponds to a final configuration for $M$ and $y$, and $T_N^y$ has the same structure as $CT_N(y)$. Now, for every $D$ which is final for $N$ and $z$, the probability of reaching the corresponding configuration in $CT_L(x)$ is clearly

$$\sum_y g(x)(y) \cdot f(y)(z) = ((f \bullet g)(x))(z),$$

which is the thesis. □

**Lemma 5 (Primitive Recursion and Computability)** *Given computable functions $g : \mathbb{N}^{k+2} \to \mathbb{P}_{\mathbb{N}}$ and $f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$, the function $rec(f, g) : \mathbb{N}^{k+1} \to \mathbb{P}_{\mathbb{N}}$ is itself computable.*

**Proof.** By hypothesis, $f$ and $g$ are both computable, and thus there are PTMs which compute them, say $M$ and $N$ respectively. We define a PTM $L$ computing $rec(f, g)$ and working on 5 tapes. The first tape is the input tape, on the next tape $L$ keeps track of a counter, on the third tape $L$ computes $g$, on the fourth tape $L$ computes $f$, and in the last tape it saves the result. The machine $L$ operates as follows:

1. it copies to the second tape the value 0 and then it copies to the fourth tape (an encoding of) the first $k$ elements of the input;
2. it computes $f$ by simulating $M$ on the fourth tape and saves the result on the last tape;

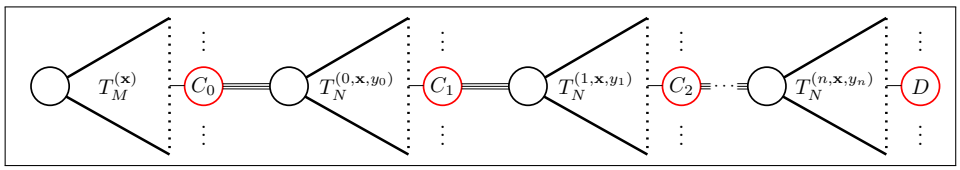


Figure 3: Computation Tree for Primitive Recursion

3. it verifies if the second tape contains the $k+1^{th}$ element of the input (which is on the first tape). In this case $L$ stops and the last tape contains the result, otherwise it copies the first $k$ elements of the input from the first tape to the third tape, then it copies the value on the second tape to the third tape and then it copies the result present on the last tape to the third tape;
4. $L$ increments the value on the second tape;
5. it computes $g$ on the third tape and saves the result on the last tape;
6. it goes back to Step 3. above.

One can observe that the computation tree $CT_L(\mathbf{x}, n)$ has a structure like the one in Figure 3, where:

- $T_M^{(\mathbf{x})}$ has the same structure as $CT_M(\mathbf{x})$;
- $C_0$ corresponds to a final configuration for $M$ and $\mathbf{x}$;
- for every $0 \leq i \leq n$, $T_N^{(i,\mathbf{x},y_i)}$ has the same structure as $CT_N(i, \mathbf{x}, y_i)$.
- for every $0 \leq i < n$, $C_{i+1}$ corresponds to a final configuration for $N$ and $y_i$.

Now, for every $D$ which is final for $N$ and $z$, the probability of reaching such a configuration in $CT_L(\mathbf{x}, n)$ can be proved to be equal to $(rec(f, g))(\mathbf{x}, n)(z)$, by induction on $n$. □

**Lemma 6 (Minimization and Computability)** *Given a computable function $f : \mathbb{N}^{k+1} \to \mathbb{P}_{\mathbb{N}}$, the function $(\mu f) : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ is itself computable.*

**Proof.** We proceed as in Lemma 4 and Lemma 5. Since $f$ is computable, there is a PTM $M$ which computes it. We define a PTM $N$ which works with 4 tapes, and which computes $\mu f$. The first tape is the input tape, in the second tape $N$ saves a counter that corresponds to the $y$ (in the definition of minimization) and which is incremented iteratively, on the third tape it computes the function $f$ and in the last tape it saves the result. The machine $M$ operates as follows:

1. it writes 0 to the second tape;

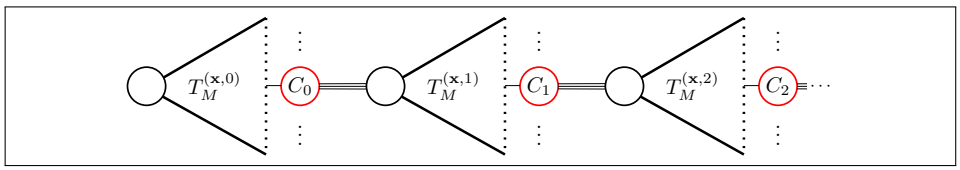


Figure 4: Computation Tree for Minimization

2. it copies to the third tape the input and the value of the second tape as a tuple;
3. it computes on the third tape the function $f$ and saves the result on the last tape;
4. it checks whether the last tape contains the value 0; in this case it saves on the last tape the element in the second tape and it stops, otherwise it increases the value of the second tape by 1;
5. it returns to the step 2. above.

One can observe that the computation tree $CT_L(\mathbf{x})$ has a structure like the one in Figure 4, where for every $y \geq 0$:

- $T_M^{(\mathbf{x},y)}$ has the same structure as $CT_M(\mathbf{x}, y)$;
- $C_y$ corresponds to a final configuration for $M$ and $k$, and is actually linked to the initial configuration of $T_M^{(\mathbf{x},y+1)}$ only if $k$ is different than 0.

By the usual kind of reasoning, we can prove that the probability of reaching a given final configuration of $C$ in $L$ is precisely the one given from the definition of $\mu f$. □

We can finally prove the following theorem, showing that all probabilistic recursive functions are computable:

**Theorem 2** $\mathscr{PR} \subseteq \mathscr{PC}$

**Proof.** The fact that $f \in \mathscr{PR}$ implies $f \in \mathscr{PC}$ can be proved by induction on the structure of the proof that $f \in \mathscr{PR}$, where lemmas 3, 4, 5 and 6 each handle an inductive case. □

### 2.2.2 Completeness

The most difficult part of the equivalence proof consists in proving that each probabilistic *computable* function is actually *recursive*. Analogously to the classic case, a good strategy consists in representing configurations as

natural numbers, then encoding the transition function of the machine at hand, call it $M$, as a (recursive) function on $\mathbb{N}$. In the classic case the proof proceeds by making essential use of the minimization operator to determine the *number* of transition steps of $M$ necessary to reach a final configuration (if such a number exists). This number can then be fed to another function which simulates $M$ (on an input) a given number of steps, and which is primitive recursive. In our case, this strategy does not work: the number of computation steps can be infinite, even when the convergence probability is 1.

