Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXV

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF/01

Probabilistic Recursion Theory and Implicit Computational Complexity

Presentata da: Sara Zuppiroli

Coordinatore Dottorato:	Relatore:
Maurizio Gabbrielli	Ugo Dal Lago

Esame finale anno 2014

Abstract

In this thesis we provide a characterization of probabilistic computation in itself, from a recursion-theoretical perspective, without reducing it to deterministic computation. More specifically, we show that probabilistic computable functions, i.e., those functions which are computed by Probabilistic Turing Machines (PTM), can be characterized by a natural generalization of Kleene's partial recursive functions which includes, among initial functions, one that returns identity or successor with probability $\frac{1}{2}$. We then prove the equi-expressivity of the obtained algebra and the class of functions computed by PTMs. In the the second part of the thesis we investigate the relations existing between our recursion-theoretical framework and sub-recursive classes, in the spirit of Implicit Computational Complexity. More precisely, endowing predicative recurrence with a random base function is proved to lead to a characterization of polynomial-time computable probabilistic functions.

Acknowledgements

I would never have been able to finish my dissertation without the guidance of my supervisor, the support of the PhD's director, the encouragement of my colleagues, the help of my friends and the guide of my family. I would like to express my deepest gratitude to my supervisor, Dr. Ugo Dal Lago, for his guidance and for teaching me an excellent research method which I try to learn. I would like to thank Prof. Maurizio Gabbrielli, who guided me towards this result. I would also like to thank Prof. Paolo Ciancarini and Prof. Simone Martini for their support in these years.

Thanks to all my friends which were always willing to help and to give useful suggestions. In particular many thanks to Giulio Pellitta, Tudor A. Lascu, Alessandro Rioli, and all the other colleagues in the laboratory. Thanks to Marco Di Felice, Diego Ceccarelli, Ioana Cristescu and Imane Sefrioui, persons that I met in these years and who became friends more than colleagues. Thanks to Cristina Ceroni, Sarah Draush, Francesca Druidi, Chiara Evangelisti, and Simona Minarini, friends who have seen my difficulties and were able to help me. I would like to thank my parents and my brother for their guide. Finally I would like to thank my friend Maurizio who was the only person who really listened to my emotions and who received all my confidences.

Contents

Chapter 1

Introduction

This dissertation introduces a probabilistic recursion theory which tries to capture the very essence of the functions which are computed by a Probabilistic Turing Machine. Our basic idea is that these functions can be characterized directly, without *reducing* them to deterministic functions, by considering as domain a discrete set and as codomain a set of *distributions* over the domain.

Before going into some more details, it is useful to recall here some basic facts about the history of computability which led to probabilistic models of computation. 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 of functions (on, say, natural numbers) they are able to compute are the same.

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 Nondeterministic Finite Automata [29] 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 certain probability. Again, probabilistically evolving computation can be a way to abstract over determinism, but also a way to model situations in which algorithms have access to a source of true randomness.

Probabilistic models are nowadays very common: 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., secure public key encryption schemes need to be probabilistic [17]). A nice way to deal computationally with probabilistic models is to allow probabilistic choice as a primitive when designing algorithms, this way switching from usual, deterministic computation to a new paradigm, called probabilistic computation. Examples of application areas in which probabilistic computation has proved to be useful include natural language processing [24], robotics [34], computer vision [6], and machine learning [27].

But what does the presence of probabilistic choice give us in terms of expressivity? Is it that we increase the power of deterministic computation? And what about efficiency: is it that probabilistic choice permits to solve any computational problem more efficiently? These questions have been among the most central in the theory of computation, and in particular in computational complexity, in the last forty years. Starting from the early fifties, various forms of automata in which probabilistic choice is available have been considered (e.g. [28]). The inception of probabilistic choice into an universal model of computation, namely Turing machines, is due to Santos [31, 32], but is (essentially) already there in an earlier work by De Leeuw and others [12]. Some years later, Gill [13] considered probabilistic Turing machines with bounded complexity: his work has actually 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 [19], in which algorithms (e.g. encryption schemes, authentication schemes, and adversaries for them) are very often assumed to work in probabilistic polynomial time. Recently, some investigations on the interplay between implicit com-

plexity and probabilistic computation have also started to appear [9]. There is however an intrinsic difficulty in giving *implicit* characterizations of probabilistic classes like $\mathbf{BPP}$ or $\mathbf{ZPP}$ which are semantic classes defined by imposing a polynomial bound on time, but also appropriate bounds on the probability of error. These bounds makes the task of enumerating machines computing problems in the classes above much harder and, ultimately, prevents from deriving implicit characterization of the classes themselves. Roughly, all these results can be summarized saying that while probability has been proved not to offer any advantage in the absence of resource constraints, it is not known whether probabilistic classes such as $\mathbf{BPP}$ or $\mathbf{ZPP}$ are different from $\mathbf{P}$. It is worth noticing that all these works follow an approach that we call *reductionist*: probabilistic computation is studied by *reducing* or *comparing* it to deterministic computation.

This thesis goes in a somehow different direction: as mentioned before, we want to study probabilistic computation directly, without necessarily *reducing* it to deterministic computation. In our prospective, 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.

1.1 Contributions

The main contributions of this dissertation are briefly summarized as follows.

- The first one is rooted in classic theory of computation, and in particular in the definition of partial computable functions as introduced by Church and later studied by Kleene [20]: 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. 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 useless. We later show how to recover the latter by replacing minimization with a more powerful operator. We also discuss how probabilistic recursion theory offers characterizations of concepts like the one of a computable distribution and of a computable real number.

- The second main contribution of this dissertation consists in the application of the aforementioned recursion-theoretical framework to polynomial-time computation. We do that by following Bellantoni and Cook's and Leivant's works [2, 21], 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, we to provide a characterization of polynomial-time computable distributions, a key notion in average-case complexity [3].

Our results provide a first step in the direction of characterizing probabilistic computation in itself, from a recursion-theoretical perspective. The significance for this study is genuinely foundational, since it allows us to better understand the nature of probabilistic computation and also to investigate the implicit complexity of a generalization of Leivant's predicative recurrence, all in a unified framework. In a future perspective our study could provide the basis to define a recursion theory for quantum computing. The results contained in this thesis have been published in [10, 11].

1.2 Thesis outline

The remaining chapters of this thesis are organized as follows.

Next chapter contains the basic notions of classic recursion theory. In particular, we recall the definition of partial recursive functions and the equivalence of this formalism with functions computable by Turing Machines. This equivalence proof will provide an intuitive idea for the main proof in the first contribution. Chapter 3 contains a short introduction to implicit computational complexity (ICC). Chapter 4 contains the preliminaries about probability theory and probabilistic Turing machines (PTM), in particular we give

an operational definition of functions computed by a PTM and we discuss some *reductionist* approaches. In Chapter 5 we provide the first main contribution of this thesis by defining probabilistic recursive functions and proving their equivalence with the functions computed by a PTM. Chapter 6 provides our second contribution: we provide a characterization of the probabilistic functions which can be computed in polynomial time by using an algebra of functions acting on word algebras. Finally Chapter 7 concludes.

Chapter 2

Classic Recursion Theory

In this chapter we introduce some basic notions concerning recursion theory with the aim of recalling some well known, classic results and also to define some notation that we will use in the following. We mainly extract these notions from [7, 30, 20]. We start with the basic definitions and methods developed in the context of Recursive Function Theory (1936-38), by Turing and Kleene [20], and we terminate with the Church Turing Thesis. Recursion Theory, (also called Computability Theory) is the study of the functions which are "computable" with the aim also of characterizing their computational complexity. Informally, a computable function is a function which can be represented by an algorithm. The algorithm is a finite set of instructions which, given an input x, after a finite number of steps yields an output y. The algorithm specifies how to obtain each step in the calculation from the previous steps, starting from the input x. So the algorithm computes a function f which, given the input x, produces as output $f(x) = y$. For some inputs an algorithm could not terminate; in this case we say that the algorithm computes a *partial* function. On the other hand, if for all inputs the algorithm terminates we say that it computes a *total* function.

Now we give two precise mathematical formulations of computable functions : the Turing computable functions, and the partial recursive functions. In Section 2.4 we present a sketch of the proof of the equivalence of these two formulations. It is worth noting that here are several other (equivalent) formalizations of these classes, but for they are not needed for the present treatment and therefore they are not considered here.

We start with some basic notions. Here and in the following we indicate the concatenation of the strings v and w by $v \cdot w$.

Definition 2.1 (Σ^*) *We define the set $\Sigma_0 = \{\epsilon\}$ where ϵ is the empty string. An alphabet is a finite set of symbols. Such a set is denoted by Σ. Let Σ be an alphabet and Σ_0 be defined as above. We define inductively*

$$\Sigma_n = \{w | w = a \cdot w' \text{ where } w' \in \Sigma_{n-1} \text{ and } a \in \Sigma\}.$$

Then we define Σ^* as: $\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma_i$. Moreover we define Σ^+ as: $\Sigma^+ = \bigcup_{i \in \mathbb{N}^+} \Sigma_i$.

2.1 Register Machines

Now we define the classical Register machines which will be needed later when we will describe probabilistic Register machines and predicative functions.

Definition 2.2 (Register Machine) *A* register machine (RM) *consists of a finite set of registers $\Pi = \{\pi_1, \cdots, \pi_r\}$ and a sequence of instructions, called* program. *Each register π_i can store a string and each instruction in the program is indexed by a natural number and takes one of the following five forms*

$$\varepsilon(\pi_d); \qquad c_a(\pi_s)(\pi_d); \qquad p_a(\pi_s)(\pi_d); \qquad j(\pi_s)(m);$$

where π_s, π_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 $\varepsilon(\pi_d)$ stores in the register π_d the empty string and then transfer the control to the next instruction.
- The instruction $c_a(\pi_s)(\pi_d)$ stores in the register π_d the term $a \cdot w$, where w is the string contained in the register π_s. It then transfers the control to the next instruction.
- The instruction $p_a(\pi_s)(\pi_d)$ is the predecessor instruction, which stores in the register π_d the string resulting from erasing the leftmost character a from the string contained in π_s, if any. The control is then transferred to the next instruction.

– The instruction $j(\pi_s)(m)$ transfers the instruction to the m-th instruction if π_s contains a non-empty string, while it goes to the next instruction otherwise.

We can now give a formal definition of configurations.

Definition 2.3 (Configuration of a RM) *Let R be a RM as in Definition 2.2, and let Σ be the underlying alphabet. We define a RM* configuration *as a tuple $\langle v_1, \ldots, v_r, n\rangle$ where:*

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

We denote the set of all configurations with $\mathcal{CCR}_R$. If $n = 1$ we have an initial configuration and we denote it by $\mathcal{ICR}_R^{\mathbf{s}}$. If $n = m + 1$, where m is the largest index of an instruction in the program, we have a final configuration, denoted by $\mathcal{FCCR}_R^s$, where s is the string stored in π_1.

Next we show how previous instructions allow to change a configuration.

$\epsilon(\pi_s)(\pi_l)$ If we apply the instruction $\epsilon(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{l-1}, v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$c_a(\pi_s)(\pi_l)$ If we apply the instruction $c_a(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{l-1}, a \cdot v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$p_a(\pi_s)(\pi_l)$ If we apply the instruction $p_a(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, a{\cdot}v_s, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, a \cdot v_s, \ldots, v_{l-1}, v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$j(\pi_s)(m)$ If we apply $j(\pi_s)(m)$ to the configuration $\langle v_1, \ldots, a \cdot v_s, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_s, \ldots, v_r, m\rangle$.

If we apply $j(\pi_s)(m)$ to the configuration $\langle v_1, \ldots, v_{s-1}, \epsilon, v_{s+1}, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{s-1}, \epsilon, v_{s+1}, \ldots, v_r, n+1\rangle$.

2.2 Classical Turing Machines

Now we define the classical Turing machines and the class of functions computed by them. We will need the notion of classical Turing machine when we introduce later probabilistic Turing machines.

Definition 2.4 (Classical Turing Machine) *A classical Turing machine (CTM) is a device which performs operations on a tape infinite in both directions. A CTM is defined as a tuple $\langle Q, \Sigma_b, O, \delta \rangle$ where*

- $Q = \{q_s, q_1, \cdots, q_n\}$ *is a finite set that identify the states of the CTM. We assume that Q contains an initial state q_s and there exists a finite subset $Q_f = \{q_{f1}, \cdots, q_{fk}\} \subset Q$ of final states.*

- $\Sigma_b = \Sigma \cup \{\square\}$*, where Σ is the alphabet, and $\square$ is the blank symbol;*

- $O = \{\rightarrow, \leftarrow\} \cup \Sigma_b$ *is a finite set that identify the possible operations;*

- $\delta : \Sigma_b \times Q \setminus Q_f \rightarrow O \times Q$ *is a transition function which has the following meaning. If*

  $$\delta(a, q) = (\rightarrow, q')$$

  then the head moves to the right on the tape and the machine switches to the state q'. If

  $$\delta(a, q) = (\leftarrow, q')$$

  then the head moves to the left on the tape and the machine switches to the state q'. If

  $$\delta(a, q) = (a, q')$$

  with $a \in \Sigma_b$ then the head does not move, the machine writes a on the tape and the machine switches to the state q'.

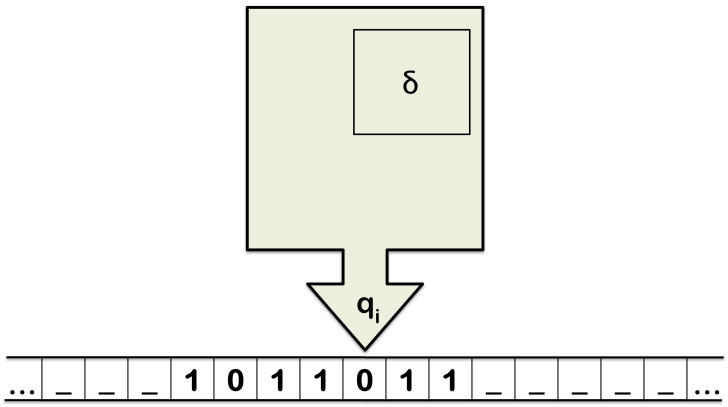

Figure 2.1: Turing Machine

The function δ may be undefined for some arguments and it can be seen as a *program* (considered as a finite set of 4-tuples). The Turing machine can be represented as in Figure 2.2.

The "picture" of each Turing machine after n steps of computation can be represented by using the notion of configuration. We have tree types of configurations:

- A *General Configuration* is any configuration of a machine CM;
- An *Initial Configuration* is the configuration which contain the initial state of CM;
- A *Final Configuration* is a configurations which contains a final state of CM.

Definition 2.5 (General Configuration of a CTM) *Let CM be a CTM as in Definition 2.4. We define a CTM configuration as a 4-tuple $\langle \mathbf{s}, a, \mathbf{t}, q \rangle \in (\Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q)$ such that:*

- *The first component, $\mathbf{s} \in \Sigma_b^*$, 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, $\mathbf{t} \in \Sigma_b^*$, 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 configuration as $\mathcal{CS}_{CM}$.

Definition 2.6 (Initial and Final Configurations of CTM) *Let CM be a CTM as in Definition 2.4. We define an initial configuration of CM for the string s as a configuration in the form $\langle \epsilon, a, \mathbf{v}, q_s \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that: $\mathbf{s} = a \cdot \mathbf{v}$ and $q_s \in Q$ is the initial state. We denote by $\mathcal{IN}_{CM}^s$ the set of all such initial configurations. Similarly, we define a final configuration of CM as a configuration $\langle \mathbf{s}, \square, \epsilon, q_f \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q_f$. The set of all final configurations for a CTM CM is denoted by $\mathcal{FC}_{CM}^s$*

Intuitively, the function computed by a CTM CM associates to each input s, a string which indicates the output present in the tape when reaching a final configuration of CM from $\mathcal{IN}_{CM}^s$. In fact, a *Turing computation* for a CTM specified by the program δ, with input s, is a sequence of configurations $cs_1, \cdots, cs_k$ such that $cs_1 \in \mathcal{IN}_{CM}^s$ and each configuration cs_i with $i \geq 2$ is obtained from the previous one by means of the application of a transition specified by δ.

More precisely, the operational semantics of a CTM, and therefore the notion of computed function, can be defined as follows.

Definition 2.7 (Transition) *Let M be a CTM as defined in Definition 2.4, and let CS be a set as defined in Definition 2.5. We define a function $\vdash$: $CS \to CS$ as follows:*

$$\vdash (\langle \mathbf{u}, c, \mathbf{t}, q_n \rangle) = \begin{cases} \langle \mathbf{u} \cdot c, b, \mathbf{r}, q_m \rangle & \text{if } \delta(q_n, c) = (q_m, \rightarrow) \ , \ \mathbf{t} = b \cdot \mathbf{r} \ , \ b \cdot \mathbf{r} \in \Sigma_b^* \\ \langle \mathbf{w}, a, c \cdot \mathbf{t}, q_m \rangle & \text{if } \delta(q_n, c) = (q_m, \leftarrow) \ , \ \mathbf{u} = \mathbf{w} \cdot a \ , \ \mathbf{w} \cdot a \in \Sigma_b^* \\ \langle \mathbf{u} \cdot c, \square, \mathbf{t}, q_m \rangle & \text{if } \delta(q_n, c) = (q_m, \rightarrow) \ , \ \mathbf{t} = \epsilon \\ \langle \mathbf{u}, \square, c \cdot \mathbf{t}, q_m \rangle & \text{if } \delta(q_n, c) = (q_m, \leftarrow) \ , \ \mathbf{u} = \epsilon \\ \langle \mathbf{u}, d, \mathbf{t}, q_m \rangle & \text{if } \delta(q_n, c) = (q_m, d) \ , \ d \in \Sigma_b^* \end{cases}$$

In the following $\vdash^*$ denotes the transitive closure of function $\vdash$.

Definition 2.8 (Turing computable function) *Let CM be a CTM defined as in Definition 2.4. The function $f : CS \rightharpoonup CS$ computed by CM is defined as follows*

$$f(cs_i) = \begin{cases} cs_{fi} & \text{if } \vdash^* (cs_i) = cs_{fi} \text{ and } cs_{fi} \in \mathcal{FC}_{CM}^s \\ undefined & \text{otherwise} \end{cases}$$

We can also define the function computed by a $CTM\ CM$ through a fixpoint construction. Following [25] we remind here briefly the basic mathematical notions needed for the fix-point construction.

Definition 2.9 (Reflexive) *Let D be a set with a relation $\sim$, we say that $\sim$ is reflexive if for all $a \in D$ we have that $a \sim a$.*

Definition 2.10 (Transitive) *Let D be a set with a relation $\sim$, we say that $\sim$ is transitive if for all $a, b, \in D$ we have that $a \sim b\ and\ b \sim c\ implies\ a \sim c$.*

Definition 2.11 (Antisymmetric) *Let D a set with an operation $\sim$, we say that $\sim$ is Antisymmetric if for all $a, b \in D$ we have that $a \sim b\ and\ b \sim a$ implies $a = b$.*

A partial order is a relation over a set D which is reflexive, antisymmetric and transitive. We call POSET (partially ordered set) a set with a partial order relation.

The least upper bound or supremum (sup) of a subset S of a partially ordered set D is the least element of D that is greater than or equal to all the elements of S. A POSET D is *complete* (abbreviated CPO) if every directed set C has a least upper bound $\bigsqcup C$ in D[1] An ω Complete Partial Order (ωCPO) is a POSET in which every *chain* C (i.e. evry linearly ordered subset $x_1 \leq x_2 \leq x_3 \leq x_4 \leq \cdots$) has a least upper bound. These two definitions can be shown to be equivalent for denumerable POSETs.

A mapping $f : D \to E$ from one POSET D to a POSET E is *continuous*, if for every chain $C \subset D\ f(\bigsqcup C) = \bigsqcup f[C]$ holds (where $f[C]$ denotes the set $\{y\ s.t.\ y = f(x)\ and\ x \in C\}$. Next we have the following well known result where we define $\tau^{n+1}(x) = \tau(\tau^n(x))$ and $\perp$ denotes the bottom element.

Theorem 2.1 (Least-Fixed-Point) *Every continuous function $\tau : D \to D$ on a ωCPO has a least fixed point $\bigsqcup\ \tau^n(\perp)$.*

This result is the basic tool used by [33] in his *denotational semantics* of programming languages and since then has been used in many different contexts. In our case, we can use this result to define a fix point semantic for a CTM as follows. First we associate to each CTM a functional F_{CM} as follows:

[1].

Definition 2.12 *Given a CTM CM, we define a functional $F_{CM} : (CS \rightharpoonup CS) \to (CS \rightharpoonup CS)$ as:*

$$F_{CM}(f)(C) = \begin{cases} id(C) & \text{if } C \in \mathcal{FC}^s_{CM}; \\ f(\vdash (C)) & \text{otherwise}. \end{cases}$$

One can show that $CS \rightharpoonup CS$ is a POSET (by extending in the natural way to functions the flat ordering on CS). Then we have the following proposition.

Proposition 2.1 *The functional F_{CM} is continuos $CS \rightharpoonup CS$.*

Hence we have the following.

Corollary 2.1 *The functional defined in 2.12 has a least fix point which is equal to*

$$\bigsqcup_{n \geq 0} F^n_{CM}(\bot).$$

Proof: Immediate from Theorem 2.1. □

Such a least fixpoint is, once composed with a function returning $\mathcal{IN}^s_M$ from s, the *function computed by the machine CM*. This is denoted by $\mathcal{IO}_{CM} : CS \rightharpoonup CS$ while the set of all functions which can be computed by any CTMs is denoted by $\mathscr{CC}$.

2.3 Partial Recursive Functions

It is well known that the (partial) functions that can be computed by a Turing machine, or Turing computable functions, can be characterized in terms of a function algebra due to Kleene [20]. Such an algebra defines the partial recursive functions, which consists of the smallest class of functions containing the three basic functions defining zero (or constant), successor and projection, and then closed under composition (also called substitution), recursion and minimization (also called unbounded search). Below we give the formal definitions following [7].

Definition 2.13 (Basic Functions) *The Basic Functions are the following:*

- *The Zero function* $z : \mathbb{N} \to \mathbb{N}$ *is defined as:* $z(n) = 0$ *for every* $n \in \mathbb{N}$.
- *The Sucessor function* $s : \mathbb{N} \to \mathbb{N}$ *is defined as:* $s(n) = n + 1$ *for every* $n \in \mathbb{N}$.
- *The Projection function* ${\Pi^n}_m : \mathbb{N}^n \to \mathbb{N}$ *is defined as:* ${\Pi^n}_m(\mathbf{k}) = k_m$ *for every positive* $n \in \mathbb{N}$ *and for all* $m \in \mathbb{N}$*, such that* $1 \leq m \leq n$*, where* $\mathbf{k} = (k_1, \cdots k_n)$.

Definition 2.14 (Composition) *The function* $h : \mathbb{N}^k \to \mathbb{N}$ *is defined by composition if it is defined as:*

$$h(\mathbf{i}) = f(g_1(\mathbf{i}), \cdots, g_n(\mathbf{i}))$$

where $f : \mathbb{N}^n \to \mathbb{N}$ *and* $g_m : \mathbb{N}^k \to \mathbb{N}$ *for every* $1 \leq m \leq n$ *are partial recursive functions.*

Definition 2.15 (Recursion) *The function* $h : \mathbb{N}^{k+1} \to \mathbb{N}$ *is defined by recursion if it is defined as:*

$$h(\mathbf{i}, 0) = f(\mathbf{i})$$
$$h(\mathbf{i}, m + 1) = g(\mathbf{i}, m, h(\mathbf{i}, m))$$

where $\mathbf{i} \in \mathbb{N}^k$*,* $g : \mathbb{N}^{k+2} \to \mathbb{N}$ *and* $f : \mathbb{N}^k \to \mathbb{N}$ *are partial recursive functions.*

Definition 2.16 (Minimization) *The function* $g : \mathbb{N}^k \to \mathbb{N}$ *is defined by minimization if it is defined as*

$$g(\mathbf{i}) = \mu\, y(f(\mathbf{i}, y) = 0)$$

where $\mu\, y : \mathbb{N}^{k+1} \to \mathbb{N}$ *is defined as*

$$\mu\, y(f(\mathbf{i}, y) = 0) = \begin{cases} y & \text{if exist } y \text{ such that } f(\mathbf{i}, y) = 0\,, \text{ and } f(\mathbf{i}, h) \text{ is defined} \\ & \text{and } f(\mathbf{i}, h) \neq 0 \text{ for all } 0 \leq h < y \\ undefined & \text{otherwise} \end{cases}$$

Definition 2.17 (Partial Recursive Functions) *The class* $\mathscr{R}$ *of partial recursive functions is the smallest class of partial functions that contains the basic functions as defined in Definition 2.13 and is closed under the operation of Composition 2.14, Recursion 2.15 and Minimization 2.16.*

Many common functions are partial recursive: below we give some examples which will be useful when constructing the proof of the inclusion $\mathscr{CC} \subseteq \mathscr{R}$.

Example 2.1 *T*he following are examples of recursive functions:

- The *identity function* $id : \mathbb{N} \to \mathbb{N}$, defined as $id(x) = x$. In fact, since for all $x \in \mathbb{N}$ we have that

$$id(x) = x$$

  we have that $id = \Pi^1_1$, and, since the latter in a basic function (Definition 2.13) id is in $\mathscr{R}$.

- The function $add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for every $x, y \in \mathbb{N}$, $add(x, y) = x+y$ can be easily shown to be recursive. In fact the function add is defined by $add(x, 0) = id(x)$ and $add(x_1, x_2 + 1) = g(x_1, x_2, add(x_1, x_2))$. Now we have $add(x, 0) = id(x)$ and we know that id is in $\mathscr{R}$. Moreover we know that g in $\mathscr{R}$, because $g = s(\Pi^3_3)$. Hence add is a recursive function, since it can be obtained from basic functions using composition and primitive recursion.

  Wee can see that also the function $add_n : \mathbb{N}^n \to \mathbb{N}$, where $n \geq 3$, defined as $add_n(x_1, \cdots, x_n) = x_1 + \cdots + x_n$, is partial recursive. We observe that

$$\begin{aligned} add_n(x_1, \cdots, x_n) = \quad & add(\Pi^n_1(x_1, \cdots, x_n), add(\cdots \\ & (add(\Pi^n_{n-1}(x_1, \cdots, x_n), \Pi^n_n(x_1, \cdots, x_n))) \cdots)). \end{aligned}$$

  Hence $add_n \in \mathscr{R}$ because we apply composition operation to add and to the basic function Π^n_k.

- The function $rap_{CS} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined as $rap_{CS}(a_1, a_2, a_3, a_4) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + 2^{a_1+a_2+a_3+a_4+3} - 1$ can be easily shown to be recursive, since:

$$\begin{aligned} rap_{CS}(x_1, x_2, x_3, x_4) = \quad & pred(add_4(pow(2, x_1), pow(2, add_4(x_1, x_2, 1)), \\ & pow(2, add_4(x_1, x_2, x_3, 2)), pow(2, add_4(x_1, x_2, x_3, x_4, 3)))) \end{aligned}$$

  So rap_{CS} is the composition of the functions add_4, power, multiplication, and predecessor. These functions are partial recursive functions hence also rap_{CS} is a partial recursive function.

□

Next we use the function rap_{CS} to encode an entire element of $\mathbb{N}^k$. This encoding is based on the unique representation of each natural number in terms of a binary number.

This encoding is not the only possible one. For instance, a very famous encode is the Gödel one which can be defined as follows. Given a sequence $(x_1, x_2, x_3, ..., x_n)$ of positive integers, the encoding of the sequence is number (called Gödel number) obtained as the product of the first n primes power the value of the n-th number in the sequence, that is, $\mathrm{enc}(x_1, x_2, x_3, \ldots, x_n) = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdots p_n^{x_n}$. Since a fundamental theorem of arithmetic ensure us that any number can be uniquely factored into prime factors, it is possible to recover the original sequence from its Gödel number (for any given number n of symbols to be encoded).

2.4 Equivalence Result

We sketch now the proof of the equivalence between Turing computable functions and partial recursive functions. This proof will be the basis on which we will build our main equivalence result in the next chapters.

The proof consists of two parts. The first one is easier, and consists in implementing each basic recursive functions, the composition, the recursion and the minimization by means of a suitable Turing machine.

For the other part of the proof, given a Turing machine CM we need to define some partial recursive functions which allow to simulate CM. This can be done in several steps. First one shows how each configuration can be encoded as a natural number and then represented by means of composition of recursive functions. Next, one has to show that the transition function of a Turing machine can be simulated by a recursive function that, given a natural number representing the configuration, decomposes it into its four components, and depending on the state and the character on the head, compute the (natural corresponding to the) new configuration. This can be done because all the needed arithmetic functions division are partial recursive. Finally, one has to specify that the transition function has to be repeated until it reaches a final configuration. For this we can use the minimization function: assuming that SR_{CM} is the function which corresponds

to the transition function on configurations, by using the minimization we can compute the minimum number of times p that SR_{CM} has to be performed in order to reach a final configuration.

Below we give some more details on this equivalence proof. As previously mentioned, we first need to see each tape of the Turing machine as a natural number. In order to simply the notation, in the follow we assume without loss of generality $\Sigma_b = \{0, \cdots, k\}$, where 0 represents the blank symbol.

The notation $|\cdot|$ is used to represent the length of the element $\cdot$. More precisely, if $\cdot$ is a string $a \in \Sigma^*$ then $|a|$ represents the length of the string, while if $\cdot$ is an element $\mathbf{u} \in \mathbb{N}^k$ then $|\mathbf{u}|$ is k.

Definition 2.18 (Turing tape as a natural number) *Let* $\mathbf{u}$ *be the right (part of) the tape and* $\mathbf{t}$ *be the left tape in a* CTM *as defined in Definition 2.4. We can see* $\mathbf{u}, \mathbf{t}$ *as the representations of two natural numbers, in base* k*, by using the functions* $rep_r : \Sigma_b^* \to \mathbb{N}$*, and* $rep_l : \Sigma_b^* \to \mathbb{N}$ *defined as follows:*

$$n = rep_r(\mathbf{u}) = \sum_{i=0}^{|\mathbf{u}|} b_i * k^i$$

$$m = rep_l(\mathbf{t}) = \sum_{i=0}^{|\mathbf{t}|} c_i * k^i$$

where $\mathbf{u} = b_0 b_1 b_2 \ldots$ *and* $\mathbf{t} = \ldots c_2 c_1 c_0$ *(so the less representative digit is that one closer to the head in both cases).*

Note that in the previous definition n and m are finite numbers because $\mathbf{u}$ and $\mathbf{t}$ have only a finite number of digits different from 0. Using previous definition we can see the function computed by a Turing machine as a function on natural numbers:

Definition 2.19 (Turing computable function from $\mathbb{N}$ to $\mathbb{N}$) *Let* CM *be a* CTM *as defined in Definition 2.4. The function* $fn : \mathbb{N} \to \mathbb{N}$ *computed by* CM *is defined as:*

$$fn(n) = \begin{cases} m & \text{if } f(\mathbf{v}) = \mathbf{u} \text{ and } rep_r(\mathbf{v}) = n \text{ and } rep_l(\mathbf{u}) = m \\ undefined & \text{if } f(\mathbf{v}) \text{ is undefined} \end{cases}$$

where f is the function defined in 2.8, rep_r and rep_l are the functions defined in Definition 2.18.