Before entering into the technicalities, we need some preliminary definitions. First we need to encode the rational numbers $\mathbb{Q}$ into $\mathbb{N}$. Let $pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be any recursive bijection between pairs of natural numbers and natural numbers such that *pair* and its inverse are both computable. Let then *enc* be just $p_{pair}$, i.e. the function $enc : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows:

$$enc(a,b)(q) = \begin{cases} 1 & \text{if } q = pair(a,b); \\ 0 & \text{otherwise.} \end{cases}$$

The function *enc* allows us to represent positive rational numbers as pairs of natural numbers in the obvious way and is probabilistic recursive. It is now time to define a few notions on computation trees

**Definition 17 (Computation Tree and String Probability)** *The function $PT_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ is defined by stipulating that $PT_M(x,y)$ is the probability of observing the string $\overline{y}$ in the tree $CT_M(x)$, namely $\frac{1}{2^{|\overline{y}|}}$.*

Of course, $PT_M$ is partial recursive, thus $p_{PT_M}$ is probabilistic recursive. Since the same configuration $C$ can label more than one node in a computation tree $CT_M(x)$, $PT_M$ does not indicate the probability of reaching $C$, even when $C$ is the label of the node corresponding to the second argument. Such a probability can be obtained by summing the probability of all nodes labelled with the configuration at hand:

**Definition 18 (Configuration Probability)** *Suppose given a PTM $M$. If $x \in \mathbb{N}$ and $z \in \mathcal{C}_M$, the subset $CC_M(x,z)$ of $\mathbb{N}$ contains precisely the indices of nodes of $CT_M(x)$ which are labelled by $z$. The function $PC_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ is defined as follows: $PC_M(x,z) = \Sigma_{y \in CC_M(x,z)} PT_M(x,y)$.*

Contrary to $PT_M$, there is nothing guaranteeing that $PC_M$ is indeed computable. In the following, indeed, we will not prove completeness through a proof of computability for $PC_M$, but rather through a long detour.

Please recall the example computation tree $CT_M(x)$ for an hypothetical PTM $M$ and an input $x$ as in Figure 1(a). As can be easily checked, $PC_M(x, C) = 1$, while $PC_M(x, E) = \frac{3}{4}$. Indeed, notice that there are three nodes in the tree which are labelled with $E$, namely those corresponding to the binary strings 00, 01, and 10.

As we already mentioned, our proof separates the classic part of the computation performed by the underlying PTM, which essentially computes the configurations reached by the machine in different paths, from the probabilistic part, which instead computes the probability values associated to each computation by using minimization. These two tasks are realized by two suitable probabilistic recursive functions, which are then composed to obtain the function computed by the underlying PTM. We start with the probabilistic part, which is more complicated.

We need to define a function which returns the *conditional* probability of terminating at the node corresponding to the string $\overline{y}$ in the tree $CT_M(x)$, given that all the nodes $\overline{z}$ where $z < y$ are labelled with non-final configurations. This is captured by the following definition:

**Definition 19** *Given a PTM $M$, we define $PT^0_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ and $PT^1_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ as follows:*

$$PT^1_M(x, y) = \begin{cases} 1 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ 1 - PT^0_M(x, y) & \text{otherwise;} \end{cases}$$

$$PT^0_M(x, y) = \begin{cases} 0 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ \frac{PT_M(x,y)}{\prod_{k<y} PT^1_M(x,k)} & \text{otherwise;} \end{cases}$$

Note that, according to the previous definition, $PT^1_M(x, y)$ is the probability of *not* terminating the computation in the node $y$, while $PT^0_M(x, y)$ represents the probability of terminating the computation in the node $y$, both *knowing* that the computation has not terminated in any node $k$ preceding $y$.

**Proposition 7** *The functions $PT^0_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ and $PT^1_M : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ are partial recursive.*

**Proof.** Please observe that $PT_M$ is partial recursive and that the definitions above are mutually recursive, but the underlying order is well-founded. Both functions are thus intuitively computable, thus partial recursive by the Church-Turing thesis. □

The reason why the two functions above are useful is because they associate the distribution $\{0^{PT^0_M(x,y)}, 1^{PT^1_M(x,y)}\}$ to each pair of natural numbers $(x, y)$. In Figure 5, we give the quantities we have just defined for the tree from Figure 1(a). Each internal node is associated with the same distribution $\{0^0, 1^1\}$. Only the leaves are associated with nontrivial distributions. As an example, the distribution associated to the node 01 is $\{0^{\frac{1}{3}}, 1^{\frac{2}{3}}\}$, because we have that

$$\begin{aligned} PT^0_M(x, \overline{01}) &= \frac{PT_M(x, \overline{01})}{\prod_{k<\overline{01}} PT^1_M(x, k)} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{00}) \cdot PT^1_M(x, \overline{1}) \cdot PT^1_M(x, \overline{0}) \cdot PT^1_M(x, \overline{\varepsilon})} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{00})}. \end{aligned}$$

As it can be easily verified, $PT^1_M(x, \overline{00}) = \frac{3}{4}$. Thus, $PT^0_M(x, \overline{01}) = \frac{1}{3}$.

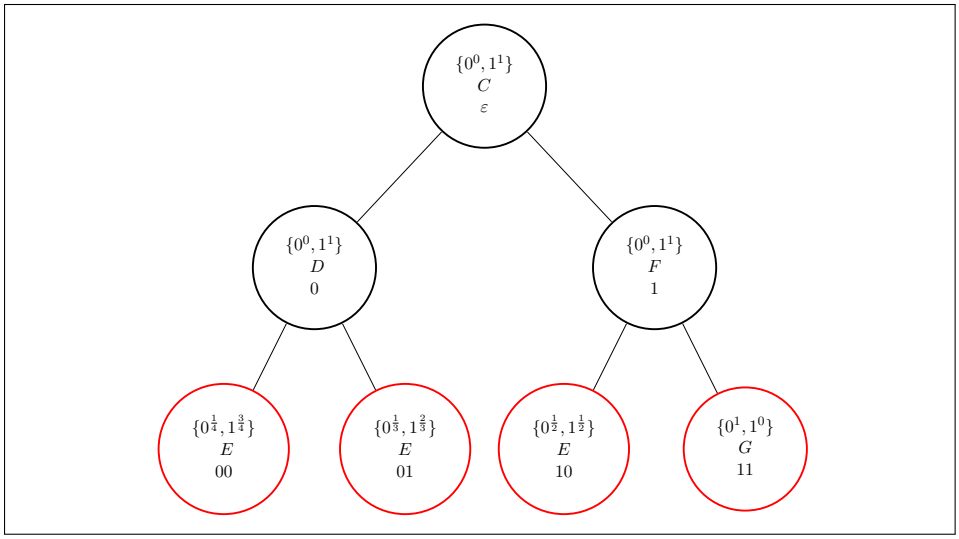


Figure 5: The Conditional Probabilities for the Computation Tree from Figure 1(a)