Along these lines we can also see any configuration as a natural number. Indeed, considering Definition 2.5, we can assume that $\Sigma_b = \{0, 1, 2, \cdots, k\}$, $Q = \{1, 2, \cdots, q_n\}$, and $v \in \Sigma_b^*$ represents a natural number as shown above. Finally we can assume that $O = \{k+1, k+2\} \bigcup \Sigma_b$. Then in order to see a configuration as a natural number we need to establish a bijection between $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. This can be done by defining $rap_{CS} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as follows:

$$rap_{CS}(\mathbf{a}) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + 2^{a_1+a_2+a_3+a_4+3} - 1.$$

The function defined above is a bijection because each natural number has a unique expression as binary decimal, so given $x \in \mathbb{N}$ we find $k = 4$ unique numbers such that $x + 1 = 2^{b_1} + 2^{b_2} + 2^{b_3} + 2^{b_4}$.

We have seen in the previous section that above defined representation function is partially recursive. Also the inverse of rap_{CS} is partially recursive, in fact it can be obtained as:

$$rap_{CS}^{-1}(x) \quad = \langle rap_{CS1}^{-1}(x), rap_{CS2}^{-1}(x), rap_{CS3}^{-1}(x), rap_{CS4}^{-1}(x) \rangle$$

where the functions $rap_{CS1}^{-1} : \mathbb{N} \to \mathbb{N}$, $rap_{CS2}^{-1} : \mathbb{N} \to \mathbb{N}$, $rap_{CS3}^{-1} : \mathbb{N} \to \mathbb{N}$, $rap_{CS4}^{-1} : \mathbb{N} \to \mathbb{N}$ are defined as follows:

$$rap_{CS1}^{-1}(x) = \log_2(x+1)$$

$$rap_{CS2}^{-1}(x) = \log_2(x + 1 - 2^{rap_{CS1}^{-1}(x)} - 2^1) - 1$$

$$rap_{CS3}^{-1}(x) = \log_2(x + 1 - 2^{rap_{CS1}^{-1}(x)} - 2^{rap_{CS2}^{-1}(x)} - 2^1) - 1$$

$$rap_{CS4}^{-1}(x) = \log_2(x + 1 - 2^{rap_{CS1}^{-1}(x)} - 2^{rap_{CS2}^{-1}(x)} - 2^{rap_{CS3}^{-1}(x)} - 1) - 1$$

and all these functions can be shown to be partial recursive.

Next, we can see the transition function of a Turing machines as a function operating on 4-tuples of natural numbers, given the characterization of configurations shown above.

Definition 2.20 (Transition from $\mathbb{N}^4$ to $\mathbb{N}^4$) *Let CM be a CTM as defined in Definition 2.4, and let CS be as defined in Definition 2.5. We define a function $\vdash_N : \mathbb{N}^4 \to \mathbb{N}^4$ as:*

$$\vdash_N (n, c_a, m, q_{q_r}) = \begin{cases} (n * k + c_a, c_b, y, q_{q_t}) & \text{if } \delta(q_{q_r}, c_a) = (q_{q_t}, k+1),\ m = c_b + y * k \\ (x, c_b, c_a + m * k, q_{q_t}) & \text{if } \delta(q_{q_r}, c_a) = (q_{q_t}, k+2)\ ,\ n = x * k + c_b \\ (n * k + c_a, 0, m, q_{q_t}) & \text{if } \delta(q_{q_r}, c_a) = (q_{q_t}, k+1)\ ,\ m = 0 \\ (n, 0, c_a + m * k, q_{q_t}) & \text{if } \delta(q_{q_r}, c_a) = (q_{q_t}, k+2)\ ,\ n = 0 \\ (n, c_b, m, q_{q_t}) & \text{if } \delta(q_{q_r}, c_a) = (q_{q_t}, c_b)\ ,\ c_b \in \mathbb{N} \end{cases}$$

Using this definition and the previous bijection among $\mathbb{N}^4$ and $\mathbb{N}$ we can see a transition of a CTM as a function from $\mathbb{N}$ to $\mathbb{N}$. We omit its definition and, by a slight abuse of notation, we denote such a function still by $\vdash_N$. Furthermore, in a way analogous to the previous cases, it is possible to show that such a function is partial recursive.

Using the transition function on natural numbers we can now define a recursive function SR_{CM} which uses $\vdash_N$ k times. So this function returns the natural number corresponding to the configuration at the k^{th} step of computation. Below we use the notation $\vdash_N^k$ to denote k applications of the function $\vdash_N$.

Definition 2.21 *Let CM be a CTM as in Definition 2.4 we define a function $SR_{CM} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as:*

$$SR_{CM}(x, 0) = id(x);$$
$$SR_{CM}(x, m+1) = \vdash_N (SR_{CM}(x, m))$$

The previous function SR_{CM} is partial recursive function since it is defined by recursion using the partial recursive function $\vdash_N$.

Definition 2.22 *Let CM be a CTM as in Definition 2.4. We define a function $final : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as:*

$$final(x, y) = \begin{cases} 0 & \text{if } q(SR_{CM}(x, y)) \in Q_f \\ 1 & \text{otherwise} \end{cases}$$

where the function q extracts from a configuration the state and Q_f is the set of final states.

Proposition 2.2 *The function $final$ defined in Definition 2.22 is partial recursive.*

Proof: We observe that Q_f is finite, so we can use the function $case$ in the definition of $final$, which is then the composition of recursive functions. □

Using the function final we can apply the minimization thus obtaining a function $\mathcal{F}_{CM}$ (defined below) which finds the minimal number of steps needed to obtain a final configuration, if it exists. So, if our CTM terminates, this function returns the number of steps needed to terminate the computation, otherwise minimization diverges.

Definition 2.23 (Minimization of Function Computed by a Classical Turing Machine) *Let CM be a CTM as in Definition 2.4. We define $\mathcal{F}_{CM} : \mathbb{N} \to \mathbb{N}$ as follows*

$$\mathcal{F}_{CM}(x) = \mu\, y\, (final(x, y)) = 0)$$

Proposition 2.3 *The function $\mathcal{F}_{CM}$ defined in Definition 2.23 is partial recursive functions.*

Proof: We observe that: $\mathcal{F}_{CM}$ is the composition of recursive function, because we apply the minimization at the function $final$. □

Theorem 2.2 *The class of Partial Recursive Functions $\mathscr{R}$ includes the class of the functions computable by a CTM.*

Proof: The function computed by a CTM is defined in Definition 2.19. This function is a partial recursive function because it can be obtained as the composition $SR_{CM}(x, \mathcal{F}_{CM}(x)$ of functions which, by previous results, are partial recursive functions. □

Hence we have the following.

Theorem 2.3 $\mathscr{R} = \mathscr{CC}$

Proof: The proof that a partial recursive function is computable by a CTM is easy, by constructing the machine which actually computes a function corresponding to the Definition 2.13. The other implication is proved by Theorem 2.2. □

These equivalence results of classic recursion theory will be suitably extended in the next chapters where we will introduce probabilistic recursion theory.

We conclude this chapter with a few lines on the Church-Turing thesis , one of the most important concepts in theoretical computer science which established the universal relevance of the notion of CTM (and of other, equivalent, computing devices). Church, Turing and also Markov realized that the class of function they had defined was general and each one of them put forward the claim that such a class coincided with the class of all the functions that, intuitively, are algorithmically computable. In terms of CTM such a claim, known indeed as the Church-Turing thesis, can be stated as follows (see [7]): *The intuitively and informally defined class of effectively computable partial functions coincides exactly with the class of functions which can be computed by* $CTMs$.

Even though obviously this claim cannot be proved, there has been an impressive number of evidences of its truth, including the fact that many completely different formalisms computer the same class of functions as $CTMs$ and the fact that, so far, no one has found a function which is intuitively computable and which is not computable by a CTM.

Chapter 3

Implicit Computational Complexity

Computability theory studies what can and what cannot be computed by an algorithm: a function f is computable whenever the process of computing its value $f(x)$ from x is effective, meaning that it can be turned into an algorithm. There is no guarantee, however, about the amount of resources such an algorithm requires. Some algorithms can be very inefficient, e.g. they can require an enormous amount of time to produce their output from their input. Sometimes, this inefficiency is due to poor design. But it can well happen that the function f intrinsically requires a great amount of resources to be computed.

Complexity theory refines this analysis by exploring the class of computable functions and classifying them on the basis of the amount of resources required by algorithms computing those functions when they are executed by paradigmatic machines (like Turing machines or Register Machines). Resources of interest here are computation time or space and bounds are expressed as functions of the size of the input to the algorithm. In other words, computational complexity aims at understanding how many resources is necessary and sufficient to perform certain computational tasks and to solve specific problems, trying to establish tight upper and lower bounds [5].

For many computational problems no tight bounds are known. An illustrative example is the well known **P** versus **NP** problem: for all **NP-Complete** problems the current upper and lower bounds lie exponentially far apart. That is, the best known algorithms for these computational problems need exponential time (in the size of the input) but the best lower bounds are of a linear nature.

An important basic idea here is that one of a complexity class. A complexity class can be thought of as a collection of computational problems, all of which share some common features with respect to the computational resources needed to solve those problems. A complexity class can be defined as the collection of all those functions which can be computed by an algorithm working within resource bounds of a certain kind. As an example, we can define the class **FP** of those functions which can be computed in polynomial time, i.e. within an amount of computation time bounded by a polynomial on the size of the input. As another example, we can form the class of those functions which can be computed in logarithmic space.

Starting from the early nineties, several implicit characterizations of the various complexity classes have been introduced, starting from **FP** and later generalizing the approach to many other classes. Implicit here means that classes are not given by constraining the amount of resources a machine is allowed to use, but rather by imposing linguistic constraints on the way algorithms are formulated in a suitable programming language. This idea has developed into an area called implicit computational complexity (ICC) which studies machine-free characterizations of complexity classes. ICC has been mainly developed in the functional programming paradigm using ideas from primitive recursion (Bellantoni and Cook [2]; Leivant [21]) , proof-theory and linear logic (Girard [14]), rewriting systems and type systems (Leivant and Marion [22]; Hofmann [18]). ICC results usually include both a soundness and a completeness theorem: the first one states that all programs which satisfy a given criterion ensure a certain quantitative property. The second one states that all the functions (or problems) of a functional complexity class can be programmed in a given language (consisting of programs which satisfy some specific criteria). For instance, in the case of polynomial time complexity, soundness refers to the possibility of evaluating programs in polynomial time, whereas completeness refers to the class **P** of problems solvable in polynomial time. Theorems of this kind have been given for many systems, including safe and ramified recursion (Bellantoni and Cook [2]; Leivant [21]), variants of linear logic (Girard [14]).

One of the main motivations for ICC comes from programming language theory, because ICC suggests ways to control the complexity properties of programs, which is a

difficult issue because of its infinite nature. However, extensional correspondence with complexity classes is usually not enough: a programming language (or a static analysis methodology for it) offering guarantees in terms of program safety is plausible only if it captures enough interesting and natural algorithms. In turn, this cannot be guaranteed by merely requiring that the system corresponds extensionally to a complexity class. This issue has been pointed out by several authors as i.e. Hofmann [18], and advances have been made in the direction of more liberal ICC systems: some examples are type systems for non-size-increasing computation (Hofmann [18]) and quasi-interpretations (Bonfante et al. [4]).

In the remaining of this chapter we introduce some basic notions concerning ICC and in particular we describe the approaches of Bellantoni and Cook [2] and of Leivant [21] because this will be the basis for our work.

3.1 Prelimiaries

In order to explain more precisely ICC we first need to be more precise on how we model the time and the space needed by a program executing an algorithm. Here we will be concerned with time only, however analogous considerations apply to space.

First of all we have to refer to an execution machine. In our case this can be the Turing Machine or the Register Machine. Given a, say, Turing program δ computed by a Turing Machine P. We say that it works in time $f : \mathbb{N} \to \mathbb{N}$ iff for every initial configuration $\mathcal{IN}_P^s$, a final configuration $\mathcal{FC}_P^s$ is reached in, at most, $f(|s|)$ computation steps (that is, each computation uses at most $n \leq f(|s|)$ transitions). In case f is polynomial we say that P works in polynomial time (the terminology is analogous for other classes of functions). The class **FP** is the class of functions which are computable in polynomial time:

$$\{f : \Sigma^* \to \Sigma^* | f \text{ is computed by } \textit{some} \text{ Turing machine } P \text{ working in polynomial time}\}$$

FP is one of the most interesting classes since it is universally accepted as somehow capturing the intuitive notion of a feasible function, namely one which can be computed

in an amount of time which grows in an acceptable way with respect to the size of the input. Such a class is closed by composition, see for example [8].

Now the question that implicit complexity addresses is whether it is possible to define such classes as **FP** by way of function algebras, thus avoiding reference to explicit machine models, but also without any reference to polynomials. Usually these implicit characterizations of complexity classes are obtained by defining suitable restrictions on the form that generic partial recursive functions (or their counterpart in a suitable formalism such a a functional language) assume. More precisely, often one poses restrictions of the form of recursion which is allowed.

For example, one of the earliest results in ICC, due to Bellantoni and Cook [2], shows that an implicit characterization of the polynomial time computable functions can be obtained by means of a function algebra in the style of the one of general recursive functions where, however, a restricted form of recursion (called *safe recursion*) is used. This will be made more precise in the next section.

3.2 Safe Recursion

In this section we describe *safe recursion* following the original paper [2]. Safe functions are functions which are identified by pairs (f, n), where $f : \mathbb{B}^m \to \mathbb{B}$ and $0 \leq n \leq m$ and $\mathbb{B} = \{0,1\}^*$ (we could use a different alphabet here to construct words, however we use $B = \{0,1\}$ for simplicity). The number n identifies the number of the, so called, normal arguments of f (which by convention are the first n). The remaining $(m - n)$ arguments are called safe. Following [2] we use semicolons to separate normal and safe arguments: so, if (f, n) is a safe function, we write $f(\mathbf{x}; \mathbf{y})$ to emphasize that $\mathbf{x}$ are the n normal arguments while $\mathbf{y}$ are the safe arguments.

The idea of separating "normal" arguments from "safe" ones is that safe functions can perform any (polytime) operation on their normal arguments, while they can apply only a restricted set of operations to safe inputs: these restricted operations do not increase the length by more than an addictive constant. The specific notion of Safe recursion and safe composition used by safe functions ensure that the recursive value is substituted in a

safe position and will remain in a safe position without being copied to a normal position. This ensures that the depth of sub-recursion cannot depend on the value being recursively computed, thus avoiding the blowup in complexity. In other words, one can input a large value in safe positions without greatly increasing the size of the function.