We now need to go further, and prove that the probabilistic function returning, on input $(x, y)$, the distribution $\{0^{PT^0_M(x,y)}, 1^{PT^1_M(x,y)}\}$ is recursive. This is captured by the following definition:

**Definition 20** *Given a PTM $M$, the function $PTC_M : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$PTC_M(x,y)(z) = \begin{cases} PT^0_M(x,y) & \text{if } z = 0; \\ PT^1_M(x,y) & \text{if } z = 1; \\ 0 & \text{otherwise.} \end{cases}$$

The function $PTC_M$ is really the core of our encoding. On the one hand, we will show that it is indeed recursive. On the other, minimizing it is going to provide us exactly with the function we need to reach our final goal, namely proving that the probabilistic function computed by $M$ is itself recursive. But how should we proceed if we want to prove $PTC_M$ to be recursive? The idea is to compose $p_{PT^1_M}$ with a function that turns its input into the probability of returning 1. This is precisely what the following function does:

**Definition 21** *The function $I2P : \mathbb{Q} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$I2P(x)(y) = \begin{cases} x & \text{if } (0 \leq x \leq 1) \wedge (y = 1); \\ 1 - x & \text{if } (0 \leq x \leq 1) \wedge (y = 0); \\ 0 & \text{otherwise.} \end{cases}$$

Please observe how the input to *I2P* is the set of rational numbers, as usual encoded by pairs of natural numbers. Previous definitions allow us to treat (rational numbers representing) probabilities in our algebra of functions. Indeed:

**Proposition 8** *The probabilistic function I2P is recursive.*

**Proof.** We first observe that $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ defined as $h(x)(y) = \frac{1}{2^{y+1}}$ is a probabilistic recursive function, because $h = \mu\ (rand \odot \Pi^2_1)$. Next we observe that every $q \in \mathbb{Q} \cap [0,1]$ can be represented in binary notation as: $q = \sum_{i \in \mathbb{N}} \frac{c^q_i}{2^{i+1}}$ where $c^q_i \in \{0,1\}$ (i.e., $c^q_i$ is the $i$-th element of the binary representation of $q$). Moreover, a function computing such a $c^q_i$ from $q$ and $i$ is partial recursive. Hence we can define $b : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ as follows

$$b(q,i)(y) = \begin{cases} 1 & \text{if } y = c^q_i; \\ 0 & \text{otherwise;} \end{cases}$$

and conclude that $b$ is indeed a probabilistic recursive function (because $\mathscr{PR}$ includes all the partial recursive functions, seen as probabilistic functions). Observe that:

$$b(q,i)(y) = \begin{cases} c^q_i & \text{if } y = 1; \\ 1 - c^q_i & \text{if } y = 0. \end{cases}$$

From the definition of composition, it follows that

$$\begin{aligned}(b \odot (id, h))(q)(y) &= \sum_{x_1,x_2} b(x_1, x_2)(y) \cdot id(q)(x_1) \cdot h(q)(x_2) \\ &= \sum_{x_2} b(q, x_2)(y) \cdot h(q)(x_2) \\ &= \sum_{x_2} b(q, x_2)(y) \cdot \frac{1}{2^{x_2+1}} \\ &= \begin{cases} \sum_{x_2} \frac{c^q_{x_2}}{2^{x_2+1}} & \text{if } y = 1 \\ \sum_{x_2} \frac{1-c^q_{x_2}}{2^{x_2+1}} & \text{if } y = 0 \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} q & \text{if } y = 1 \\ 1 - q & \text{if } y = 0 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

This shows that $I2P = b \odot (id, h)$, and hence that $I2P$ is probabilistic recursive. □

The following is an easy corollary of what we have obtained so far:

**Proposition 9** *The probabilistic function $PTC_M$ is recursive.*

**Proof.** Just observe that $PTC_M = I2P \odot p_{PT^1_M}$. □

The probabilistic recursive function obtained as the minimization of $PTC_M$ allows to compute a probabilistic function that, given $x$, returns $y$ with probability $PT_M(x, y)$ *if* $y$ is a leaf (and otherwise the probability is just 0).

**Definition 22** *The function $\mathcal{CF}_M : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

$$\mathcal{CF}_M(x)(y) = \begin{cases} PT_M(x, y) & \textit{if } y \textit{ corresponds to a leaf;} \\ 0 & \textit{otherwise.} \end{cases}$$

**Proposition 10** *The probabilistic function $\mathcal{CF}_M$ is recursive.*

**Proof.** The probabilistic function $\mathcal{CF}_M$ is just the function obtained by minimizing $PTC_M$, which we already know to be recursive. Indeed, if $y$

corresponds to a leaf, then:

$$\begin{aligned}(\mu PTC_M)(x)(y) &= PTC_M(x,y)(0) \cdot \prod_{z<y} \sum_{k>0} PTC_M(x,z)(k) \\ &= PTC_M(x,y)(0) \cdot \prod_{z<y} PTC_M(x,z)(1) \\ &= PT^0_M(x,y) \cdot \prod_{z<y} PT^1_M(x,z) \\ &= \frac{PT_M(x,y)}{\prod_{z<y} PT^1_M(x,z)} \cdot \prod_{z<y} PT^1_M(x,z) = PT_M(x,y).\end{aligned}$$

If, however, $y$ does not correspond to a leaf, then:

$$\begin{aligned}(\mu PTC_M)(x)(y) &= PTC_M(x,y)(0) \cdot \prod_{z<y} \sum_{k>0} PTC_M(x,z)(k) \\ &= PT^0_M(x,y)(0) \cdot \prod_{z<y} \sum_{k>0} PTC_M(x,z)(k) = 0.\end{aligned}$$

This concludes the proof. □

We are almost ready to wrap up our result, but before proceeding further, we need to define the function $SP_M : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that, given in input a pair $(x, y)$ returns the (encoding) of the string found in the configuration labeling the node $y$ in $CT_M(x)$. $SP_M$ is of course recursive. We can now prove the desired result:

**Theorem 3** $\mathscr{PC} \subseteq \mathscr{PR}$.

**Proof.** It suffices to note that, given any PTM $M$, the function computed by $M$ is nothing more than $p_{SP_M} \odot (id, \mathcal{CF}_M)$. Indeed, one can easily realize that a way to simulate $M$ consists in generating, from $x$, all strings corresponding to the leaves of $CT_M(x)$, each with an appropriate probability. This is indeed what $\mathcal{CF}_M$ does. What remains to be done is simulating $p_{SP_M}$ along paths leading to final configurations. □

We are finally ready to prove the main result of this Section:

**Corollary 1** $\mathscr{PR} = \mathscr{PC}$

**Proof.** Immediate from Theorem 2 and Theorem 3. □

The way we prove Corollary 1 implies that we cannot deduce Kleene's Normal Form Theorem from it: minimization has been used many times, some of them "deep inside" the construction. A way to recover Kleene's Theorem consists in replacing minimization with a more powerful operator, essentially corresponding to computing the fixpoint of a given probabilistic function.

## 3 Characterizing Probabilistic Complexity by Tiering

In this section we provide a characterization of the probabilistic functions which can be computed in polynomial time by an algebra of functions acting on word algebras. More precisely, we define a type system inspired by Leivant's notion of tiering [13], which permits to rule out functions having a too-high complexity, thus allowing to isolate the class of *predicatively recursive probabilistic functions*. Our main result in this section is that the class $\mathscr{PPC}$ of probabilistic functions which can be computed by a PTM in *polynomial* time equals the class of predicatively recursive probabilistic functions.

The constructions from Section 2 can be easily generalized to a function algebra on strings in a given alphabet $\Sigma$, which themselves can be seen a *word algebra* $\mathbb{W}$. Base functions include a function computing the empty string, called $e$, and concatenation with any character $a \in \Sigma$, called $c_a$. Projections remain of course available, while the only truly random function is one that concatenate a random symbol from $\Sigma$ to a given string, called again $r_a$. Composition and primitive recursion are available, although the latter takes the form of recursion *on notation*. We do not need minimization: the distribution a polytime computable probabilistic function returns (on any input) is always finite, and primitive recursion is powerful enough for our purposes.

Now we give a formal definition of our functions starting from the sets of domain and codomain of our functions.

**Definition 23 (String Distribution)** *A* (pseudo)distribution *on* $\mathbb{W}$ *is a function* $D : \mathbb{W} \to \mathbb{R}_{[0,1]}$ *such that* $\sum_{w \in \mathbb{W}} D(w) = 1$. *The set* $\mathbb{P}_{\mathbb{W}}$ *is defined as the set of all (pseudo)distributions on* $\mathbb{W}$.

The functions in our algebra have domain $\mathbb{W}^k$ and codomain $\mathbb{P}_{\mathbb{W}}$. The idea, as usual, is that $f(\mathbf{v})(w) = p$ means that $w$ is the output obtained

from the input $\mathbf{v}$ with probability $p$. Base functions are defined as follows:

$$e(v)(w) = \begin{cases} 1 & \text{if } w = \varepsilon; \\ 0 & \text{otherwise;} \end{cases}$$

$$c_a(v)(w) = \begin{cases} 1 & \text{if } w = a \cdot v; \\ 0 & \text{otherwise.} \end{cases}$$

Note that, for every $v \in \mathbb{W}$, the length of the word obtained after the application of one of the constructors $c_a$ is $|v| + 1$ with probability 1. Projections $\Pi^n_m : \mathbb{W}^n \to \mathbb{P}_{\mathbb{W}}$ are defined as follows:

$$\Pi^n_m(\mathbf{v})(w) = \begin{cases} 1 & \text{if } w = v_m; \\ 0 & \text{otherwise.} \end{cases}$$

As previously mentioned, the only truly random functions in our algebra are probabilistic functions in the form $r_a : \mathbb{W} \to \mathbb{P}_{\mathbb{W}}$, which concatenate $a$ to the input string (with probability $\frac{1}{2}$), or leave it unchanged (with probability $\frac{1}{2}$). Formally,

$$r_a(v)(w) = \begin{cases} 1/2 & \text{if } w = a \cdot v; \\ 1/2 & \text{if } w = v; \\ 0 & \text{otherwise.} \end{cases}$$

Next we recall the concept of composition and recurrence introduced in Definition 4 and Definition 5 and we instantiate them to the case of our algebra. We first introduce the *generalized composition* of functions $f : \mathbb{W}^n \to \mathbb{P}_{\mathbb{W}}$, $g_1, \ldots, g_n : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ as the function $f \odot (g_1, \ldots, g_n) : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ defined as follows:

$$((f \odot (g_1, \ldots, g_n))(\mathbf{v}))(w) = \sum_{z_1, \ldots, z_n \in \mathbb{W}} \left( f(z_1, \ldots, z_n)(w) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{x})(z_i) \right).$$

Recurrence over $\mathbb{W}$ takes the following form:

$$f(\varepsilon, \mathbf{v}) = g_\varepsilon(\mathbf{v});$$
$$f(a \cdot w, \mathbf{v}) = g_a(w, \mathbf{v}, f(w, \mathbf{v}));$$

where $f : \mathbb{W}^{m+1} \to \mathbb{P}_{\mathbb{W}}$, $g_a : \mathbb{W}^{m+2} \to \mathbb{P}_{\mathbb{W}}$, for all $a \in \Sigma$ and $g_\varepsilon : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$. Analogously to what we have done in Section 2 we write $f = rec(g_\varepsilon, \{g_a\}_{a \in \Sigma})$ in this case. The following construction is redundant in presence of primitive recursion, but becomes essential when predicatively restricting it.

**Definition 24 (Case Distinction)** *If $g_\varepsilon : \mathbb{W}^k \to \mathbb{P}_\mathbb{W}$ and for every $a \in \Sigma$, $g_a : \mathbb{W}^{k+1} \to \mathbb{P}_\mathbb{W}$, the function $h : \mathbb{W}^{k+1} \to \mathbb{P}_\mathbb{W}$ such that $h(\varepsilon, \mathbf{v}) = g_\varepsilon(\mathbf{v})$ and $h(a \cdot w, \mathbf{v}) = g_a(w, \mathbf{v})$ is said to be defined by* case distinction *from $g_\varepsilon$ and $\{g_a\}_{a\in\Sigma}$ and is denoted as $case(g_\varepsilon, \{g_a\}_{a\in\Sigma})$.*

In the following we will need also the following definition of simultaneous recursion:

**Definition 25 (Simultaneous Recursion)** *We say that the functions $\mathbf{f} = (f^1, \ldots, f^n)$ are defined by* simultaneous primitive recursion *over a word algebra $\mathbb{W}$ from the functions $g^j_\varepsilon : \mathbb{W}^m \to \mathbb{W}$ and $g^j_a : \mathbb{W}^{n+m+1} \to \mathbb{W}$ (where $j \in \{1, \ldots, n\}$ and $a \in \Sigma$) if the following holds for every $j$ and for every $a$:*

$$\begin{aligned} f^j(\varepsilon, \mathbf{w}) &= g^j_\varepsilon(\mathbf{w}); \\ f^j(a \cdot v, \mathbf{w}) &= g^j_a(v, \mathbf{w}, f^1(v, \mathbf{w}), \ldots, f^n(v, \mathbf{w})). \end{aligned}$$

A function $f^j$ as defined above will be indicated with $simrec^j(\{g^j_\varepsilon\}_j, \{g^j_a\}_{j,a})$.

**Example 2** *The previous definition allows us to define, for instance, two functions $f^1$ and $f^2$ over a word algebra with $\Sigma = \{a, b\}$, as follows:*

$$\begin{aligned} f^j(\varepsilon, \mathbf{v}) &= g^j_\varepsilon(\mathbf{v}) \qquad \forall j \in \{1, 2\}; \\ f^j(a \cdot w, \mathbf{v}) &= g^j_a(w, \mathbf{v}, f^1(w, \mathbf{v}), f^2(w, \mathbf{v})) \qquad \forall j \in \{1, 2\}; \\ f^j(b \cdot w, \mathbf{v}) &= g^j_b(w, \mathbf{v}, f^1(w, \mathbf{v}), f^2(w, \mathbf{v})) \qquad \forall j \in \{1, 2\}. \end{aligned}$$

### 3.1 Tiering as a Typing System

Now we define our type system which will then be used to introduce the definition of the class of *predicatively probabilistic functions* and therefore to obtain our complexity result. The type system is inspired by the tiering approach due to Leivant [13]. The idea behind tiering consists in working with denumerably many copies of the underlying algebra $\mathbb{W}$, each indexed by a natural number $n \in \mathbb{N}$ and denoted by $\mathbb{W}_n$. Type judgments take the form $f \triangleright \mathbb{W}_{n_1} \times \ldots \times \mathbb{W}_{n_k} \to \mathbb{W}_m$, where $f : \mathbb{W}^k \to \mathbb{W}$. In the following, with slight abuse of notation, $\mathbf{W}$ stands for any expression in the form $\mathbb{W}_{i_1} \times \cdots \times \mathbb{W}_{i_j}$. Typing rules are given in Figure 6. The idea here is that, when generating functions by primitive recursion, one goes from a level (tier)

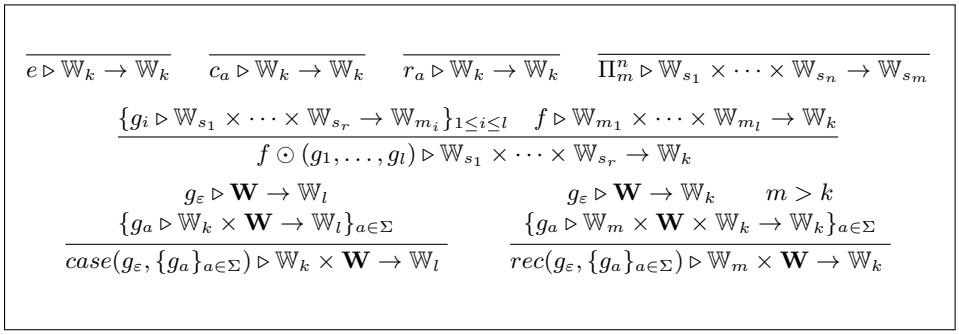


Figure 6: Tiering as a Typing System

$m$ for the domain to a *strictly* lower level $k$ for the result. This predicative constraint ensures that recursion does not cause any complexity explosion.

Those probabilistic functions $f : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ such that $f$ can be given a type through the rules in Figure 6 are said to be *predicatively recursive*. More precisely, the class $\mathscr{PT}$ of all predicatively recursive functions is defined as follows.

**Definition 26** *The class $\mathscr{PT}$ of* predicatively recursive (probabilistic) functions *is the smallest class of functions that contains the basic functions and is closed under the operations of general composition, primitive recursion, case distinction (Definition 24) and such that each function can be given a type through the rules in Figure 6.*

Next we give the definition of the class of simultaneously predicative recursive functions $\mathscr{ST}$.

**Definition 27** *The class $\mathscr{ST}$ of* simultaneously predicative recursive (probabilistic) functions *is the smallest class of probabilistic functions that contains the basic functions and is closed under the operations of general composition, simultaneous recursion (Definition 25), case distinction (Definition 24) and such that each function can be given a type through the rules in Figure 6, plus the rule below:*

$$\frac{\{g^j_\varepsilon \triangleright \mathbf{W} \to \mathbb{W}_k\}_j \quad m > k \quad \{g^j_a \triangleright \mathbb{W}^n_k \times \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k\}_{j,a}}{simrec^j(\{g^j_\varepsilon\}_j, \{g^j_a\}_{j,a}) \triangleright \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k}$$

### 3.2 Simultaneous Primitive Recursion and Predicative Recursion

By closely following Leivant [13], we can show that simultaneous primitive recursion can be encoded into predicative recursion. Since the proof of this result is precisely the one given by Leivant ([13] Section 4.2), we only sketch the main ingredients of it here.