We provide now the definition of safe functions starting from basic functions.

- The Constant function $(e, 0)$ where $e : \mathbb{B} \to \mathbb{B}$ always returns the empty string ϵ.
- The Successor function $(c_a, 0)$ where $c_a : \mathbb{B} \to \mathbb{B}$ is defined as follows: $c_a(; w) = a \cdot w$, where $a \in \{0, 1\}$.
- The Predecessor function $(p, 0)$ where $p : \mathbb{B} \to \mathbb{B}$ is defined as follows: $p(; \epsilon) = \epsilon$, $p(; a \cdot w) = w$ where $a \in \{0, 1\}$.
- The Case function $(case, 0)$ where $case : \mathbb{B}^3 \to \mathbb{B}$ is defined as follows: $case(; \epsilon, w, v) = w$, $case(; a \cdot x, w, v) = w$ and $case(; b \cdot x, w, v) = v$, where $a, b \in \{0, 1\}$ and $a \neq b$
- A projector operator that is defined in the usual way (as in the previous chapter).

Then, functions can be formed by safe recursion on notation or by safe composition:

- Suppose that $(f : \mathbb{B}^m \to \mathbb{B}, n)$, that $(\bar{g} : \mathbb{B}^k \to \mathbb{B}, k)$ and $\bar{g}$ is a vector of n functions, and that $(\bar{h} : \mathbb{B}^{k+i} \to \mathbb{B}, k)$ where $\bar{h}$ is a vector of m functions. Then the safe composition of safe functions above is the safe function $(p : \mathbb{B}^{k+i} \to \mathbb{B}, k)$ defined as follows:

$$p(\mathbf{w}; \mathbf{v}) = f(\bar{g}(\mathbf{w};); \bar{h}(\mathbf{w}; \mathbf{v})).$$

- Suppose that, for every $1 \leq i \leq n$ and for every $a \in \{0, 1\}$, functions $(h : \mathbb{B}^m \to \mathbb{B}, n)$ and $(g_a : \mathbb{B}^{m+2} \to \mathbb{B}, n + 1)$ are safe functions. Then for every $1 \leq i \leq n$, the function $(f : \mathbb{B}^{m+1} \to \mathbb{B}, n + 1)$ defined as follows:

$$\begin{aligned} f(\epsilon, \mathbf{w}; \mathbf{v}) &= h(\mathbf{w}; \mathbf{v}); \\ f(a \cdot x, \mathbf{w}; \mathbf{v}) &= g_a(x, \mathbf{w}; \mathbf{v}, f(x, \mathbf{w}; \mathbf{v})); \end{aligned}$$

  are said to be defined by safe recursion on notation from (h, n) and $(g_a, n + 1)$.

We denote by $\mathscr{B}$ the smallest set of functions which includes the basic functions above and which is closed under safe composition and safe recursion. It is worth noting that in $\mathscr{B}$ we do not have a minimization operator and we have a set of basic functions which is larger than in the standard case of partial recursive functions. In the work [2] it is proved that a soundness result, showing that $\mathscr{B} \subseteq$ **FP**, and a completeness ones, which shows that **FP** $\subseteq \mathscr{B}$.

3.3 Ramified Recurrence

We introduce now another form a restricted recursion, called *Ramified Recurrence* which has been defined by Leivant [21] in order to restrict the set of functions definable via recursion and which also allows to characterize the polytime functions. The main idea of ramified recurrence is to restrict the domain and codomain of recursion by using the concept of *tier*.

Intuitively, any function and argument position comes with a tier. Equivalently, given a set of base data $\mathbb{W}$ we have an infinite number of copies of it $\mathbb{W}_0, \mathbb{W}_1, \cdots$ where $\mathbb{W}_j$ represents the elements at the j^{th}abstraction level (tier). The constructor c_a^j is the general constructor c at the level j. When a variable x ranges over $\mathbb{W}_i$ we write that the $tier(x) = i$.

Base functions are available at any tier and composition is tier-preserving. Recursion is possible only over a variable with tier greater than that of the function and this restriction avoid complexity blow up.

More precisely, following [21], we have the following basic functions:

- the ϵ function $\epsilon : \mathbb{W} \to \mathbb{W}$ is defined as $\epsilon(v) = \epsilon$
- the Concatenation fucntion $c_a : \mathbb{W} \to \mathbb{W}$ is defined as $c_a(v) = a \cdot v$
- the Projection function $\Pi_n^k : \mathbb{W}^n \to \mathbb{W}$ is defined as: $\Pi_n^k(\mathbf{v}) = v_k$

The *Composition* of functions $f : \mathbb{W}^n \to \mathbb{W}$, $g_1 : \mathbb{W}^k \to \mathbb{W}, \ldots, g_n : \mathbb{W}^k \to \mathbb{W}$ as the function $f(g_1, \ldots, g_n) : \mathbb{W}^k \to \mathbb{W}$ defined as follows: $h(\mathbf{v}) = f(g_1(\mathbf{v}), \cdots, g_n(\mathbf{v}))$

In [21] is proved that composition is a particular case of Recurrence over $\mathbb{W}$ which takes the form:

$$\begin{aligned} f(\epsilon, \mathbf{y}) &= g_\epsilon(\mathbf{y}) \\ f(a \cdot w, \mathbf{y}) &= g_a(a', w, \mathbf{y}) \text{ where } a' = f(w, \mathbf{y}) \end{aligned}$$

where $a \in \Sigma$ and $f : \mathbb{W} \times \mathbb{W}^m \to \mathbb{W}$, $g_a : \mathbb{W} \times \mathbb{W} \times \mathbb{W}^m \to \mathbb{W}$, for all $a \in \Sigma$. We use $f = rec(g_\epsilon, g_{a\in\Sigma})$ as a shorthand for previous definition of recurrence.

We call functions g_a the *recurrence functions*, a' is the *critical argument* and $a \cdot w$ is the *recurrence argument*.

The following construction is redundant in presence of primitive recursion, but we prefer to introduce explicitly it following Leivant's paper which defines the function *case* as an example of *flat* recursion. The *flat* recursion is a recursion which has not *critical argument* in his definition.

Definition 3.1 (Case Distinction) *If $g_\varepsilon : \mathbb{W}^k \to \mathbb{W}$ and for every $a \in \Sigma$, $g_a : \mathbb{W}^{k+1} \to \mathbb{W}$, we define a function $h : \mathbb{W}^{k+1} \to \mathbb{P}_\mathbb{W}$ by* case distinction *by stipulating that $h(\varepsilon, \mathbf{y}) = g_\varepsilon(\mathbf{y})$ while $h(a \cdot w, \mathbf{y}) = g_a(w, \mathbf{y})$. The function h is denoted as $case(g_\varepsilon, \{g_a\}_{a\in\Sigma})$.*

Finally Ramified recurrence is defined as follows:

$$f_{ram}(a \cdot v, \mathbf{w}) = g_a(f_{ram}(v, \mathbf{w}), v, \mathbf{w})$$

where the tier j of the recurrence argument of f_{ram} be higher than the tier of the critical argument.

In [21] it is proved that the class of functions constructed by using ramified recurrence is the same as that one of polytime functions. This result is proved by performing the following steps:

- First one can prove, by using a careful encoding, that a form of *simultaneous primitive recursion* is available in predicative recursion.
- One knows that CTMs are equivalent, in terms of expressivity, to *register* machines;
- Then any function definable by predicative recurrence can be proved computable by a polytime register machine.
- Last, one can give an embedding of any polytime register machine into a predicatively recursive function by using of simultaneous recurrence.

Chapter 4

Probabilistic Turing Machines

In this chapter we first recall some basic notions on Probabilistic Turing Machines (PTM) and then we provide a notion of function computed by a PTM which is somehow different from those which are common in the literature. In fact, as mentioned in the introduction, rather than *reducing* a probabilistic computation to a deterministic one, we want to study probabilistic computation directly. From this perspective for us a PTM is a device which 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.

As we have seen in the previous chapter, Turing defined a class of computing machines now known as Turing machines, which may be used to characterize a class of functions known as the partially recursive functions. Turing machines are deterministic machines. On the other hand, probabilistic computation devices have received a wide interest in computer science already in the Fifties [12] and early Sixties [28]. 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 (PTM) [12, 31] which, essentially, are Turing machines with the ability to flip coins in order to make random decisions, and to several results concerning the computational complexity of PTMs (see [26]). In particular, one of the main questions, which has been asked from many different technical perspectives, is the following: does the addition of random elements increase the expressive power of the machine? Already the authors of [12] investigated whether a machine which has access to random inputs has an expressive

power greater than a traditional deterministic machine. The random inputs were required to be drawn from a source of independent, equiprobable binary digits, and the tasks to be performed by the machines were the enumerations of sets of natural numbers. This question was answered in the negative. Later on, Santos [31, 32] showed that probabilistic Turing machines could be used to characterize the class of partially computable random functions. The class of ordinary functions which are partially computable random functions was shown to be equivalent to the class of partially recursive functions even though there exist classes of ordinary functions characterizable by PTM's which contain the class of partially recursive functions as proper sub-class [32].

It is worth noticing that in most works on probabilistic Turing machines the results of a PTM's computation are considered only the elements which have an associated probability greater than $\frac{1}{2}$. For example, Gill [13] defines the output of a PTM M on input x as the element y such that $P\{M(x = y)\} > \frac{1}{2}$. This means that, according to this approaches, the final result of a PTM is at most one element of the distribution on the results and for this reason we call all these approaches *reductionist*, since a probabilistic computation is somehow reduced to a deterministic one, where for one input one has at most one output.

As mentioned above, in this thesis we take a completely different view, since from our prospective a probabilistic Turing machine is a device computing a "probabilistic function", i.e. a function from a discrete set to distributions over the same set. Hence, in the following, after some standard notions on PTM, we will provide a different definition of the function computed by a PTM. Finally we observe that distributions which correspond to outputs of PTM's computation are essentially lower semi-computable semi-distributions (or measures) which have been extensively studied in other contexts (see the book [23]) .

4.1 Basic definitions

A randomized algorithm is an algorithm that may involve choices such as initializing a variabile with a random value chosen from some range. In practice randomized algo-

rithms are implemented using a random number generator. In order to model randomized algorithms we use probabilistic Turing machines, as we use classical Turing machine to model deterministic algorithms.

Following [13], PTMs M can be seen as a Turing Machine with two transition functions δ_0, δ_1. At each computation step, either δ_0 or δ_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 M. In the following, Σ_b denotes the set of possible symbols on the tape, including a blank symbol $\square$; 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. Below we give the formal definitions.

Definition 4.1 (Probabilistic Turing Machine) *A probabilistic Turing machine is a Turing machine endowed with two transition functions δ_0, δ_1. At each computation step the transition function δ_0 can be applied with probability $\frac{1}{2}$ and the transition δ_1 can be applied with probability $\frac{1}{2}$.*

According to our definition of PTM, the functions δ_0, δ_1 are defined also in case they produce the same output (starting from the same input).

Analogously to the classic case, our PTM has three types of configurations:

- A *General Configuration* is any configuration of a PTM M;
- An *Initial Configuration* is the configuration which contain the initial state of a PTM M;
- A *Final Configuration* is a configurations which contains a final state of the PTM M.

Definition 4.2 (General 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 4.3 (Initial and Final Configurations of a PTM) *Let M be a PTM. We define an* initial configuration *of M for the string s as a configuration in the form $\langle \epsilon, a, v, q_s \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ such that: $s = a \cdot v$ and the fourth component, $q_s \in Q$, is the initial state. We denote with $\mathcal{IN}_M^s$ the set of all such initial configurations. Similarly, we define a* final configuration *of M for s as a configuration $\langle s, \_, \varepsilon, q_f \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q_f$. The set of all such final configurations for a PTM 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.

As previously mentioned, for us the function computed by a PTM M associates to each input s, a distribution 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 by different computations, 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.

Definition 4.4 (String Distribution and Probabilistic String Function) *A* string distribution *on Σ^* 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 distributions on Σ^*. A* probabilistic string function *(PSF) is a function from $(\Sigma^*)^k$ to $\mathbb{P}_{\Sigma^*}$, where $(\Sigma^*)^k$ stands for the set of k-tuples in Σ^*. We use the expression $\{s_1^{p1}, \ldots, s_k^{pk}\}$ to denote the distribution $\mathcal{D}$ defined as $\mathcal{D}(s) = \sum_{s_i = s} p_i$. Observe that $\sum \mathcal{D} = \sum_{i=1}^{k} p_i$. When this does not cause ambiguity, a string distribution is simply called a distribution.*

In order to define more precisely the function computed by a PTM we first observe that, given a PTM M with a initial state cs_i and input x, we can represent a computation as a binary number n, where the i-th digit represent the δ function which has been applied at the i-th derivation step: if the digit is 0 then δ_0 has been applied, if it is 1 then δ_1 was used.

We give a formal definition of the function $\vdash^{\overline{n}}$, which intuitively represents the sequence of transactions applied starting from the initial configuration cs_i in order to obtain the final configuration cs_n.

Definition 4.5 *Let M be a PTM. For let $a \in \{0,1\}$ we denote by $\vdash_a$ the function defined in Definition 2.7 where it is assumed that the function δ_a of M is used. Given a binary number $\overline{n}$ we defined inductively (on the number of digits in $\overline{n}$) the function $\vdash^{\overline{n}} : \mathcal{C}_M \to \mathcal{C}_M$ as follows: if, for all $w \in \Sigma^*$ where w is an Initial Configuration*

$$\vdash^{\overline{n}} (cs_i) = \begin{cases} \vdash^a (cs_i) & \text{if } \overline{n} = a \text{ and } a \in \{0,1\} \\ \vdash^a (\vdash^{\overline{m}} (cs_i)) & \text{if } \overline{n} = a \cdot \overline{m} \text{ and } a \in \{0,1\} \end{cases}$$

Using the previously defined function we can define $CC_M(x, cs_k)$ as the set of binary numbers representing all the computations for the machine M starting with the configuration cs_i, input x and ending with the configuration cs_k. More precisely we can define

$$CC_M(x, cs_k) = \{y \mid \vdash^{\overline{y}} (cs_i) = cs_k\}$$

Then we can define the computed function of a PTM as follows.

Definition 4.6 (Probabilistic Turing computable function) *Let M be a PTM the function $f : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}$ computed by M is defined as follows*

$$f(cs_i)(cs_{fi}) = \begin{cases} \Sigma_{y \in CC_M(x, cs_{fi})} \frac{1}{2^{|\overline{y}|}} & \text{if } cs_{fi} \in \mathcal{FC}^s_M \\ 0 & \text{otherwise} \end{cases}$$

4.2 A Fixpoint Characterization of the Function Computed by a PTM

In this section is presented the function computed by a PTM through a fixpoint construction. We can define a partial order on string distributions by a point wise extension of the usual order on $\mathbb{R}$:

Definition 4.7 *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 4.1 *The structure* $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$ *is a POSET.*

Now we can define the domain $\mathcal{CEV}$ of those functions computed by a PTM M from a given configuration[1]. This set is defined as follows and will be used as the domain of the functional whose least fixpoint gives the function computed by a PTM.

Definition 4.8 *The set* $\mathcal{CEV}$ *is defined as* $\{f | f : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}\}$

Inheriting the structure on $\mathbb{P}_{\Sigma^*}$ we can define a partial order on $\mathcal{CEV}$ as follows.

Definition 4.9 *The relation* $\sqsubseteq_{\mathcal{CEV}} \subseteq \mathcal{CEV} \times \mathcal{CEV}$ *is defined for* $A, B \in \mathcal{CEV}$ $A \sqsubseteq_{\mathcal{CEV}} B$ *if and only if, for all* $c \in \mathcal{C}_M$, $A(c) \sqsubseteq_{\mathbb{P}_{\Sigma^*}} B(c)$

Also the proof of the following is immediate.

Proposition 4.2 *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 (see also Chapter 2). Next, the bottom elements are defined as follows.

Lemma 4.1 *Let* $d_\perp : \Sigma^* \to \mathbb{R}_{[0,1]}$ *be defined by stipulating that* $d_\perp(s) = 0$ *for all* $s \in \Sigma^*$. *Then,* $d_\perp$ *is the bottom element of the poset* $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$.

Lemma 4.2 *Let* $b_\perp : \mathcal{C}_M \to \mathbb{P}_{\Sigma^*}$ *be defined by stipulating that* $b_\perp(c) = d_\perp$ *for all* $c \in \mathcal{C}_M$. *Then,* $b_\perp$ *is the bottom element of the poset* $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$.

Now, it is time prove that the posets at hand are also ω-complete:

Proposition 4.3 *The POSET* $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$ *is a* ω*CPO.*

[1] 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_{LIM} as the distribution such that $c_{LIM}(s) = \lim_{i\to\infty} c_i(s)$, we have that $c_{LIM} = \bigsqcup_i c_i$. Indeed, $c_{LIM} \sqsupseteq_{\mathbb{P}_{\Sigma^*}} c_i$, and any upper bounds of the family $\{c_i\}_{i\in\mathbb{N}}$ is clearly greater or equal to c_{LIM}. □

Proposition 4.4 *The POSET $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$ is a ωCPO.*

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

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

The following proposition is needed in order to apply the usual fix point result.

Proposition 4.5 *The functional F_M is continuos.*

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,

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

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 4.1 *The functional defined in 4.10 has a least fix point which is equal to* $\bigsqcup_{n\geq 0} F^n_M(b_\perp)$.

Proof: Immediate from the well-known fix point theorem for continuous maps on a ωCPO. □

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 $\mathcal{IO}_M : \Sigma^* \to \mathbb{P}_{\Sigma^*}$. The set of those functions which can be computed by any PTMs is denoted as $\mathscr{PC}$.

The notion of a computable probabilistic function subsumes other key notions in probabilistic and real-number computation. As an example, *computable distributions* can be characterized as those distributions on Σ^* 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 $f \in \mathscr{PC}$.

Chapter 5

Probabilistic Recursion Theory

In this chapter 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 show that this class coincide with the class $\mathscr{PC}$ of probabilistic functions computed by a Probabilistic Turing Machine.

5.1 Probabilistic Recursive Functions

Since PTMs compute probability distributions, the functions that we consider in our algebra have domain $\mathbb{N}^k$ 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 $y \in \mathbb{N}$, then p is the probability of getting y 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 $\mathbb{N}^k \to \mathbb{R}_{[0,1]}$, however we use $\mathbb{N} \to \mathbb{R}_{[0,1]}$ in order to simplify the presentation.

Definition 5.1 (Distributions and Probabilistic Functions) *A* distribution *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 as* $\sum \mathcal{D}$. *Let* $\mathbb{P}_{\mathbb{N}}$ *be the set of all distributions on* $\mathbb{N}$. *A* probabilistic function *(PF) is a function from* $\mathbb{N}^k$ *to* $\mathbb{P}_{\mathbb{N}}$, *where* $\mathbb{N}^k$ *stands for the set of* k*-tuples in* $\mathbb{N}$. *We use the expression* $\{n_1^{p1}, \ldots, n_k^{pk}\}$ *to denote the distribution* $\mathcal{D}$ *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 distribution 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 5.2 (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:* $z(n)(0) = 1$ *for every* $n \in \mathbb{N}$*;*
- *The* successor function $s : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $s(n)(n+1) = 1$ *for every* $n \in \mathbb{N}$*;*
- *The* projection function $\Pi^n_m : \mathbb{N}^n \to \mathbb{P}_{\mathbb{N}}$ *defined as:* $\Pi^n_m(k_1, \cdots, k_n)(k_m) = 1$ *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:*

$$r(x)(y) = \begin{cases} \frac{1}{2} & \text{if } y = x \\ \frac{1}{2} & \text{if } y = x + 1 \end{cases}$$

The first three BPFs are the same as the basic functions from classic recursion theory, while r is the only truly probabilistic BPF.

The next step consists in defining how PFs *compose*. Function composition of course cannot be used here, because when composing two PFs g and f 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 $z \in \mathbb{N}$ and an output $y \in \mathbb{N}$ for the composition $f \bullet g$, we apply the distribution $g(z)$ to any value $x \in \mathbb{N}$. This gives a probability $g(z)(x)$ which is then multiplied by the probability that the distribution $f(x)$ associates to the value $y \in \mathbb{N}$. If we then consider the sum of the obtained product $g(z)(x) \cdot f(x)(y)$ on all possible $x \in \mathbb{N}$ we obtain the probability of $f \bullet g$ returning y when fed with z. The sum is due to the fact that two different values, say $x_1, x_2 \in \mathbb{N}$, which provide two different distributions $f(x_1)$ and $f(x_2)$ must both contribute to the same probability value $f(x_1)(y) + f(x_2)(y)$ for a specific 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 we have the following definition.

Definition 5.3 (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:*

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

The previous definition can be generalized to functions taking more than one parameter in the expected way:

Definition 5.4 (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:*

$$((f \odot (g_1, \ldots, g_n))(\mathbf{z}))(y) = \sum_{x_1, \ldots, x_n \in \mathbb{N}} \left(f(x_1, \ldots, x_n)(y) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{z})(x_i) \right).$$

With a slight abuse of notation, we can treat random functions as ordinary functions when forming expressions. Suppose, as an example, that $x \in \mathbb{N}$ and that $f : \mathbb{N}^3 \to \mathbb{P}_{\mathbb{N}}$, $g : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ and $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$. 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 in PF.

The way we have defined probabilistic functions and their composition is reminiscent of, and indeed inspired by, the way one defines the Kleisly category for the Giry monad [15], 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 that one uses composition as previously defined:

Definition 5.5 (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:*

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

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, the minimization operator allows from a partial function $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$, 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, if such a value exists (and it 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 $\mathbf{z} \in \mathbb{N}^k$, returns a distribution associating to each natural y the probability that the result of $f(\mathbf{z}, y)$ is 0 *and* the result of $f(\mathbf{z}, x)$ is positive for every $x < y$. Formally:

Definition 5.6 (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:*

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

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

Definition 5.7 (Probabilistic Recursive Functions) *The class* $\mathscr{PR}$ *of* probabilistic recursive functions *is the smallest class of probabilistic functions that contains the basic functions (Definition 5.2) and is closed under the operation of General Composition (Definition 5.4), Primitive Recursion (Definition 5.5) and Minimization (Definition 5.6).*

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

Definition 5.8 (Extended Recursive Functions) *For every partial recursive function* $f : \mathbb{N}^k \to \mathbb{N}$ *we define the extended function* $p_f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ *as follows:*

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

Proposition 5.1 *If* f *is a 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.

- f is the zero function, so $f : \mathbb{N} \to \mathbb{N}$ defined as: $f(x) = 0$ for every $x \in \mathbb{N}$. Thus p_f is in $\mathscr{PR}$ because $p_f = z$.

- f is the successor function, so $f : \mathbb{N} \to \mathbb{N}$ defined as: $f(x) = x + 1$ for every $x \in \mathbb{N}$. Thus p_f is in $\mathscr{PR}$ because $p_f = s$.

- f is the projection function, so$f^n_m : \mathbb{N}^n \to \mathbb{N}$ defined as: $f^n_m(x_1, \cdots, x_n) = x_m$ for every positive $n \in \mathbb{N}$ and for all $m \in \mathbb{N}$, such that $1 \leq m \leq n$. Thus p_f is in $\mathscr{PR}$ because $p_f = \Pi^n_m$.

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

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

  where $h : \mathbb{N}^n \to \mathbb{N}$ and $g_i : \mathbb{N}^k \to \mathbb{N}$ for every $1 \leq i \leq n$ are partial recursive functions. By definition of $f(\mathbf{x})$ we have

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

  We see that $h, g_1, \cdots, g_n$ are all partial recursive functions. So we have by definition of p_f that

  $$p_{g_1}(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = g_1(\mathbf{x}) \\ 0 & otherwise \end{cases}$$

  $$\vdots$$

$$p_{g_n}(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = g_n(\mathbf{x}) \\ 0 & otherwise \end{cases}$$

$$p_h(\mathbf{z})(y) = \begin{cases} 1 & \text{if } y = h(\mathbf{z}) \\ 0 & otherwise \end{cases}$$

By hypothesis we observe that $p_{g_1}, \cdots, p_{g_n}, p_h \in \mathscr{PR}$ and

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

Thus p_f is in $\mathscr{PR}$ because $p_f = ((p_h \odot (p_{g_1}, \ldots, p_{g_n}))(\mathbf{x}))(y)$.

- f is defined by primitive recursion so $f : \mathbb{N}^k \times \mathbb{N} \to \mathbb{N}$ defined as:

$$f(\mathbf{x}, 0) = h(\mathbf{x})$$

$$f(\mathbf{x}, n+1) = g(\mathbf{x}, n, f(\mathbf{x}, n))$$

where $g : \mathbb{N}^k \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N}^k \to \mathbb{N}$ are partial recursive functions.

p_f is defined as:

$$p_f(\mathbf{x}, n)(z_n) = \begin{cases} 1 & \text{if } z_n = rec(h, g) \\ 0 & otherwise \end{cases}$$

We see that h, g are all partial recursive functions. So we have by definition of p_f that

$$p_h(\mathbf{x})(z_0) = \begin{cases} 1 & \text{if } y = h(\mathbf{x}) \\ 0 & otherwise \end{cases}$$

$$p_g(\mathbf{x}, n, z_n)(z_{n+1}) = \begin{cases} 1 & \text{if } z_{n+1} = g(\mathbf{x}, n, f(\mathbf{x}, n)) \\ 0 & otherwise \end{cases}$$

By hypothesis we observe that $p_g, p_h \in \mathscr{PR}$. Now if $n = 0$ then $p_f(\mathbf{x}, 0) = p_h(\mathbf{x})$ and if $n > 0$ then $p_f(\mathbf{x}, n+1) = p_g(\mathbf{x}, n, z_n)$. We observe that $((p_g \odot (id, p_f))(\mathbf{x}, n))(z_{n+1})$ =

$$= \sum_{x_1,\ldots,z_k,n,z_n \in \mathbb{N}} p_g(x_1, \ldots, x_k, n, z_n)(z_{n+1}) \cdot (\prod_{1 \leq i \leq k+1} id(\mathbf{x}, n)(\mathbf{x}, n) \cdot f(\mathbf{x}, n)(z_n))$$

$$= \sum_{x_1,\ldots,z_k,n,z_n \in \mathbb{N}} p_g(x_1, \ldots, x_k, n, z_n)(z_{n+1}) \cdot (\prod_{\mathbf{x}=\mathbf{x}, n=n, z_n=f(\mathbf{x},n)} 1)$$

$$= \sum_{z_{n+1}=g(x_1,\ldots,x_k,n,z_n)} 1 \cdot (\prod_{\mathbf{x}=\mathbf{x}, n=n, z_n=f(\mathbf{x},n)} 1)$$

$$= \sum_{z_{n+1}=g(\mathbf{x},n,f(\mathbf{x},n))} 1$$

$$= p_f(\mathbf{x}, n+1)(z_{n+1})$$

Thus p_f is in $\mathscr{PR}$ because $p_f = rec(p_h, p_g)$.

- f is defined by minimization so:

$$f(\mathbf{x}) = \mu\ y\ (g(\mathbf{x}, y) = 0)$$

p_f is defined as:

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

by definition of $f(\mathbf{x})$ we have:

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

We know that g is a recursive function, so we have that:

$$p_g(\mathbf{x}, z)(k) = \begin{cases} 1 & \text{if } k = g(\mathbf{x}, z) \\ 0 & otherwise \end{cases}$$

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

$$\begin{aligned}
\mu\ p_g(\mathbf{x})(z) &= p_g(\mathbf{x}, z)(0) \cdot (\prod_{n<z} (\sum_{k>0} p_g(\mathbf{x}, n)(k))) \\
&= p_g(\mathbf{x}, z)(0) \cdot (\prod_{n<z} (\sum_{k>0, k=g(\mathbf{x},n)} 1)) \\
&= p_g(\mathbf{x}, z)(0) \cdot (\prod_{n<z, k>0, k=g(\mathbf{x},n)} 1) \\
&= \begin{cases} 1 & \text{if z is the minimal values such that} \\ & g(\mathbf{x}, z) = 0 \text{ and for all } n < z\ g(\mathbf{x}, z) > 0 \\ 0 & otherwise \end{cases} \\
&= \begin{cases} 1 & \text{if } z = \mu\ y\ (g(\mathbf{x}, y) = 0) \\ 0 & otherwise \end{cases} \\
&= p_f(\mathbf{x})(z)
\end{aligned}$$

Thus p_f is in $\mathscr{PR}$ because $p_f = \mu\ p_g$

□

Example 5.1 *T*he following are examples of probabilistic recursive functions:

– The *identity function* $id : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$, defined as $id(x)(x) = 1$. For all $x, y \in \mathbb{N}$ we have that

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

on the other hand for every $x, y \in \mathbb{N}$ we have as a consequence, $id = \Pi^1_1$, and, since the latter is a basic function (Definition 5.2) id is in $\mathscr{PR}$.

– The probabilistic function $rand : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ such that for every $x \in \mathbb{N}$, $rand(x)(0) = \frac{1}{2}$ and $rand(x)(1) = \frac{1}{2}$ can be easily shown to be recursive, since $rand = r \odot z$ (and we know that both r and z are BPF). Actually, $rand$ could itself be taken as the only genuinely probabilistic BPF, i.e., r can be constructed from $rand$ and the other BPF by composition and primitive recursion. We proceed by defining $g : \mathbb{N}^3 \to \mathbb{P}_{\mathbb{N}}$ as follow:

$$g(x_1, x_2, z)(y) = \begin{cases} 1 & \text{if } y = z + 1 \\ 0 & \text{otherwise} \end{cases}$$

g is in $\mathscr{PR}$ because $g = s \odot (\Pi^3_3)$. Now we observe that the function add defined by $add(x, 0) = id(x)$ and $add(x_1, x_2 + 1) = g(x_1, x_2, add(x_1, x_2))$ is a probabilistic recursive function, since it can be obtained from basic functions using composition and primitive recursion. We can conclude by just observing that $r = add \odot (\Pi^1_1, rand)$.