According to Definition 25, if, e.g., two functions $f^0, f^1$ over a word algebra with $\Sigma = \{a, b\}$, are defined by simultaneous recursion, then we have that

$$\begin{aligned}
f^0(\varepsilon, \mathbf{x}) &= g^0_\varepsilon(\mathbf{x});\\
f^0(a \cdot w, \mathbf{x}) &= g^0_a(w, \mathbf{x}, f^0(w, \mathbf{x}), f^1(w, \mathbf{x}));\\
f^0(b \cdot w, \mathbf{x}) &= g^0_b(w, \mathbf{x}, f^0(w, \mathbf{x}), f^1(w, \mathbf{x}));\\
f^1(\varepsilon, \mathbf{x}) &= g^1_\varepsilon(\mathbf{x});\\
f^1(a \cdot w, \mathbf{x}) &= g^1_a(w, \mathbf{x}, f^0(w, \mathbf{x}), f^1(w, \mathbf{x}));\\
f^1(b \cdot w, \mathbf{x}) &= g^1_b(w, \mathbf{x}, f^0(w, \mathbf{x}), f^1(w, \mathbf{x})).
\end{aligned}$$

The two functions $f^0$ and $f^1$ can indeed be computed by *one* function $\widetilde{f}$ once a pairing operator $\langle \cdot, \cdot \rangle$ is available:

$$\begin{aligned}
\widetilde{f}(\varepsilon, \mathbf{x}) &= \langle g^0_\varepsilon(\mathbf{x}), g^1_\varepsilon(\mathbf{x}) \rangle;\\
\widetilde{f}(a \cdot w, \mathbf{x}) &= \langle g^0_a(w, \mathbf{x}, \widetilde{f}(w, \mathbf{x})), g^1_a(w, \mathbf{x}, \widetilde{f}(w, \mathbf{x})) \rangle;\\
\widetilde{f}(b \cdot w, \mathbf{x}) &= \langle g^0_b(w, \mathbf{x}, \widetilde{f}(w, \mathbf{x})), g^1_b(w, \mathbf{x}, \widetilde{f}(w, \mathbf{x})) \rangle.
\end{aligned}$$

The pairing function $\langle \cdot, \cdot \rangle$ is of course primitive recursive, and the same holds for the corresponding projection function. But can we give all these functions a "balanced" type, a type in which the tier of the argument(s) is the same as the tier of the output? (This is of course necessary if one wants to encode simultaneous primitive recursion the way suggested by the equations above.) A positive answer can indeed be given *provided* pairing and projections take an additional parameter (of an higher tier) big enough to "drive" the recursion necessary for computing pairing and projections. More details can be found in [13].

### 3.3 Register Machines vs. Turing Machines

Register machines are abstract computational models which, when properly defined, are Turing powerful. Here we extend the classical definition of a

register machine to the probabilistic case. Again, the way register machines are defined closely follows Leivant's proof [13].

**Definition 28 (Probabilistic Register Machine)** *A* probabilistic register machine (*PRM*) *consists of a finite set of registers* $\Pi = \{\pi_1, \ldots, \pi_r\}$ *and a sequence of instructions, called a* program. *Each register* $\pi_i$ *can store a string in* $\mathbb{W}$*, and each instruction in the program is indexed by a natural number and takes one of the following six forms*

$$\varepsilon(\pi_d); \quad c_a(\pi_s)(\pi_d); \quad r_a(\pi_s)(\pi_d); \quad p(\pi_s)(\pi_d); \quad j_\varepsilon(\pi_s)(m); \quad j_a(\pi_s)(m);$$

*where* $\pi_s, \pi_d$ *are registers and* $m$ *is an instruction index.*

The semantic of previous instructions can be described as follows. We assume that the index of the current instruction is $n$.

- The instruction $e(\pi_d)$ stores in the register $\pi_d$ the empty string and then transfers the control to the next instruction.
- The instruction $c_a(\pi_s)(\pi_d)$ stores in the register $\pi_d$ the term $a \cdot w$, where $w$ is the string contained in the register $\pi_s$. It then transfers the control to the next instruction.
- The instruction $p(\pi_s)(\pi_d)$ is the predecessor instruction, which stores in the register $\pi_d$ the string resulting from erasing the leftmost character from the string contained in $\pi_s$, if any. The control is then transferred to the next instruction.
- If $w$ is the string contained in $\pi_s$, the instruction $r_a(\pi_s)(\pi_d)$ stores in the register $\pi_d$ either the string $w$ (with probability $\frac{1}{2}$) or the string $a \cdot w$ (with probability $\frac{1}{2}$).
- The instruction $j_\varepsilon(\pi_s)(m)$ transfers the control to the $m$-th instruction if $\pi_s$ contains the empty string, and goes to the next instruction otherwise.
- The instruction $j_a(\pi_s)(m)$ transfers the control to the $m$-th instruction if $\pi_s$ contains a string whose leftmost character is $a$, and goes to the next instruction otherwise.

We can now describe more precisely the semantics of a PRM in terms of configurations.

**Definition 29 (Configuration of a PRM)** *Let* $R$ *be a PRM as in Definition 28, and let* $\Sigma$ *be the underlying alphabet. We define a PRM* configuration *as a tuple* $\langle v_1, \ldots, v_r, n\rangle$ *where:*

- *each* $v_i \in \Sigma^*$ *is the value of the register* $\pi_i$*;*
- $n \in \mathbb{N}$ *is the index of the next instruction to be executed.*

*We denote the set of all configurations as $\mathcal{CR}_R$. If $n = 1$ we have an initial configuration for a $k$-tuple of strings $\mathbf{s}$, which is indicated with $\mathcal{INR}_R^{\mathbf{s}}$. If $n = m + 1$ (where $m$ is the largest index of an instruction in the program), we have a final configuration, called $\mathcal{FCR}_R^s$, where $s$ is the string stored in $\pi_1$.*

First we observe that the meaning of a PRM program $R$ can be defined by way of two functions $\delta_0$ and $\delta_1$: if the next instruction to be executed is $r_a$, then $\delta_0(C)$ is potentially different than $\delta_1(C)$, otherwise the two are equal. In other words, we can consider two functions $\delta_0 : \mathcal{CR}_R \to \mathcal{CR}_R$ and $\delta_1 : \mathcal{CR}_R \to \mathcal{CR}_R$ which, given a configuration in input:

- both produce in output the (unique) configuration resulting from the application of the next instruction, if different than $r_a$;
- produce the two configurations resulting from the two branches of the next instruction, if it is $r_a$.

Similarly to what we have done for PTMs (see Section 2), we can define a (complete) partial order with carrier $\mathcal{CEVR}$ (which is the set of all functions from $\mathcal{CR}_R$ to $\mathbb{P}_{\Sigma^*}$). And hence, we can define a functional $FR_R$ on $\mathcal{CEVR}$ which will be used to define the function computed by $R$ via a fixpoint construction. Intuitively, the application of the functional $FR_R$ describes *one* computation step. More formally:

**Definition 30** *Given a PRM $R$, we define a functional $FR_R : \mathcal{CEVR} \to \mathcal{CEVR}$ as:*

$$FR_R(f)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FCR}_M^s; \\ \frac{1}{2}f(\delta_0(C)) + \frac{1}{2}f(\delta_1(C)) & \text{otherwise.} \end{cases}$$

Using similar arguments to those in the proofs of Proposition 6 and Theorem 1, we can show that the least fixpoint of $FR_R$ actually exists. Such a least fixpoint, once composed with a function returning $\mathcal{INR}_R^{\mathbf{s}}$ from $\mathbf{s}$, is the *function computed by the register machine $R$* and is denoted by $\mathcal{IO}_R : \Sigma^* \to \mathbb{P}_{\Sigma^*}$. The next Lemma shows the relations between PTMs and PRMs.

**Lemma 7** *PTMs are linear time reducible to PRMs, and PRMs over $\mathbb{W}$ are polytime reducible to PTMs.*

**Proof.** A single tape PTM $M$ can be simulated by a PRM $R_M$ that has tree registers. A configuration $\langle w, a, v, s\rangle$ of $M$ can be coded by the configuration $\langle w^r, a, v, s\rangle$ where $w^r$ denotes the reverse of the string $w$. Each move of $M$ is

simulated by at most 2 moves of $R_M$. In order to simulate the probabilistic part given by the functions $\delta_0$ and $\delta_1$ we use the instructions $e$, $r_a$ and $j$, plus a dedicated register $\pi_{coin}$, in the natural way. Conversely, a PRM $R$ over $\mathbb{W}$ with $m$ registers is simulated by a PTM $M_R$ with $m$ tapes. Some moves of $R$ may require copying the contents of one register to another for which $M$ may need as many steps to complete as the maximum of the current lengths of the corresponding tapes. Thus if $R$ runs in time $O(n^k)$, then $M_R$ runs in time $O(n^{2k})$. We can then conclude by remembering that Turing machines with multiple tapes can be simulated by single-tape Turing machine with a polynomial slowdown. □

### 3.4 Polytime Soundness

In this section we prove that any function definable by predicative recurrence can be computed in polynomial time by a PTM. In view of Lemma 7, in order to obtain this result it suffices to show that predicative recurrence can be simulated by a probabilistic register machines working in polynomial time (dubbed PPRM in the following). This result is not difficult and is proved below by exhibiting a PPRM which computes any function $f$ such that $f \triangleright \mathbf{W} \to \mathbb{W}_m$. In the following we denote by $\mathscr{PPR}$ the class of functions computed by PPRMs.

The length $|v|$ of a string is simply the number of characters in it. Given a string distribution $\mathcal{D} \in \mathbb{P}_\Sigma$, its *length* $|\mathcal{D}|$ is simply the maximal length of strings in the support of $\mathcal{D}$. Moreover, if $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{W}^n$ and $\mathbf{W} = \mathbb{W}_{m_1} \times \ldots \times \mathbb{W}_{m_n}$, then $|\mathbf{v}|_k = \max_{m_i=k} |v_i|$. Analogously for $|v|_{<k}$ and $|v|_{>k}$. The following is again from [13]:

**Lemma 8 (Max-Plus)** *If $h \triangleright \mathbf{W} \to \mathbb{W}_m$, then there is a polynomial $q_h : \mathbb{N} \to \mathbb{N}$ such that for every $\mathbf{v}$, it holds that $|h(\mathbf{v})| \leq |\mathbf{v}|_m + q_h(|\mathbf{v}|_{>m})$*

**Proof.** This is an induction on the structure of a derivation for $h \triangleright \mathbf{W} \to \mathbb{W}_m$. □

**Proposition 11** *If $h \triangleright \mathbf{W} \to \mathbb{W}_m$, then there is a PPRM $R_h$ that computes $h$.*

**Proof.** The proof is by induction on the structure of a derivation for $h \triangleright \mathbf{W} \to \mathbb{W}_m$:

- We first of all need to show that for every basic function, we can construct a PPRM that computes such a function. The proof is immediate for the

functions $e$, $c_a$, and $r_a$, all of which can be easily computed by eponymous register machine instructions. The projections $\Pi^n_m(v)(w)$ can be simulated by the instruction $c_a$, followed by $p$.

- Assume that $h$ is defined by composition, namely that $h = f \odot (g_1, \ldots, g_n) : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$. We give an intuitive proof by exhibiting a PPRM, called $R_h$, which computes $h$ in polynomial time. $R_h$ operates by using $R_f, R_{g_1}, \ldots, R_{g_n}$ (all of which exist by induction hypothesis), as subroutines in the natural way. The fact that this process takes polynomial time is a consequence of the fact that the machines $R_f, R_{g_1}, \ldots, R_{g_n}$ are themselves polytime, and that the $R_f$ is called on inputs of polynomial length, itself a consequence of Lemma 8 above.
- Assume now that $h$ is defined by case distinction, namely that $h = \mathit{case}(g_\varepsilon, \{g_a\}_{a\in\Sigma})$. In this case the PPRM $R_h$ which computes $h$ can be defined as a machine which analyze one of its inputs, deciding based on its value (by way of instructions $j_\varepsilon$ and $j_a$) which ones among $R_{g_\varepsilon}, R_{g_a}$ (where $a \in \Sigma$) to call to analyze the rest of the input. Please notice that the machines above exist and work in polynomial time by the induction hypothesis.
- Finally, assume that $h$ is defined by primitive recursion, namely that $h = \mathit{rec}(g_\varepsilon, \{g_a\}_{a\in\Sigma},)$. In this case the PPRM $R_h$ which computes $h$ can be defined as a machine which iteratively calls as subroutines the machines $R_{g_\varepsilon}, R_{g_a}$ (where $a \in \Sigma$), which exist and work in polynomial time, based on the value of one its inputs. The machines above are clearly called a number of times linear in the size of one of the inputs, while the fact that each calls takes itself polynomial time is a consequence of Lemma 8.

This concludes the proof. □

### 3.5 Polytime Completeness

There is a relatively easy (although not elegant) way to prove polytime completeness of probabilistic ramified recurrence, namely going through the same result for deterministic ramified recurrence [13]. The argument goes as follows:

- First of all, it is easy to prove that for every $k$ and for every $n$ the function $f_{k,n}$ outputting a sequence of random bits of length $|s|^k + n$ (where $s$ is the input) is a ramified probabilistic function.
- Then, one can observe that for every polynomial time computable function $g : \Sigma^* \to \Sigma^*$, it holds that $p_g \triangleright \mathbb{W}_n \to \mathbb{W}_m$ for some $n$ and $m$, this as a

consequence of Leivant's result [13].
- Finally, one can observe that any polytime probabilistic function can be seen as a deterministic polytime function taking an additional input consisting of a "long enough" sequence of random bits.