– All functions we have proved recursive so far have the property that the returned distribution is *finite* for any input. Indeed, this is true for every probabilistic *primitive* recursive function, since minimization is the only way to break this form of finiteness. Consider the function $f : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ defined by

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

We define $h : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ as follows:

$$h(x) = \begin{cases} \frac{1}{2^y} & \text{if } y \geq 1 \\ 0 & otherwise \end{cases}$$

that is a probabilistic recursive function because

$$\mu\ rand(y) = rand(y)(0) \cdot (\prod_{y<z} (\sum_{k>0} rand(z)(k)))$$

thus $rand(y)(0) = \frac{1}{2}$ and $\sum_{k>0} rand(z)(k) = \sum_{k=1,2,\cdots} rand(z)(k) = rand(z)(1) + rand(z)(2) + \cdots = \frac{1}{2} + 0 + \cdots = \frac{1}{2}$ for definition of rand. $\prod_{y<z} \frac{1}{2} = \frac{1}{2^{y-1}}$. So

$$\mu\ rand(y) = \begin{cases} \frac{1}{2^y} & \text{if } y \geq 1 \\ 0 & otherwise \end{cases}$$

Then we observe that:

$$g(x)(y) = add \odot (\mu\ rand, id)(x)(y)$$

So

$$add \odot (\mu\ rand, id)(x)(y) = \sum_{x_1,x_2} add(x_1, x_2)(y) \cdot (\mu\ rand(x_1) * id(x)(x_2)) = \frac{1}{2^{y-x}}$$

because the function $id(x)(x) = 1$ and so $x_2 = x$ and $\mu\ rand(x_1) = \frac{1}{2^{x_1}}$ if $x_1 > 0$. So $x_1 > 0$. Now this is true if and only if $x_2 = x$ and $x + x_1 = y$ and finally $x_1 = y - x$. □

5.2 Probabilistic Recursive Functions equals Functions computed by Probabilistic Turing Machines

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 $\overline{(\cdot)}$.

In order to prove the equivalence result we first need to show that a 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 by composition, primitive recursion and minimization.

Analogously to the classic case this is done by the following Lemmata. These Lemmata allow us to define $PTMs$ for all BPF and all operations.

Lemma 5.1 (Basic Functions are Computable) *All Basic Probabilistic Functions are Computable.*

Proof: For every basic function from Definition 5.2, we can construct a probabilistic Turing machine that computes it quite easily. More specifically, the proof is immediate for functions, z, s, Π, by observing that they are deterministic, thus the usual Turing machine

for them (seen as a PTM) suffice. As for the function r it can be simulated by a PTM M which operates as follows:

1. M deletes all the input written on the tape;
2. M writes $\overline{1}$ or $\overline{0}$ on the tape, both with probability $\frac{1}{2}$, and then halts.

This concludes the proof. □

The composition of two computable probabilistic functions is itself computable:

Lemma 5.2 (Generalized Composition and Computability) *Given Turing-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 Turing-Computable*

Proof: We give an informal proof. We define a PTM, say M, working on $n+2$ tapes. (We know that PTMs with $m > 1$ tapes compute the same class of functions of PTMs with a single tape.) The first tape is the input tape, on the next n tapes M computes $g_1, \cdots, g_n$, while on the last tape, M computes the function f on the results of $g_1, \cdots, g_n$. The machine M operates as follows:

1. it copies the input from the first to the next n tapes;
2. in the $i+1$-th tape, the machine M computes the respective function g_i, where $1 \leq i \leq n$; this can of course be done, because, by induction, the g_i are computable;
3. it copies the n outputs in the n tapes numbered $2, \ldots, n+1$ to the last tape;
4. computes the function f on the last tape and return the result z.

The machine M define above, by construction, computes $g(\mathbf{i})$ on each of the n tapes and then compose the result of these computation with the computation of f on the last tape. One can then see that the distribution computed by M is

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

Indeed, by our construction each $g_{\mathbf{i}}$ and f are computable by PTM (operating on a single tape); we call these PTM $M_{g_{\mathbf{i}}}$ and M_f, respectively. Then we denote by $f_{M_{g_{\mathbf{i}}}}$, f_{M_f} the output of these machines which, according to Definition 4.6, are the following:

$$f_{M_{g_i}}(\mathbf{x})(z_i) = \begin{cases} \Sigma_{k \in CC_M(x,z)} \frac{1}{2^{|k|}} & \text{if } z \in \mathcal{FC}_M^x \\ 0 & \text{otherwise} \end{cases}$$

$$f_{M_f}(\mathbf{z})(y) = \begin{cases} \Sigma_{p \in CC_M(z,y)} \frac{1}{2^{|p|}} & \text{if } y \in \mathcal{FC}_M^z \\ 0 & \text{otherwise} \end{cases}$$

Then in our construction of the machine M above we compose the PTMs M_{g_i} and M_f. Since the output of M_{g_i} is an input for M_f, in the computation of the composition we have to consider the path of each machine M_{g_i} to arrive to a solution that will be the input for M_f and the path of M_f that, starting from such a input, allow to reach a possible solution y. Hence M computes the function f_M:

$$f_M(\mathbf{x})(y)) = \begin{cases} \prod_{1 \leq i \leq n} (\Sigma_{k \in CC_M(k,z)} \frac{1}{2^{|k|}}) * \Sigma_{p \in CC_M(z,y)} \frac{1}{2^{|p|}} & \text{if } y \in \mathcal{FC}_M^{\mathbf{x}} \\ 0 & \text{otherwise} \end{cases}$$

This concludes the proof. □

Lemma 5.3 (Primitive Recursion and Turing-Computability) *Given Turing-Computable* $g : \mathbb{N}^{k+2} \to \mathbb{P}_{\mathbb{N}}$ *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 Turing-Computable.*

Proof: We give an informal proof. We define a PTM, say M, working on 5 tapes. The first tape is the input tape, on the next tape M computes the count down of our $k + 1^{th}$ variable, on the third tape M computes g, on the fourth tape M computes the function f, and in the last it saves the result. The machine M operates as follows:

1. it copies in the second tape the $k + 1^{th}$ element of the input, and then it copies on the fourth tape the first k elements of the input;
2. it computes f and saves the result on the last tape;
3. it verifies if the second tape is 0. In this case M stops and the last tape contains the result, otherwise it copies the first k elements of the input from the first tape in the third tape and then it copies the result present in the last tape on the third tape;
4. M decrements of the value on the second tape;
5. it computes g on the third tape and save the result on the last tape;

6. it returns to the step 3.

In this case we can see M as the composition of content of $k+1^{th}$ element in input of g and of f. This concludes the proof.

□

Lemma 5.4 (Minimization and Turing-Computability) *Given Turing-Computable* $f : \mathbb{N}^{k+1} \to \mathbb{P}_{\mathbb{N}}$*, the function* $\mu\ f : \mathbb{N}^k \to \mathbb{P}_{\mathbb{N}}$ *is itself Turing-Computable.*

Proof: We give an intuitive proof. We take a PTM, said M with 4 tapes. The first tape is the input tape, on the next tape M saves one element that we name y, 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 in the second tape 0 and it copies on the third tape the input and the value y (present in the second tape);
2. it computes on the third tape the function f and saves the result on the last tape;
3. it verifies if 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 y;
4. it copies in the third tape the input and y;
5. it returns to the step 3.

Assuming that on the last tape it is saved the result y of the function μ, we can see that M computes the distribution:

$$f(\mathbf{i}, y)(0) \cdot (\prod_{x<y} (\sum_{k>0} f(\mathbf{i}, x)(k)))$$

In fact, by hypothesis we have a PTM M_f which computes f and whose output distribution is f_{M_f} defined as follows:

$$f_{M_f}(\mathbf{z}, s)(0) = \begin{cases} \Sigma_{p \in CC_M((\mathbf{z}s),0)} \frac{1}{2^{|p|}} & \text{if } 0 \in \mathcal{FC}_M^{\mathbf{z},s} \\ 0 & \text{otherwise} \end{cases}$$

The machine M that we have constructed starts the machine M_f, with input 0. If the machine M_f returns 0 then the machine M ends its work and returns the probability associated to the computation which lead to the result 0. On the other hand, if M_f returns

a value x different from 0 then M restarts the computation of the machine M_f with input 1 , 2 and so until the computation of M_f returns the value 0. Thus the computation of M produces a distribution which associates to y the probability that the result of M_f with input y is 0 and is different from 0 on all inputs $x < y$. Hence the distribution f_M which is the output of M is the following:

$$f_M(\mathbf{z})(y) = \begin{cases} \Sigma_{p \in CC_M((\mathbf{z}s),0)} \frac{1}{2^{|\overline{p}|}} \cdot \Sigma_{t \in CC_M((\mathbf{z}k),x)} \frac{1}{2^{|\overline{t}|}} & \text{if } 0 \in \mathcal{FC}_M^{\mathbf{z},s}\mathbf{z}, s, k < s \text{ and } x \neq 0 \\ & \text{otherwise} \end{cases}$$

This concludes the proof.

□

Hence we can prove the following theorem, showing that Probabilistic Recursive Functions are computable by a probabilistic Turing machine.

Theorem 5.1 $\mathscr{PR} \subseteq \mathscr{PC}$

Proof: Immediate from Lemmata 5.1, 5.2, 5.3 and 5.4. □

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 the encoding the transition functions 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 by which one determines the *number* of transition steps of M necessary to reach a final configuration, if such number exists. This number can then be fed into 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. Given our definition of minimization which involves distributions, this is delicate, since we have to define a suitable function on the PTM computation tree to be minimized.

In order to adapt the classic proof, we need to formalize the notion of a *computation tree* which represents all computation paths corresponding to 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 $\mathcal{IN}_M^x$ 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 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.

We can naturally associate a probability with each node, 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 ε. 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 number $b \cdot 1$. 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 for x will be denoted as $CT_M(x)$

We give now a more explicit description of the constructions described above. 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 $enc : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ 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 to represent positive rational numbers as pairs of natural numbers in the obvious way and is recursive.

It is now time to define a few notions on computation trees

Definition 5.9 (Computation Trees and String Probabilities) *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 5.10 (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, however, what we do is precisely to show that this is the case.

In Figure 5.1 we show an example of computation tree $CT_M(x)$ for an hypothetical PTM M and an input x. The leaves, depicted as red nodes, represent the final configurations of the computation. So, for example, $PC_M(x, C) = 1$, 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

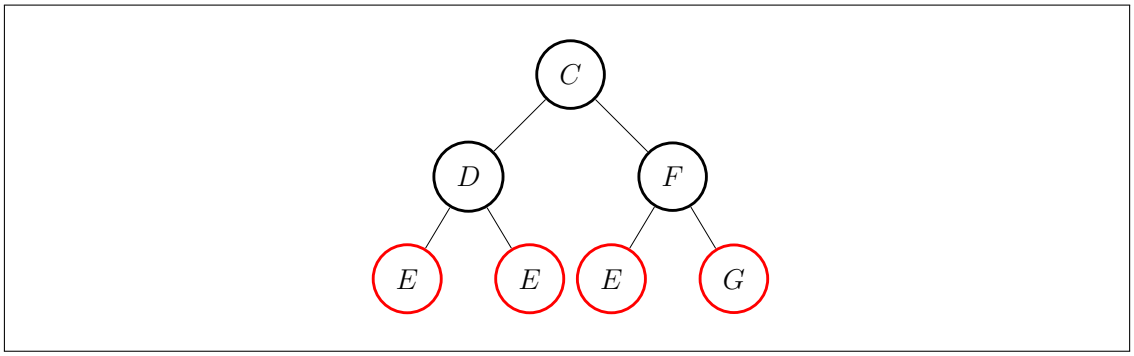

Figure 5.1: An Example of a Computation Tree

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 5.11 *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:*

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

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

Note that, according to 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 5.2 *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 $\{0^{PT^1_M(x,y)}, 1^{PT^0_M(x,y)}\}$ to each pair of natural numbers (x,y). In Figure 5.2, we give the quantities we have just defined for the tree from Figure 5.1. Each internal node

is associated with the same distribution $\{0^0, 1^1\}$. Only the leaves are associated with non-trivial distributions. As an example, the distribution associated to the node 10 is $\{0^{\frac{1}{2}}, 1^{\frac{1}{2}}\}$, because we have that

$$\begin{aligned} PT^0_M(x, \overline{10}) &= \frac{PT_M(x, \overline{10})}{\prod_{k<\overline{10}} PT^1_M(x, k)} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{01}) \cdot PT^1_M(x, \overline{00}) \cdot PT^1_M(x, \overline{1}) \cdot PT^1_M(x, \overline{0}) \cdot PT^1_M(x, \overline{\varepsilon})} \\ &= \frac{1}{4 \cdot PT^1_M(x, \overline{01}) \cdot PT^1_M(x, \overline{00})}. \end{aligned}$$

As it can be easily verified, $PT^1_M(x, \overline{00}) = \frac{3}{4}$, while $PT^1_M(x, \overline{01}) = \frac{2}{3}$. Thus, $PT^0_M(x, \overline{10}) = \frac{1}{2}$.

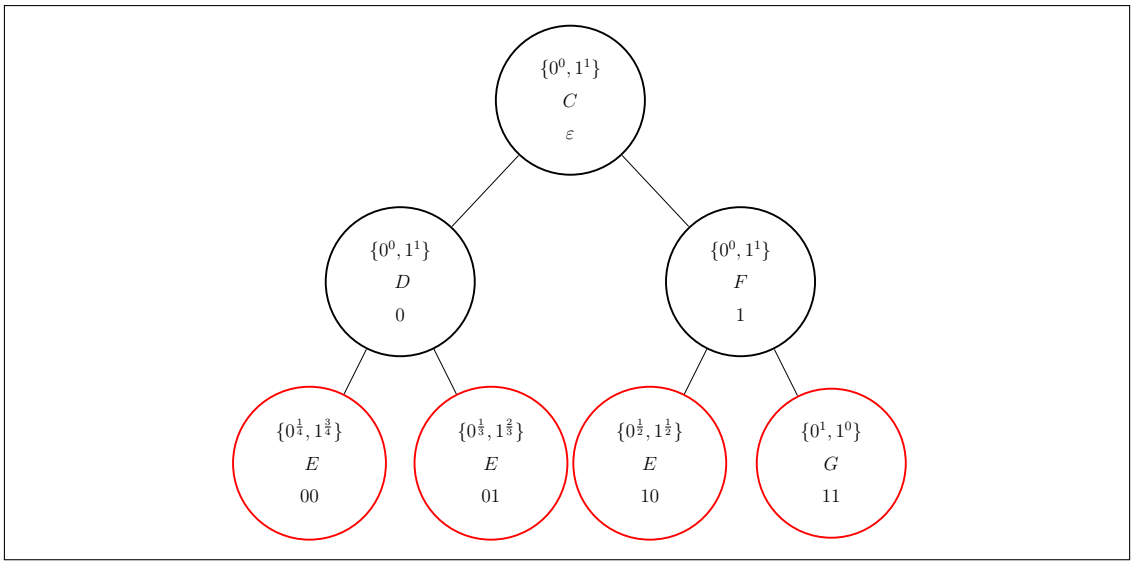

Figure 5.2: The Conditional Probabilities for the Computation Tree from Figure 5.1

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

Definition 5.12 *Given a PTM M, the function $PTC_M : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as*

follows

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

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

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

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 5.3 *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 $h = \mu\ (r \odot \Pi^2_1)$. 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 $c_i^q \in \{0,1\}$ (i.e., c_i^q is the i-th element of the binary representation of q). Moreover, a function computing such a c_i^q from q and i is partial recursive. Hence we can define $b : \mathbb{N} \times \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ as follows

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

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:

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

From the definition of composition, it follows that

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

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 5.4 *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 5.14 *The function $\mathcal{CF}_M : \mathbb{N} \to \mathbb{P}_{\mathbb{N}}$ is defined as follows*

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

Proposition 5.5 *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 z corresponds to a leaf, then:

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

If, however, z does not correspond to a leaf, then:

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

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 $CT_M(x)$. We can now prove the desired result:

Theorem 5.2 $\mathscr{PC} \subseteq \mathscr{PR}$.