Polytime completeness is an easy corollary of the three observations above. More precisely, we now present some lemmas that allow us to prove completeness.

**Lemma 9 (Polytime Random Sequences)** *For every $k$ and for every $n$, let $f_{k,n}$ be the probabilistic function outputting a sequence of random bits of length $|s|^k + n$ (where $s$ is the input). Then $f_{k,n} \triangleright \mathbb{W}_m \to \mathbb{W}_l$ holds for some natural numbers $m$ and $l$.*

**Proof.** Let $q$ be the deterministic function on $\mathbb{W}$ which outputs $0^{|s|^k+n}$, where $s$ is the input. Clearly, $q$ is computable in polynomial time. As a consequence, $p_q$ can be typed in Leivant's system. Let *randext* be the probabilistic function which, on input $s$, outputs either $0 \cdot s$ or $1 \cdot s$, each with probability $\frac{1}{2}$. *randext* can be typed with $\mathbb{W}_m \to \mathbb{W}_m$ for every $m$ (it can be defined from $r_0$ and $r_1$ and other base functions by case distinction). What we need to obtain $f_{k,n}$, then, is just to compose a function obtained by primitive recursion from *randext*, and $p_q$. □

The next Lemma is again due to Leivant [13].

**Lemma 10 (Polytime Functions and Predicative Functions)** *For every deterministic polynomial time computable function $g : \Sigma^* \to \Sigma^*$ it holds that $p_g \triangleright \mathbb{W}_n \to \mathbb{W}_m$ for some $n$ and $m$.*

Then we have the following.

**Theorem 4** $\mathscr{PPR} \subseteq \mathscr{PT}$.

**Proof.** Consider any probabilistic polytime Turing machine $M$. From the discussion at the beginning of this section, it is clear that the probabilistic function computed by $M$ is $p_f \odot p_{pair} \odot (id, f_{k,n})$. where $f$ is a polytime computable deterministic function, *pair* is a deterministic function encoding two strings into one, and $f_{k,n}$ is the function from Lemma 9. Since the three functions can be given a type, their composition itself can. □

We are finally ready to prove the main result of this section:

**Corollary 2** $\mathscr{PPR} = \mathscr{PT}$.

**Proof.** Immediate from Theorem 4 and Proposition 11. □

A more direct way to prove polytime completeness consists in showing how single-tape PTMs can be encoded into predicative recurrence. This can be done relatively easily by exploiting simultaneous recursion, but we leave this for future work.

# 4 Conclusions

In this paper, we make a first step in the direction of characterizing probabilistic computation in itself, from a recursion-theoretical perspective, without reducing it to deterministic computation. The significance of this study is genuinely foundational: working with probabilistic functions allows us to better understand the nature of probabilistic computation, but also to study the implicit complexity of a generalization of Leivant's predicative recurrence, all in a unified framework.

More specifically, we give a characterization of computable probabilistic functions by a natural generalization of Kleene's partial recursive functions. We then prove the equi-expressivity of the obtained algebra and the class of functions computed by PTMs. In the second part of the paper, we investigate the relations existing between our recursion-theoretical framework and subrecursive classes, in the spirit of ICC. More precisely, endowing predicative recurrence with a random base function is proved to lead to a characterization of polynomial-time computable probabilistic functions.

An interesting direction for future work could be the extension of our recursion-theoretic framework to *quantum* computation. In this case one should consider transformations on Hilbert spaces as the basic elements of the computation domain. The main difficulty towards obtaining a completeness result for the resulting algebra and proving the equivalence with quantum Turing machines seems to be the definition of suitable recursion and minimization operators, given that qubits (the quantum analogues of classical bits) cannot be copied nor erased.

## References

[1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992. doi:10.1007/BF01201998.

[2] Andrej Bogdanov and Luca Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1):1–106, 2006. doi:10.1561/0400000004.

[3] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):564–577, 2003. doi:10.1109/TPAMI.2003.1195991.

[4] Nigel J. Cutland. *Computability: An introduction to recursive function theory*. Cambridge University Press, 1980.

[5] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. In Ricardo Peña, Marko C. J. D. van Eekelen, and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis (FOPARA 2011)*, volume 7177 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011. doi:10.1007/978-3-642-32495-6_1.

[6] Ugo Dal Lago and Sara Zuppiroli. Probabilistic recursion theory and implicit computational complexity. In Gabriel Ciobanu and Dominique Méry, editors, *Proceedings of the 11th International Colloquium on Theoretical Aspects of Computing (ICTAC 2014)*, volume 8687 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2014. doi:10.1007/978-3-319-10882-7_7.

[7] Karel De Leeuw, Edward F Moore, Claude E Shannon, and Norman Shapiro. Computability by probabilistic machines. *Automata studies*, 34:183–198, 1956. doi:10.1109/9780470544242.ch54.

[8] John Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977. doi:10.1137/0206049.

[9] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998. doi:10.1006/inco.1998.2700.

[10] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984. doi:10.1016/0022-0000(84)90070-9.

[11] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[12] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936. doi:10.1007/bf01565439.

[13] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and JeffreyB. Remmel, editors, *Feasible Mathematics II*, Progress in Computer Science and Applied Logic, pages 320–343. Springer, 1995. doi:10.1007/978-1-4612-2566-9_11.

[14] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1-2):167–184, 1993. doi:10.1007/bfb0037112.

[15] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[16] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[17] Michael O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963. doi:10.1016/s0019-9958(63)90290-0.

[18] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. doi:10.1147/rd.32.0114.

[19] Eugene S. Santos. Probabilistic Turing machines and computability. *Proceedings of the American Mathematical Society*, 22(3):704–710, 1969. doi:10.1090/s0002-9939-1969-0249221-4.

[20] Eugene S. Santos. Computability by probabilistic Turing machines. *Transactions of the American Mathematical Society*, 159:165–184, 1971. doi:10.2307/1996005.

[21] Robert I. Soare. *Recursively enumerable sets and degrees: a study of computable functions and computably generated sets*. Perspectives in mathematical logic. Springer, 1987.

[22] Sebastian Thrun. Robotic mapping: A survey. In Gerhard Lakemeyer and Bernhard Nebel, editors, *Exploring Artificial Intelligence in the New Millennium*, pages 1–35. Morgan Kaufmann Publishers Inc., 2003.