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 5.1 $\mathscr{PR} = \mathscr{PC}$

Proof: Immediate from Theorem 5.2, observing that $\mathscr{PR} \subseteq \mathscr{PC}$ (this implication is easy to prove).

□

The way we prove Corollary 5.1 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 function.

Chapter 6

Probabilistic Implicit Complexity

In this chapter we present some complexity probabilistic classes and then we provide a characterization of the probabilistic functions which can be computed in polynomial time by using an algebra of functions acting on word algebras.

More precisely, we define a type system inspired by Leivant's notion of tiering [21] (seen in Chapter 3) witch permits to rule out functions having a too-high complexity, thus allowing to isolate the class of *predicative probabilistic functions*.

Our main result in this chapter shows that the class $\mathscr{PPC}$ of probabilistic functions which can be computed by a PTM in *polynomial* time is equivalent to the class of predicative probabilistic functions.

6.1 Probabilistic Complexity Classes

We introduce now some complexity classes that will be needed in the following discussion. The most general class is called **PP**, which stands for Probabilistic Polynomial Time. It consists of problems for which there is a random algorithm solving the problem in polynomial time with probability greater than $\frac{1}{2}$. There is no restriction about how much the probability have to be greater than $\frac{1}{2}$. The class **RP** is the class of problems for which exists a polytime randomized algorithm that answers without errors if the right answer is "no" and answers with an error $e < \frac{1}{2}$ if the right answer is "yes" [26]. Formally, let M be a PTM for the language $L \in$ **RP**: for any possible input s we have that

if $s \in L$ then $P(M(s) = yes) > \frac{1}{2}$ and if $s \notin L$ then $P(M(s) = no)) = 1$. Class **co-RP** is the complementary class of **RP**. Hence it is easy see as this class is defined as follows : let M be a PTM for the language $L \in$ **co-RP** on all possible input s, if $s \in L$ then $P(M(s) = yes) = 1$ and if $s \notin L$ then $P(M(s) = no) \leq \frac{1}{2}$. The class **ZPP** (Zero-error Probabilistic Polynomial Time) is the class of problems for which a Las Vegas algorithm exists, solving them in polynomial time [26, 1]. Finally we see **BPP**, witch is the set of decision problems solvable by a probabilistic Turing machine in polynomial time with an error $e \leq \frac{1}{3}$ for all possible answers. Hence let M be a PTM for the language $L \in$ **BPP** on all possible input s, if $s \in L$ then $P(M(s) = yes) > \frac{2}{3}$ and if $s \notin L$ then $P(M(s) = no) \leq \frac{1}{3}$.

There is however an intrinsic difficulty in giving *implicit* characterizations of probabilistic classes like **BPP** or **ZPP**: the latter 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 class much harder and, ultimately, prevents from deriving implicit characterization of the classes above. Our point of view is different: we do not see probabilistic algorithms as devices computing functions of the same kind as those computed by deterministic algorithms, but we see probabilistic algorithms as devices outputting distributions.

6.2 Function Algebra on Strings

The constructions from Chapter 5 can be easily generalized to a function algebra on strings in a given alphabet Σ, which themselves can be seen as a *word algebra* $\mathbb{W}$. Base functions include a function computing the empty string, called ε, 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 Σ to a given string, called again r. 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 anyway 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 6.1 (String Distribution) *A* 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 distribution on* $\mathbb{W}$.

The functions that we consider in our algebra have domain $\mathbb{W}^k$ and codomain $\mathbb{P}_{\mathbb{W}}$. The idea, as usual, is that if $f(x)(y) = p$ then y is the output obtained for the input x with probability p. Base functions include a function computing the empty string, denoted by ε, and concatenation with any character $a \in \Sigma$, denoted by c_a. Formally we define these functions as follows:

$$\epsilon(v)(w) = \begin{cases} 1 & w = \epsilon \\ 0 & otherwise \end{cases}$$

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

Note that, for all $v \in \mathbb{W}$, if the length of v is k then the length of the word w obtained after the application of one of the constructors c_a is $k + 1$ with probability 1. Projections remain available in the usual form. Indeed the function $\Pi^n_m : \mathbb{W}^n \to \mathbb{P}_{\mathbb{W}}$ is defined as follows:

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

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

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

Also in this case, for all $v \in \mathbb{W}$, if the length of v is k then the length of the word w obtained after the application of one of the constructors c_i is $k+1$ with probability 1. The

different probabilities (smaller than 1) associated to the different cases in the definition reflect the different constructors used to obtain w.

Next we recall the concept of composition and recurrence introduced in Definition 5.4 and Definition 5.5 and we instantiate it to the case of our algebra. We first introduce the *Generalized Composition* of functions $f : \mathbb{W}^n \to \mathbb{P}_{\mathbb{W}}$, $g_1 : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}, \ldots, g_n : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ as the function $f \odot (g_1, \ldots, g_n) : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ defined as follows:

$$((f \odot (g_1, \ldots, g_n))(\mathbf{i}))(y) = \sum_{x_1,\ldots,x_n \in \mathbb{W}} \left(f(x_1, \ldots, x_n)(y) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{i})(x_i) \right).$$

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

$$f(\epsilon, \mathbf{y}) = g_\epsilon(\mathbf{y})$$

$$f(a_a \cdot w, \mathbf{y}) = g_a(f(w, \mathbf{y}), w, \mathbf{y})$$

where $f : \mathbb{W} \times \mathbb{W}^m \to \mathbb{P}_{\mathbb{W}}$, $g_a : \mathbb{W} \times \mathbb{W} \times \mathbb{W}^m \to \mathbb{P}_{\mathbb{W}}$, for all $a \in \Sigma$. We use $f = rec(g_\epsilon, \{g_a\}_{\{}a \in \Sigma\})$ as a shorthand for previous definitions of recurrence. The following construction is redundant in presence of primitive recursion, but becomes essential when predicatively restricting it.

Definition 6.2 (Case Distinction) *If $g_\varepsilon : \mathbb{W}^k \to \mathbb{P}_{\mathbb{W}}$ and for every $a \in \Sigma$, $g_a : \mathbb{W}^{k+1} \to \mathbb{P}_{\mathbb{W}}$, we define a function $h : \mathbb{W}^{k+1} \to \mathbb{P}_{\mathbb{W}}$ by* case distinction *by stipulating that $h(\varepsilon, \mathbf{y}) = g_\varepsilon(\mathbf{y})$ while $h(a \cdot w, \mathbf{y}) = g_a(w, \mathbf{y})$. The function h is denoted as $case(g_\varepsilon, \{g_a\}_{a \in \Sigma})$.*

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

Definition 6.3 *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 function g_a^j (where $j \in \{1, \ldots, n\}$ and $a \in \Sigma$ if the following holds for every j and for every a:*

$$f^j(a \cdot v, \mathbf{w}) = g_a^j(f^1(v, \mathbf{w}), \ldots, f^n(v, \mathbf{w}), v, \mathbf{w})$$

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

$$\frac{}{\epsilon \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{r \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{c_i \triangleright \mathbb{W}_k \to \mathbb{W}_k} \qquad \frac{}{\Pi^k_n \triangleright \mathbb{W}_{n_1} \times \cdots \times \mathbb{W}_{n_n} \to \mathbb{W}_{n_k}}$$

$$\frac{\{g_i \triangleright \mathbb{W}_{s_1} \times \cdots \times \mathbb{W}_{s_r} \to \mathbb{W}_{m_i}\}_{1 \leq i \leq l} \qquad f \triangleright \mathbb{W}_{m_1} \times \cdots \times \mathbb{W}_{m_p} \to \mathbb{W}_l}{f \odot (g_1, \ldots, g_l) \triangleright \mathbb{W}_{s_1} \times \cdots \times \mathbb{W}_{s_r} \to \mathbb{W}_l}$$

$$\frac{g_\varepsilon \triangleright \mathbf{W} \to \mathbb{W}_l \qquad \{g_a \triangleright \mathbb{W}_k \times \mathbf{W} \to \mathbb{W}_l\}_{a \in \Sigma}}{case(g_\varepsilon, \{g_a\}_{a \in \Sigma}) \triangleright \mathbb{W}_k \times \mathbf{W} \to \mathbb{W}_l} \qquad \frac{g_\varepsilon \triangleright \mathbf{W} \to \mathbb{W}_k \qquad m > k \qquad \{g_a \triangleright \mathbb{W}_k \times \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k\}_{a \in \Sigma}}{rec(g_\epsilon, \{g_a\}_{a \in \Sigma}) \triangleright \mathbb{W}_m \times \mathbf{W} \to \mathbb{W}_k}$$

Figure 6.1: Tiering as a Typing System

Example 6.1 *P*revious definition allows to define, for instance, two functions f^1 and f^2 over a word algebra with $\Sigma = \{a, b\}$, as follows:

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

□

6.3 Tiering as a Typing System

Now we define our type system which will be used to introduce the definition of the class of *predicative probabilistic functions* and therefore to obtain our complexity result. The type system is inspired by the tiering approach of Leivant [21] and the formulation of predicative (or ramified) recursion over a probabilistic word algebra $\mathbb{W}$ derives from the definition of recurrence, suitably restricted using types.

As previously mentioned, the idea behind tiering consists in working with denumerable many copies of the underlying algebra $\mathbb{W}$, each indexed by a natural number $n \in \mathbb{N}$ and denoted by $\mathbb{W}_n$. So, given a function $f : \mathbb{W}^k \to \mathbb{W}$ type judgments take the form $f \triangleright \mathbb{W}_{n_1} \times \ldots \times \mathbb{W}_{n_k} \to \mathbb{W}_m$. In the following, with slight abuse of notation, $\mathbf{W}$ stands for any expression of the form $\mathbb{W}_{i_1} \times \cdots \times \mathbb{W}_{i_j}$.

Typing rules are given in Figure 6.1.

The rules for the basic functions and for composition are immediate. The interesting rules here is the one for recursion: The idea here is that, when generating functions by primitive recursion, one passes from a level (tier) m for the domain to a *strictly* lower level k for the result. This predicative constraint ensures that recursion does not causes any exponential blowup, simply because the way one can *nest* primitive recursive definitions one inside the other is severely restricted. Please notice that case distinction, although being typed in a similar way, does *not* require the same constraints.

More precisely, the class $\mathscr{PT}$ of all predicatively recursive functions is defined as follows. 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 6.1 are said to be *predicatively recursive*.

Definition 6.4 *The class $\mathscr{PT}$ of* predicatively probabilistic recursive functions *is the smallest class of functions that contains the basic functions and is closed under the operation of General Composition (Definition 5.4), Primitive Recursion (Definition 5.5), Case Distinction (Definition 6.2) and such that each function can be given a type through the rules in Figure 6.1.*

Next we give the definition of the class of simultaneous recursive functions $\mathscr{SR}$ since this will be needed for the proof of the main theorem of this chapter.

Definition 6.5 *The class $\mathscr{SR}$ of* simultaneous recursive functions *is the smallest class of functions that contains the basic functions and is closed under the operation of General Composition (Definition 5.4), Simultaneous Recursion (Definition 6.3), Case Distinction (Definition 6.2 and such that each function can be given a type through the rules in Figure 6.1, plus the rule below:*

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

The following theorem shows that the class of predicatively recursive functions coincides with the class of probabilistic functions which can be computed by PTMs in polynomial time. The proof is in the next sections.

Theorem 6.1 $\mathscr{PT} = \mathscr{PPC}$.

Characterizing complexity classes of *probabilistic* functions allows to deal implicitly with concepts like that of a *polynomial time samplable* distribution [3, 16], which is a family $\{\mathcal{D}_n\}_{n\in\mathbb{N}}$ of distributions on strings such that a polytime randomized algorithm produces $\mathcal{D}_n$ when fed with the string 1^n. By Theorem 6.1, each of them is computed by a function in $\mathscr{PT}$ and, conversely, any predicatively recursive probabilistic function computes one such family.

6.4 Functions computed by Probabilistic Turing Machines in Polynomial Time equals Predicative Probabilistic Functions

The proof of Theorem 6.1 proceed essentially by proving (in the next sub-sections) the following points, from which the thesis can be inferred:

- First one can prove, by a careful encoding, that a form of *simultaneous primitive recursion* is available in predicative recursion.
- Then PTMs can be shown equivalent, in terms of expressivity, to probabilistic *register* machines; going through register machines has the advantage of facilitating the last two steps.
- Thirdly, any function definable by predicative recurrence can be proved computable by a polytime probabilistic register machine.
- Lastly, one can express any polytime probabilistic computable function in terms of a predicatively recursive function, by making use of simultaneous recurrence.

6.4.1 Simultaneous Primitive Recursion and Predicative Recursion

We can encode Simultaneous Primitive Recursion in Predicative Recursion.

According to Definition 6.3, 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

$$\begin{aligned}
f^1(\varepsilon, \mathbf{x}) &= g^1_\varepsilon(\mathbf{x}); \\
f^1(a \cdot w, \mathbf{x}) &= g^1_a(f^1(w, \mathbf{x}), f^2(w, \mathbf{x}), w, \mathbf{x}); \\
f^1(b \cdot w, \mathbf{x}) &= g^1_b(f^1(w, \mathbf{x}), f^2(w, \mathbf{x}), w, \mathbf{x}); \\
f^2(\varepsilon, \mathbf{x}) &= g^2_\varepsilon(\mathbf{x}); \\
f^2(a \cdot w, \mathbf{x}) &= g^2_a(f^1(w, \mathbf{x}), f^2(w, \mathbf{x}), w, \mathbf{x}); \\
f^2(b \cdot w, \mathbf{x}) &= g^2_b(f^1(w, \mathbf{x}), f^2(w, \mathbf{x}), w, \mathbf{x})
\end{aligned}$$

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:

$$\begin{aligned}
\widetilde{f}(\varepsilon, \mathbf{x}) &= \langle g^1_\epsilon(\mathbf{x}), g^2_\epsilon(\mathbf{x}) \rangle; \\
\widetilde{f}(a \cdot w, \mathbf{x}) &= \langle g^1_a(\widetilde{f}(w, \mathbf{x}), w, \mathbf{x}), g^2_a(\widetilde{f}(w, \mathbf{x}), w, \mathbf{x}) \rangle; \\
\widetilde{f}(b \cdot w, \mathbf{x}) &= \langle g^1_b(\widetilde{f}(w, \mathbf{x}), w, \mathbf{x}), g^2_b(\widetilde{f}(w, \mathbf{x}), w, \mathbf{x}) \rangle.
\end{aligned}$$

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, i. e. a type in which the tier of the argument(s) is the same as 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 [21].

6.4.2 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 register machines to the probabilistic case. Again, the way register machines are defined closely follows Leivant's proof [21].

Definition 6.6 (Probabilistic Register Machine) *A* probabilistic register machine (PRM) *consists of a finite set of registers* $\Pi = \{\pi_1, \cdots, \pi_r\}$ *and a sequence of instructions, called* program*. Each register* π_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 five forms*

$$\varepsilon(\pi_d); \qquad c_a(\pi_s)(\pi_d); \qquad r_a(\pi_s)(\pi_d); \qquad p_a(\pi_s)(\pi_d); \qquad j(\pi_s)(m);$$

where π_s, π_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 $\varepsilon(\pi_d)$ stores in the register π_d the empty string and then transfer the control to the next instruction.
- The instruction $c_a(\pi_s)(\pi_d)$ stores in the register π_d the term $a \cdot w$, where w is the string contained in the register π_s. It then transfers the control to the next instruction.
- The instruction $p_a(\pi_s)(\pi_d)$ is the predecessor instruction, which stores in the register π_d the string resulting from erasing the leftmost character a from the string contained in π_s, if any. The control is then transferred to the next instruction.
- If w is the string contained in π_d, the instruction $r_a(\pi_s)(\pi_d)$ stores w in the register π_d (with probability $\frac{1}{2}$) or the string $a \cdot w$ (with probability $\frac{1}{2}$).
- The instruction $j(\pi_s)(m)$ transfers the instruction to the m-th instruction if π_s contains a non-empty string, while it goes to the next instruction otherwise.

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

Definition 6.7 (Configuration of a PRM) *Let* R *be a PRM as in Definition 6.6, and let* Σ *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* π_i*;*
- $n \in \mathbb{N}$ *is the index of the next instruction to be executed.*

We define the set of all configurations with $\mathcal{CR}_R$*. If* $n = 1$ *we have an initial configuration for an* 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* π_1*.*

Next we show how previous instructions allow to change a configuration.

$\epsilon(\pi_s)(\pi_l)$ If we apply the instruction $\epsilon(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{l-1}, v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$c_a(\pi_s)(\pi_l)$ If we apply the instruction $c_a(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{l-1}, a \cdot v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$p_a(\pi_s)(\pi_l)$ If we apply the instruction $p_a(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, a{\cdot}v_s, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, a \cdot v_s, \ldots, v_{l-1}, v_s, v_{l+1}, \ldots, v_r, n+1\rangle$.

$r_a(\pi_s)(\pi_l)$ If we apply $r_a(\pi_s)(\pi_l)$ to the configuration $\langle v_1, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_r, n{+}1\rangle$ with probability $\frac{1}{2}$ and the configuration $\langle v_1, \ldots, v_{l-1}, a{\cdot} v_s, v_{l+1}, \ldots, v_r, n+1\rangle$ with probability $\frac{1}{2}$.

$j(\pi_s)(\mathbf{m})$ If we apply $j(\pi_s)(m)$ to the configuration $\langle v_1, \ldots, a \cdot v_s, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_s, \ldots, v_r, m\rangle$; if we apply $j(\pi_s)(m)$ to the configuration $\langle v_1, \ldots, v_{s-1}, \epsilon, v_{s+1}, \ldots, v_r, n\rangle$, we obtain the configuration $\langle v_1, \ldots, v_{s-1}, \epsilon, v_{s+1}, \ldots, v_r, n{+}1\rangle$.

First we observe that the meaning of a PRM R program can be defined by way of two functions δ_0 and δ_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 did for PTMs, we can define a (complete) partial order with carrier $\mathcal{CEV}_R$. And hence, we can define a functional FR_R on $\mathcal{CEV}_R$ 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 6.8 *Given a PRM R, we define a functional $FR_R : \mathcal{CEV}_R \to \mathcal{CEV}_R$ as:*

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

Using similar arguments to those in the proofs of Proposition 4.5 and Theorem 4.1, we can show that there exists the least fixpoint of the functional defined above. Such a least fixpoint, once composed with a function returning $\mathcal{INR}^{\mathbf{s}}_R$ from $\mathbf{s}$, is the *function computed by the register machine* R and it is denoted by $\mathcal{IO}_R : \Sigma^* \to \mathbb{P}_{\Sigma^*}$.

Next Lemma shows the relations between PTMs and PRMs.

Lemma 6.1 *PTMs are linear time reducible to a PRMs, and PRMs over $\mathbb{W}$ are poly-time reducible to PTMs.*

Proof: A single tape PTM M can be simulated by a PRM R that has tree registers. A configuration $\langle w, a, v, s\rangle$ of M can be coded by the configuration R $\langle [w^r, a, v], s\rangle$ 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 δ_0 and δ_1 we use and instructions ε, r_a and j, plus a dedicated register π_{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 R runs in time $O(n^k)$, then M_R runs in time $O(n^{2k})$. We can then conclude by remembering that Turing machine with multiple tapes can be simulated by single-tape Turing machine with a polynomial slowdown. □

6.4.3 Polytime Soundness

In this section we prove that any function definable by predicative recurrence is computable by probabilistic register machines working in polynomial time (dubbed PPRMs in the following). Let $\mathscr{PPR}$ be the class of functions computed by machines of this kind.

In view of Lemma 6.1, showing that predicative recurrence can be simulated by a PPRM suffices. This result is not difficult and is proved by exhibiting a PPRM which computes any function f such that $f \triangleright \mathbf{W} \to \mathbb{W}_m$.

Proposition 6.1 *If $f \triangleright \mathbf{W} \to \mathbb{W}_m$, then there is PPRM that computes f.*

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

- We need to show that for every basic function defined in Definition 5.2 we can construct a PPRM that computes such a function. The proof is immediate for functions, c_a, by observing that it is included in the set of PPRM operations. The function ε is simulated by using the instruction ϵ (on an empty register). The function Π is simulated by the instructions c_a, followed by p. Finally the function r_a can be simulated by the instructions r_a.
- If f is defined by composition, then we give an intuitive proof. We take a PPRM, said R_s with s registers. The first k registers have saved the input, the next $k \cdot n$ registers R_s computes the $g_1, \cdots, g_n$ functions. For each g_i the machine saves the result on the registers $(k \cdot n) + i$, with $1 \leq i \leq n$. These registers became the input registers for computing f. Finally in the last register is saved the result. R_s operates as follows:
  1. R_s copies all k registers on the $k \cdot n$ registers. The computational cost of this operation is $n \cdot k$, because it is implemented by the instruction $\epsilon(\pi_l)(\pi_s)$;
  2. R_s computes the respective functions g_i with $1 \leq i \leq n$ and saves the results in the registers $(k \cdot n) + i$. These functions are by hypothesis polynomial time computable;
  3. R_s computes the function f that by hypothesis polynomial time computable.

  Finally R_s computes the function $f \odot (g_1, \ldots, g_n)$ in time

$$
\begin{aligned}
z &= k \cdot n + \sum_{i=1}^{n} (max(|pi_i|)^{s_i} + v_i) + max(|pi_{(k \cdot n)+i}|)^t + w \\
&\leq (k \cdot n) + (n+1) \cdot (max(|pi_i|, |pi_{(k \cdot n)+i}|)^{max(s_i,t)} + max(v_i, w)) \\
&< (n+1) \cdot (y^m + q + k)
\end{aligned}
$$

  where pi denotes the element saved on the register, $max(|pi_i|, |pi_{(k \cdot n)+i}|) = y, max(s_i, t) = m$ and $max(v_i, w) = q$.

– The function Case Distinction is implemented by a PPRM, said R_{case}, that computes it as follows. The first $k+1$ registers contain the input and on the last register we have the result. R_{case} operates as follows:
  1. R_{case} applies the operation $j(\pi_{k+1})(\mathbf{m})$; if $pi_{k+1} = \epsilon$ the machine goes to the instruction $i+1$ where there is saved the first instruction in order to compute g_ε, otherwise the machine jump at the instruction m_a corresponding at the first (in pi_{k+1}) constructor function c_a and it saves the result on pi_{k+1} register. The instruction m_a is the first one which allows to compute g_a.
  2. R_{case} computes the function g_ε or g_a and it saves the result on the last register.

  Finally R_{case} computes the function Case Distinction in time z, and z is a polynomial time because:

$$z = c + \begin{cases} |s_\varepsilon|^{k_\varepsilon} + r_\varepsilon & \text{if } pi_{k+1} = \epsilon \\ |s_a|^{k_a} + r_a & \text{if } pi_{k+1} = p_a \end{cases}$$

  where c is the time constant used from the machine for computing the function j.
– We give an intuitive proof. We take a PPRM, said R_{rec}. The first $k+1$ registers contain the input, in the next k registers we save the input, in the $2k + 2^{th}$ register we have the result of intermediate computations, and in the last register we have the result. We assume that the $k+1^{th}$ input is saved inverted on the $k+1^{th}$ register. R_{rec} operates as follows:
  1. R_{rec} computes g_ε and saves the results in the $2k + 2^{th}$;
  2. R_{rec} applies an operation of $j(\pi_{k+1}), (\mathbf{m})$;
  3. if $pi_{k+1} = \epsilon$ the machine goes to the instruction $i+1$ where it saves the result on the last registers and then stop, otherwise the machine jump to the instruction m_a corresponding at the function predecessor p_a and it saves the result on pi_{k+1} register. m_a is the first instruction needed in order to compute g_a.
  4. R_{rec} jump at the instruction 2 if it is not stopped before.

  Finally R_{rec} computes the function $rec(f, g)$ in time z, and z is a polynomial time

because:

$$
\begin{aligned}
z &= c \cdot |p_{k+1}| + |p_{k+1}| \cdot \begin{cases} |s_\varepsilon|^{k_\varepsilon} + r_\varepsilon & \text{if } pi_{k+1} = \epsilon \\ |s_a|^{k_a} + r_a & \text{if } pi_{k+1} = p_a \end{cases} \\
&\leq max(|p_{k+1}|, |s_\varepsilon|, |s_a|) \cdot (c + max(r_\varepsilon, r_a)) + \\
&+ max(|p_{k+1}|, |s_\varepsilon|, |s_a|) \cdot max(|p_{k+1}|, |s_\varepsilon|, |s_a|)^{max(k_\varepsilon, k_a)} \\
&= x^t + x \cdot d
\end{aligned}
$$

where c is the time constant used from the machine for computes the function j, $x = max(|p_{k+1}|, |s_\varepsilon|, |s_a|)$, $t = max(k_\varepsilon, k_a) + 1$ and $d = c + max(r_\varepsilon, r_a)$.

□

6.4.4 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 [21]. The argument would go 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 $g : \Sigma^* \to \Sigma^*$, $p_g \triangleright \mathbb{W}_n \to \mathbb{W}_m$ for some n and m, this as a consequence of Leivant's result [21].
- Finally, one can observe that any polytime probabilistic function can be seen as a deterministic polytime function taking, as an additional input, a "long enough" sequence of random bits.

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

Lemma 6.2 (Polytime Random Sequences) *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 such that $f_{k,n} \triangleright \mathbb{W}_m \to \mathbb{W}_l$ 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 $randext$ and p_q. □

The proof of the next Lemma is the Leivant's result reported in [21].

Lemma 6.3 (Polytime Computable Function and Ramified Recurrence Function) *For every polynomial time computable function $g : \Sigma^* \to \Sigma^*$ (non probabilistic) there exists a $p_g \triangleright \mathbb{W}_n \to \mathbb{W}_m$ for some n and m.*

Then we have the following.

Theorem 6.2 $\mathscr{PPR} \subseteq \mathscr{PT}$.

Proof: Consider any probabilistic polytime Turing machine M. From the discussion at the beginning of this section, it is clear that the probabilistic function compted 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 6.2. Since the three function can be given type, their composition itself can. □

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

Corollary 6.1 $\mathscr{PPR} \subseteq \mathscr{PT}$.

Proof: Immediate from Theorem 6.2, and Proposition 6.1.

□

A more direct way to prove polytime completeness consists in showing how single-tape PTMs can be encoded into ramified recurrence. This can be done relatively easily by exploiting simultaneous recursion.

Chapter 7

Conclusions

In this thesis we made 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 for this study is genuinely foundational: working with probabilistic functions allows us to better understand the nature of probabilistic computation on the one hand, 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 introducing a natural generalization of Kleene's partial recursive functions which includes, among initial functions, one that behaviors as to identity or successor with probability $\frac{1}{2}$. We then prove the equi-expressivity of the obtained algebra and the class of functions computed by PTMs. In the second part of the thesis, we investigate the relations existing between our recursion-theoretical framework and sub-recursive 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.

As previously mentioned, the main aim of his work is foundational as it tries to capture the very essence of probabilistic computation. Nevertheless our results can have relevant practical applications, as probabilistic computability is more and more important in computer science and probabilistic models have many applications areas, including natural language processing, robotics, computer vision, and machine learning. A better

direct understanding of the functions computed by probabilistic machine, without reducing them to deterministic devices, could provide better insight on the expressive power of probabilistic models and therefore could have practical applications.

Our study could be extended in several ways. Remaining in the realm of probabilistic computation, a first extension could be the definition of a probabilistic Kleene's normal form. In Chapter 5 we have seen that the class of probabilistic *recursive* functions is the same as that one of probabilistic *computable* functions. In our proof we needed the minimization twice. This is not compatible with a Kleene's normal form, where one has to use the minimization only onece. Thus a possible extension of our work would be to modify the minimization in order to obtain a probabilistic normal form. Also the investigation of the distributions computed by a probabilistic lambda calculus could be a significant extension. Another possible further study could consist in imposing appropriate constraints on our algebra in order to define specific classes of Probabilistic Problems such as **ZPP** or **BPP** classes. This could be a step in the direction of defining and studying new probabilistic complexity classes in the area of ICC. On a more general level, 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 generalizing the ones described in this paper, given that qubits (the quantum analogues of classical bits) cannot be copied nor erased.

Index

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

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

[3] A. Bogdanov and L. Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1), 2006.

[4] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

[5] H. Buhrman and R. de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21 – 43, 2002. Complexity and Logic.

[6] D. Comaniciu, V. Ramesh, and P. Meer. Kernel-based object tracking. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(5):564–577, 2003.

[7] N. Cutland. *Computability An introduction to recursive function theory*. Cambridge University Press, 1980.

[8] U. Dal Lago. A short introduction to implicit computational complexity. In N. Bezhanishvili and V. Goranko, editors, *Lectures on Logic and Computation*, volume 7388 of *Lecture Notes in Computer Science*, pages 89–109. Springer Berlin Heidelberg, 2012.

[9] U. Dal Lago and P. P. Toldin. A higher-order characterization of probabilistic polynomial time. In *Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2012.

[10] U. Dal Lago and S. Zuppiroli. Probabilistic recursion theory and implicit computational complexity. In *Informal proceedings of the 2014 Conference on CiE2014*, 2014. Computability in Europe.

[11] U. Dal Lago and S. Zuppiroli. Probabilistic recursion theory and implicit computational complexity. In *Theoretical Aspects of Computing – ICTAC 2014*, Lecture Notes in Computer Science. Springer, 2014.

[12] K. De Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabilistic machines. *Automata studies*, 34:183–198, 1956.

[13] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[14] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[15] M. Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.

[16] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2000.

[17] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[18] M. Hofmann. A type system for bounded space and functional in-place update. In *Programming Languages and Systems*, pages 165–179. Springer, 2000.

[19] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[20] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.

[21] D. Leivant. Ramified recurrence and computational complexity i: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Springer, 1995.

[22] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. In *Typed Lambda Calculi and Applications*, pages 274–288. Springer, 1993.

[23] M. Li and P. M. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

[24] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[25] Y. N. Moschovakis. What is an algorithm. *Mathematics unlimited–2001 and beyond*, pages 919–936, 2001.

[26] C. H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

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

[28] M. O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[29] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.

[30] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

[31] E. S. Santos. Probabilistic Turing machines and computability. *Proceedings of the American Mathematical Society*, 22(3):704–710, 1969.

[32] E. S. Santos. Computability by probabilistic turing machines. *Transactions of the American Mathematical Society*, 159:165–184, 1971.

[33] D. S. Scott. Domains for denotational semantics. In *Automata, languages and programming*, pages 577–610. Springer, 1982.

[34] S. Thrun. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, pages 1–35, 2002.